

Teradata Vantage™ - SQL Data Definition Language

Detailed Topics

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 Detailed Topics | 9 |
| Changes and Additions | 9 |
| Chapter 2: ALTER FUNCTION - ALTER PROCEDURE | 10 |
| ALTER FUNCTION (External Form) | 10 |
| ALTER METHOD | 14 |
| ALTER PROCEDURE (External Form) | 17 |
| ALTER PROCEDURE (SQL Form) | 21 |
| Chapter 3: ALTER TABLE | 25 |
| ALTER TABLE (Basic Table Parameters) | 25 |
| ALTER TABLE (MODIFY Option) | 65 |
| ALTER TABLE (REVALIDATE Option) | 104 |
| ALTER TABLE (Set Down and Reset Down Options) | 111 |
| ALTER TABLE TO CURRENT | 113 |
| Chapter 4: ALTER TYPE | 121 |
| Adding Attributes To a Structured UDT | 121 |
| Workaround for Adding More Than 512 Attributes to a Structured UDT | 122 |
| Dropping Attributes From a Structured UDT | 122 |
| Adding Methods To a UDT | 123 |
| Dropping Methods From a UDT | 124 |
| Altering Attributes Of A Structured UDT Definition | 125 |
| Altering The Method Signatures Of A Structured UDT Definition | 125 |
| Recompiling a UDT | 126 |
| Related Information | 126 |
| Chapter 5: BEGIN QUERY CAPTURE- COMMENT | 127 |
| BEGIN QUERY CAPTURE | 127 |
| BEGIN QUERY LOGGING | 129 |
| COLLECT STATISTICS (Optimizer Form) | 145 |
| COMMENT (Comment Placing Form) | 170 |
| Chapter 6: CREATE AUTHORIZATION - CREATE ERROR TABLE | 173 |
| CREATE AUTHORIZATION and REPLACE AUTHORIZATION | 173 |
| CREATE CAST and REPLACE CAST | 178 |
| CREATE DATABASE | 189 |
| CREATE ERROR TABLE | 192 |

| | |
|--|------------|
| Chapter 7: CREATE FUNCTION - CREATE GLOBAL TEMPORARY TRACE TABLE | 204 |
| CREATE FUNCTION and REPLACE FUNCTION (External Form) | 204 |
| CREATE FUNCTION (Table Form) | 250 |
| CREATE FUNCTION and REPLACE FUNCTION (SQL Form) | 257 |
| CREATE GLOBAL TEMPORARY TRACE TABLE | 276 |
| Chapter 8: CREATE HASH INDEX - CREATE ORDERING | 279 |
| CREATE HASH INDEX | 279 |
| CREATE INDEX | 289 |
| CREATE JOIN INDEX | 295 |
| CREATE MACRO and REPLACE MACRO | 345 |
| CREATE METHOD | 354 |
| CREATE ORDERING and REPLACE ORDERING | 364 |
| Chapter 9: CREATE PROCEDURE and REPLACE PROCEDURE (External Form) | 370 |
| Relationship Among UDFs, Table UDFs, and External Procedures | 370 |
| Usage Restrictions for External Procedures | 370 |
| Memory Considerations for INOUT Parameters | 371 |
| Recompiling an External Procedure | 371 |
| When To Run External Procedures in Unprotected Mode | 371 |
| Differences Between CREATE PROCEDURE and REPLACE PROCEDURE | 373 |
| Dynamic Result Sets | 373 |
| External Procedures and Large Objects | 374 |
| Teradata Unity Support for External Procedures | 375 |
| Parameter Names and Data Types | 375 |
| LANGUAGE Clause | 376 |
| SQL DATA ACCESS Options | 377 |
| External Data Access Clause | 377 |
| SQL SECURITY Privilege Options | 378 |
| External SQL Procedures and SQL | 378 |
| Restrictions on Declaring an C++ External Procedure | 379 |
| Rules for Creating a Java External Procedure | 379 |
| Details About the Function of Java External Procedures | 379 |
| JAR Files | 380 |
| Data Type Mapping Between SQL and Java | 381 |
| SQLSTATE Values for Java External Procedures | 382 |
| Java External Routine-Specific Dictionary Tables | 382 |
| SQLJ Database | 383 |
| CREATE PROCEDURE Dictionary Table Actions for Java Procedures | 384 |
| REPLACE PROCEDURE Dictionary Table Actions for Java Procedures | 385 |
| SQL DATA ACCESS Clause | 385 |
| PARAMETER STYLE Clause | 386 |
| External Body Reference Clause | 387 |
| EXTERNAL NAME Clause | 389 |

| | |
|---|------------|
| External String Literal | 390 |
| External Java Reference Strings | 392 |
| Client-Server External Procedure Code Specification | 393 |
| Include Name Clause | 393 |
| Library Name Clause | 394 |
| Object File Name Clause | 395 |
| Package Name Clause | 396 |
| Specifying the CLI Option for the EXTERNAL NAME Package Clause | 397 |
| Source File Name Clause | 398 |
| External String Literal Examples | 398 |
| External Procedure Default Location Paths | 399 |
| External Procedure and UDF .so Linkage Information | 401 |
| EXTERNAL SECURITY Clause | 401 |
| Related Information | 402 |
| Chapter 10: CREATE PROCEDURE and REPLACE PROCEDURE (SQL Form) | 403 |
| SQL Data Access Options | 403 |
| SQL SECURITY Privilege Options | 403 |
| Invocation Restrictions | 403 |
| Default Session Collation For Procedures | 404 |
| Memory Considerations for INOUT Parameters | 404 |
| Recompiling an SQL Procedure | 404 |
| Special Condition Handling for SIGNAL and RESIGNAL Statements | 404 |
| Details About Dynamic Result Sets | 409 |
| Rules and Limitations for Dynamic Result Sets | 413 |
| Teradata Unity Support for SQL Procedures | 414 |
| SQL Procedure Support for Transaction Query Bands | 415 |
| SQL Procedures and Proxy Connections | 415 |
| SQL Procedure Support For UDTs | 415 |
| Supported DDL Statements in SQL Procedures | 416 |
| Usage Considerations for DDL Statements in Procedures | 417 |
| Unsupported DDL Statements in SQL Procedures | 417 |
| Session Mode Impact on DDL Requests in Procedures | 418 |
| Supported DML Statements in SQL Procedures | 418 |
| Unsupported DML Statements in SQL Procedures | 419 |
| Supported DCL Statements in SQL Procedures | 419 |
| Unsupported DCL Statements in SQL Procedures | 419 |
| Session Mode Impact on DCL Statements in SQL Procedures | 419 |
| Rules for Using the PREPARE Statement In an SQL Procedure | 420 |
| Guidelines for Using the CURRENT_TIME and CURRENT_TIMESTAMP Functions in an SQL Procedure | 420 |
| SQL Multistatement Request Support in SQL Procedures | 421 |
| Utilities and APIs Supporting CREATE/REPLACE PROCEDURE | 422 |
| SQL Procedure Options | 423 |
| Guidelines for Manipulating LOBs in an SQL Procedure | 423 |

| | |
|--|------------|
| Procedure for Creating an SQL Procedure | 427 |
| Completing the Creation of an SQL Procedure | 428 |
| Aborting the Creation of an SQL Procedure | 428 |
| Processing an SQL CREATE PROCEDURE Request | 429 |
| Retrieving an SQL Procedure Body | 429 |
| Function of REPLACE PROCEDURE Requests | 429 |
| Significance of Platform and Session Mode For REPLACE PROCEDURE | 429 |
| Chapter 11: CREATE RECURSIVE VIEW and REPLACE RECURSIVE VIEW | 430 |
| Updatability of Recursive Views | 430 |
| The Concept of Recursion | 430 |
| Controlling the Cardinality Estimate For the Final Recursive Result Spool | 434 |
| All Recursive View Definitions Must Be Linear | 435 |
| Building a Recursive View | 436 |
| Stratification of Recursive View Evaluation | 437 |
| Specifying RECURSIVE For a Non-Recursive View | 438 |
| Seed Statement Component of the View Definition | 438 |
| Recursive Statement Component of the View Definition | 438 |
| You Can Specify Multiple Seed and Recursive Statements Within a Single Recursive View Definition | 439 |
| Recursive View Definition Restrictions | 440 |
| Referential Restrictions For Recursive Views | 443 |
| Step Building Logic for Recursive Queries | 443 |
| Preventing Infinite Recursion Due To Cyclic Data | 444 |
| Preventing Infinite Recursion With Acyclic Data | 447 |
| Breadth First and Depth First Searches | 449 |
| Breadth First Reporting | 450 |
| Depth First Reporting | 451 |
| Related Information | 451 |
| Chapter 12: CREATE TABLE Options | 452 |
| CREATE TABLE | 452 |
| CREATE TABLE (Table Kind Clause) | 454 |
| CREATE TABLE (Table Options Clause) | 460 |
| CREATE TABLE (Column Definition Clause) | 473 |
| Chapter 13: CREATE TABLE Indexes | 510 |
| CREATE TABLE (Index Definition Clause) | 510 |
| Chapter 14: CREATE TABLE Global and Temporary | 592 |
| CREATE TABLE (Global Temporary/Volatile Table Preservation Clause) | 592 |
| Chapter 15: CREATE TABLE AS | 594 |
| CREATE TABLE (AS Clause) | 594 |
| Chapter 16: CREATE TABLE Queue | 611 |

| | |
|--|------------|
| CREATE TABLE (Queue Table Form) | 611 |
| Chapter 17: CREATE TRANSFORM - CREATE VIEW | 631 |
| CREATE TRANSFORM and REPLACE TRANSFORM | 631 |
| CREATE TRIGGER/ REPLACE TRIGGER | 642 |
| CREATE TYPE (ARRAY/VARRAY Form) | 669 |
| CREATE TYPE (Distinct Form) | 681 |
| CREATE TYPE (Structured Form) | 692 |
| CREATE USER | 702 |
| CREATE VIEW and REPLACE VIEW | 709 |
| Chapter 18: DATABASE - DROP USER | 719 |
| DATABASE | 719 |
| DELETE DATABASE | 720 |
| DELETE USER | 723 |
| DROP FUNCTION | 725 |
| DROP HASH INDEX | 727 |
| DROP INDEX | 728 |
| DROP JOIN INDEX | 730 |
| DROP MACRO | 731 |
| DROP ORDERING | 732 |
| DROP STATISTICS (Optimizer Form) | 733 |
| DROP TABLE | 734 |
| DROP TRANSFORM | 736 |
| DROP USER | 737 |
| Chapter 19: END LOGGING - SET TIME ZONE | 741 |
| END QUERY LOGGING | 741 |
| FLUSH QUERY LOGGING | 743 |
| MODIFY DATABASE | 746 |
| MODIFY USER | 749 |
| RENAME FUNCTION (External Form) | 755 |
| RENAME FUNCTION (SQL Form) | 757 |
| RENAME PROCEDURE | 758 |
| RENAME TABLE | 759 |
| RENAME VIEW | 760 |
| REPLACE METHOD | 761 |
| REPLACE QUERY LOGGING | 763 |
| SET QUERY_BAND | 764 |
| SET SESSION CALENDAR | 790 |
| SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL | 792 |
| SET SESSION FUNCTION TRACE | 797 |
| SET TIME ZONE | 798 |

| | |
|--|------------|
| Chapter 20: SQL HELP Statements | 804 |
| About HELP Statements | 804 |
| Object Name and Title Data in HELP Reports | 808 |
| Data Type Codes | 813 |
| TVM Kind Codes | 815 |
| HELP CAST | 816 |
| HELP COLUMN | 820 |
| HELP CONSTRAINT | 828 |
| HELP DATABASE | 832 |
| HELP ERROR TABLE | 834 |
| HELP FUNCTION | 835 |
| HELP HASH INDEX | 838 |
| HELP INDEX | 839 |
| HELP JOIN INDEX | 842 |
| HELP MACRO | 843 |
| HELP METHOD | 848 |
| HELP PROCEDURE | 851 |
| HELP SESSION | 855 |
| HELP SESSION CONSTRAINT | 864 |
| HELP STATISTICS (Optimizer Form) | 865 |
| HELP STATISTICS (QCD Form) | 868 |
| HELP TABLE | 872 |
| HELP TRANSFORM | 877 |
| HELP TRIGGER | 879 |
| HELP TYPE | 881 |
| HELP USER | 889 |
| HELP VIEW | 891 |
| HELP VOLATILE TABLE | 895 |
| HELP (Online Form) | 896 |
| Chapter 21: SQL SHOW Statements | 899 |
| About SHOW Statements | 899 |
| SHOW <i>request</i> | 900 |
| SHOW <i>object</i> | 902 |
| SHOW QUERY LOGGING | 910 |
| SHOW STATISTICS | 912 |
| Appendix A: Additional Information | 922 |

Introduction to SQL Data Definition Language

Detailed Topics

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

Teradata Vantage™ - SQL Data Definition Language Detailed Topics describes the concepts and applications of Teradata SQL language statements used to define or restructure the database. The statements used to perform these tasks are referred to as the *SQL Data Definition Language*.

This document provides supplemental information about selected SQL DDL statements as necessary. For a complete list of SQL DDL statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

You should use this document in conjunction with the other volumes of the SQL document set, particularly *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144, as well as *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

The SQL statements used to assign or revoke access privileges to the data in a database are described in *Teradata Vantage™ - SQL Data Control Language*, B035-1149. SQL statements that you use to query or update a database or to analyze queries and query workloads are described in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Changes and Additions

| Date | Description |
|-----------|---|
| July 2021 | <ul style="list-style-type: none"> Fallback subtables are no longer compressed by default. Updated Block-Level Compression and Tables and BlockCompression Reserved Storage Management Query Bands. DEFINER and INVOKER authorizations are deprecated for NOS (updated DEFINER and INVOKER Authorizations). You can now create error tables for foreign tables (see Rules and Restrictions for Error Tables and Error Logging Limits). |
| June 2020 | Recommendation on storing source files on the server changed to storing the files on the client system, except on Intellicloud systems. Source files on Intellicloud systems must reside on the database server. EXTERNAL NAME Clause |

ALTER FUNCTION - ALTER PROCEDURE

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

ALTER FUNCTION (External Form)

When to Specify Unprotected Mode

To protect the system from new external routines that have not passed your quality metrics, the default execution mode for all external routines is EXECUTE PROTECTED. If the default specification for CREATE/REPLACE FUNCTION were EXECUTE NOT PROTECTED, then an error in the created function could crash the system.

Protected mode is a database state that traps on memory address violations, corrupted data structures, and illegal computations, such as divide-by-zero, to ensure they do not crash the system. Protected mode does *not* detect or protect against CPU and memory loops or operating system I/Os such as opening external files to read from or write to them.

This execution mode isolates all of the data the external routine might access as a separate process, or “platform” in its own local work space (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details). If any memory violation or other system error occurs, then the error is localized to the routine and the transaction executing it, not the entire database (see [Protected and Unprotected Execution Modes](#) for more information).

Because this isolation also makes the C or C++ routine run slower, you should change its protection mode to unprotected once it has passed all your quality metrics for being put into production use. You change the execution mode for the routine with a simple ALTER FUNCTION request and specifying the EXECUTE NOT PROTECTED option.

Both Java and row-level security constraint UDFs *must* run in protected mode and cannot be altered to run in unprotected mode (see [Special Considerations for Java UDFs](#)).

Do *not* make this change for any UDF that makes OS system calls. Any such UDF should always be run in protected mode. You do not need to recompile or relink the external routine after changing its protection mode.

As a rule, the DBA should only specify direct execution mode (assigned by the EXECUTE NOT PROTECTED option) for a function that performs CPU-only operations after it has been thoroughly debugged. No other user than the DBA, not even the software engineer that writes a UDF, should be granted the privileges to alter it to run in unprotected mode. Do not run external routines that cause the OS to consume system resources in unprotected mode. This includes anything that causes the OS to allocate system context, including open files, pipes, semaphores, tokens, threads (processes), and so on.

The following table summarizes how the UDF protection mode options should be used:

| IF ... | THEN specify ... |
|---|--|
| you are still developing and debugging a function | EXECUTE PROTECTED. |
| the function is written in Java | EXECUTE PROTECTED. You cannot alter a Java function to run in unprotected mode. |

| IF ... | THEN specify ... |
|--|--|
| the function opens a file or uses another operating system resource that requires tracking by the operating system This includes anything that causes the OS to allocate system context, including open files, pipes, semaphores, tokens, threads (processes), and so on. | EXECUTE PROTECTED. Running such a function in unprotected mode could interfere with the proper operation of the database. |
| the function is a computational function that does not use any operating system resources | EXECUTE NOT PROTECTED. Running a UDF in unprotected mode speeds up the processing of the function considerably. Use this option only after thoroughly debugging the function and making sure it produces the correct output. |

Special Considerations for Java UDFs

Java UDFs can only run in EXECUTE PROTECTED mode.

For Java UDFs, the COMPILE option operates identically to UDFs written in C or C++ with the exception that the extension the JAR file references in the Java UDF EXTERNAL NAME clause is redistributed to all affected nodes on the system. Redistribution is useful when a JAR file is missing on a given node. Note that you cannot specify the COMPILE ONLY option for Java UDFs.

The ONLY clause on the COMPILE option for UDFs written in Java generates preamble files, but does not redistribute their associated JAR file.

For more information, see [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#) and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

UDF Servers

Protected mode UDFs, including secure mode UDFs, run under a separate process called a server. UDFs set up to require external security run in a subtype of protected mode called secure mode. The system sets up separate processes, or servers, to support both protected mode and secure mode functions (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details).

The following series of events occurs the first time a secure mode UDF (the process is essentially identical for protected mode UDFs) is invoked in a session.

1. The UDF server setup logic checks at the authorization and determines whether a secure server process is already set up for the OS user authorization.

Because this is the first time the UDF has been invoked in the session, no secure server process has been set up.

2. Checks the user name and password.
3. Creates the secure server process and the UDF executes.

If this is not the first time the UDF has been executed for the session, the following series of events occurs:

1. The UDF secure server setup logic reads the authorization and determines that there is already a secure server process set up with that authorization.
2. The UDF is executed using the existing secure server.

If the maximum number of secure servers has been created and a new request comes in for a given authorization for which there is no established secure server, the following series of events occurs:

1. The UDF secure server logic attempts to find a secure server with the given authorization. It does not find one and determines that the maximum number of secure servers has already been set up.
2. The authorization is not set up, so the process validates the OS user by making a validation check on the user name and password.

If the logon attempt using the provided authorization information fails, the system returns an error to the session.

3. The UDF secure server logic finds the least used secure server process and terminates it.
4. The new secure server process is created and the UDF is executed.

Recompiling and Redistributing an Existing Function

The COMPILE option is used by DBAs to recompile functions that have been moved to a platform, system, or restored database other than the one in which they were created.

Specify the COMPILE option to recompile an existing function. If the source code is present, the system recompiles it, generates the object, recreates the .so file for functions written in C or C++, and distributes the .so files to all nodes of the system. If the procedure is written in Java, the COMPILE option recompiles the existing function, generates the object, and distributes the associated JAR files to all nodes of the system. The object is replaced with the recompiled version.

If the function was created with the object only (that is, there is no source code present to be recompiled), the function object is used to recreate the .so file and distribute it to all nodes of the system.

If the function is restored from a different platform to a new platform whose objects are not compatible, the COMPILE option fails. When this happens, you must recreate the function with the correct object or source code.

You cannot specify this option if the function was installed as part of a package.

Functions defined to run in unprotected mode and recompiled with this option do not retain the ability to run in unprotected mode unless you specify EXECUTE NOT PROTECTED as part of a follow-up ALTER FUNCTION request. See [When to Specify Unprotected Mode](#).

For example, suppose you have a UDF named *WriteMQ*, which has been set to run in EXECUTE NOT PROTECTED mode by an ALTER FUNCTION request that was submitted after the function was created using a CREATE FUNCTION request and then thoroughly debugged. See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#). Later, it becomes necessary for *WriteMQ* to be recompiled for some reason, so you must perform the following multistep procedure if *WriteMQ* is to be recompiled and

to retain its ability to run in unprotected mode. You cannot specify both COMPILE [ONLY] and EXECUTE [NOT] PROTECTED in the same ALTER FUNCTION request.

1. You first submit the following ALTER FUNCTION request to recompile the *WriteMQ* UDF:

```
ALTER FUNCTION WriteMQ COMPILE;
```

2. After the successful completion of this request, the system resets the protection mode for *WriteMQ* to EXECUTE PROTECTED, so you must run a second ALTER FUNCTION request to set its protection mode back to EXECUTE NOT PROTECTED:

```
ALTER FUNCTION WriteMQ EXECUTE NOT PROTECTED;
```

When you specify COMPILE ONLY, then only the UDF is recompiled and no new dynamic linked library are distributed to database nodes. The COMPILE ONLY option is not available for Java functions.

When a UDF is loaded onto another platform of a different type, it is marked as non-valid. All non-valid UDFs must be recompiled. If there are many UDFs in one database, you can save time by specifying the ONLY option for all but the last compilation in that database to avoid having to generate and distribute a new library. When you do this, make certain not to specify the ONLY option for the last UDF you recompile in that database.

One .so file is built for all C or C++ UDFs in each application category per database per node. The .so files are stored outside the database on the system disk of each node. If a .so file becomes corrupted on one node, you can regenerate it by issuing an ALTER FUNCTION request with the COMPILE option to rebuild the .so file.

To regenerate a corrupted .so file, you only need to compile one function in that database. The system regenerates the .so file, and includes all other user-defined functions defined in that database in the regenerated .so file.

Note:

Changing the protection mode for a UDT-related UDF also causes the system-generated UDT constructor function to be recompiled invisibly (invisible in that the system does not return any compilation messages unless the compilation fails for some reason, in which case the system returns an appropriate error message to the requestor).

Restrictions on Altering Functions That Implement UDT Transform and Ordering Functionality

You cannot use ALTER FUNCTION to change the protection mode for a function and to request its recompilation within the same request. You must submit separate requests to to alter the execution protection mode and recompile.

If you attempt to request both a change to the execution mode and a recompilation within the same request, the system returns an error.

ALTER METHOD

When to Specify Unprotected Mode

To protect the system from new external routines that have not passed your quality metrics, the default execution mode for all external routines is EXECUTE PROTECTED. If the default specification for CREATE METHOD and REPLACE METHOD were EXECUTE NOT PROTECTED, then an error in the created function could crash the system.

Protected mode is a database state that traps on memory address violations, corrupted data structures, and illegal computations, such as divide-by-zero, to ensure they do not crash the system. Protected mode does *not* detect or protect against CPU and memory loops or operating system I/Os such as opening external files to read from or write to them.

This execution mode isolates all of the data the external routine might access as a separate process, or “server” in its own local work space (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details). If any memory violation or other system error occurs, then the error is localized to the routine and the transaction executing it, not the entire database (see [Protected and Unprotected Execution Modes](#) for more information).

Because this isolation also makes the routine run slower, you should change its protection mode to unprotected once it has passed all your quality metrics for being put into production use. You change the execution mode for the routine with a simple ALTER METHOD request and specifying the EXECUTE NOT PROTECTED option.

Both Java and row-level security constraint UDFs must run in protected mode and cannot be altered to run in unprotected mode (see [Special Considerations for Java UDFs](#)).

Do *not* make this change for any method that makes OS system calls. Any such method should always be run in protected or secure mode. You do not need to recompile or relink the external routine after changing its protection mode.

As a rule, the DBA should only specify direct execution mode (assigned by the EXECUTE NOT PROTECTED option) for a method that performs CPU-only operations after it has been thoroughly debugged. No other user than the DBA, not even the software engineer that writes a UDF, should be granted the privileges to alter it to run in unprotected mode. *Never* run external routines that cause the OS to consume system resources in unprotected mode. This includes anything that causes the OS to allocate system context, including open files, pipes, semaphores, tokens, threads (processes), and so on.

The following table summarizes how the method protection mode options should be used:

| IF ... | THEN specify ... |
|--|--|
| you are still developing and debugging a method | EXECUTE PROTECTED. |
| the method opens a file or uses another operating system resource that requires tracking by the operating system This includes anything that causes the OS to allocate system context, including open files, pipes, | EXECUTE PROTECTED. Running such a method in unprotected mode could interfere with the proper operation of the database. |

| IF ... | THEN specify ... |
|--|--|
| semaphores, tokens, threads (processes), and so on. | |
| the method does not use any operating system resources | <p>EXECUTE NOT PROTECTED.</p> <p>Running a method in unprotected mode speeds up processing considerably.</p> <p>Use this option only after thoroughly debugging the method and making sure it produces the correct output.</p> |

Recompiling and Redistributing an Existing Method

The COMPILE option is intended to be used by DBAs to recompile methods that have been moved to a platform, system, or restored database other than the one in which they were created.

Specify the COMPILE option to recompile an existing method. If its source code is present, then the system recompiles it, generates the object, recreates the .so file, and distributes it to all nodes of the system. The object is replaced with the recompiled version.

If the method was created with the object only (that is, there is no source code present to be recompiled), the method object is used to recreate the .so file and distribute it to all nodes of the system.

If the method is restored from a different platform to a new platform whose objects are not compatible, the COMPILE option fails. When this happens, you must recreate the method with the correct object or source code.

You cannot specify this option if the method was installed as part of a package.

Methods defined to run in unprotected mode and recompiled with this option do not retain the ability to run in unprotected mode unless you specify EXECUTE NOT PROTECTED as part of a follow-up ALTER METHOD request. See [When to Specify Unprotected Mode](#).

For example, suppose you have a method named *in_state()* for the UDT named *address*, and *in_state()* has been set to run in EXECUTE NOT PROTECTED mode by an ALTER METHOD request that was submitted after the method was created using a CREATE METHOD request and then thoroughly debugged. See [CREATE METHOD](#). Some time later, it becomes necessary for *in_state()* to be recompiled for some reason, so you must perform the following multistep procedure if *in_state()* is to be both recompiled and to retain its ability to run in unprotected mode (because you cannot specify both COMPILE [ONLY] and EXECUTE [NOT] PROTECTED in the same ALTER METHOD request).

1. You first submit the following ALTER METHOD request to recompile the *in_state()* method:

```
ALTER METHOD in_state FOR SYSUDTLIB.address COMPILE;
```

2. After the successful completion of this request, the system resets the protection mode for *in_state()* to EXECUTE PROTECTED, so you must run a second ALTER METHOD request to set its protection mode back to EXECUTE NOT PROTECTED:


```
ALTER METHOD in_state FOR SYSUDTLIB.address
EXECUTE NOT PROTECTED;
```

When you specify `COMPILE ONLY`, then only the method is recompiled and no new dynamic linked libraries are distributed to database nodes.

See [Restrictions on Altering Methods That Implement UDT Transform and Ordering Functionality](#) for a description of the restrictions imposed on using `ALTER METHOD` to change the protection mode, or to recompile the object code for methods that implement UDT transform or ordering functionality.

Also see [CREATE ORDERING and REPLACE ORDERING](#) and [CREATE TRANSFORM and REPLACE TRANSFORM](#).

When a method is loaded onto another platform of a different type, it is marked as non-valid. All non-valid methods must be recompiled. If there are many methods in one database, you can save time by specifying the `ONLY` option for all but the last compilation in that database to avoid having to generate and distribute a new library. When you do this, make certain *not* to specify the `ONLY` option for the last method you recompile in that database.

One `.so` file is built for all methods in each database. The `.so` files are stored outside the database on the system disk of each node. If a `.so` file becomes corrupted on one node, the DBA can regenerate it by issuing an `ALTER METHOD` request with the `COMPILE` option to rebuild the `.so` file.

To regenerate a corrupted `.so` file, you only need to compile one method in that database. The system regenerates the `.so` file, and includes all other user-defined methods defined in that database in the regenerated `.so` file.

Note:

Changing the protection mode for a method also causes the system-generated UDF constructor function to be recompiled invisibly. That is, the system does not return any compilation messages unless the compilation fails. Then the system returns an appropriate error message to the requestor.

Restrictions on Altering Methods That Implement UDT Transform and Ordering Functionality

You cannot use `ALTER METHOD` to change the protection mode for a method and to request its recompilation within the same request. You must submit separate requests to do this: one to alter the execution protection mode and a second to request the recompilation.

If you attempt to request both a change to the execution mode and a recompilation within the same request, the system returns an error.

ALTER PROCEDURE (External Form)

Invocation Restrictions

Valid for external procedures only.

Not valid inside a procedure body.

Function of ALTER PROCEDURE (External Form) Requests

ALTER PROCEDURE (External Form) enables you to control whether an existing external procedure can be executed directly or indirectly as a separate process by the database. As a general rule, you should only permit a function to run in direct execution mode after it has been thoroughly debugged. When you create a new external procedure, it executes indirectly in protected execution mode by default (see [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#)).

ALTER PROCEDURE (External Form) also allows you to recompile or relink an external procedure and redistribute it (and its associated JAR file if the external routine is written in Java) using the COMPILE option. This option is primarily intended for use by DBAs to recompile existing procedures that have been moved to another platform, system, or restored database.

When to Specify Unprotected Mode

To protect the system from new external routines that have not yet passed your quality metrics, the default execution mode for all external routines is EXECUTE PROTECTED. If the default specification for CREATE/REPLACE PROCEDURE were EXECUTE NOT PROTECTED, then an error in the created function could crash the system.

Protected mode is a database state that traps on memory address violations, corrupted data structures, and illegal computations, such as divide-by-zero, to ensure they do not crash the system. Protected mode does *not* detect or protect against CPU and memory loops or operating system I/Os such as opening external files to read from or write to them.

This execution mode isolates all of the data the external routine might access as a separate process, or “server” in its own local work space (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details). If any memory violation or other system error occurs, then the error is localized to the routine and the transaction executing it, not the entire database (see [Protected and Unprotected Execution Modes](#) for more information).

Because this isolation also makes the routine run slower, you should change its protection mode to unprotected once it has passed all your quality metrics for being put into production use. You change the execution mode for the routine with a simple ALTER PROCEDURE request (see [ALTER PROCEDURE \(External Form\)](#)) and specifying the EXECUTE NOT PROTECTED option.

Do *not* make this change for any procedure that makes OS system calls. Any such procedure should always be run in protected mode. You do not need to recompile or relink the external routine after changing its protection mode.

As a rule, the DBA should only specify direct execution mode (assigned by the EXECUTE NOT PROTECTED option) for a procedure that performs CPU-only operations after it has been thoroughly debugged. No other user than the DBA, not even the software engineer who writes a UDF, should be granted the privileges to alter it to run in unprotected mode. *Never* run external routines that cause the OS to consume system resources in unprotected mode. This includes anything that causes the OS to allocate system context, including open files, pipes, semaphores, tokens, threads (processes), and so on.

The following table summarizes how the external procedure protection mode options should be used.

| IF ... | THEN specify ... |
|---|---|
| you are still developing and debugging a procedure | EXECUTE PROTECTED. |
| the procedure opens a file or uses another operating system resource that requires tracking by the operating system This includes anything that causes the OS to allocate system context, including open files, pipes, semaphores, tokens, threads (processes), and so on. | EXECUTE PROTECTED. Running such a procedure in unprotected mode could interfere with the proper operation of the database. |
| the procedure is linked with CLIV2 or is written in Java | EXECUTE PROTECTED. External procedures that are linked with CLIV2 can <i>only</i> run in protected mode. |
| the procedure is a computational procedure that does not use any operating system resources | EXECUTE NOT PROTECTED. Running an external procedure in unprotected mode speeds up the processing of the function considerably. Use this option only after thoroughly debugging the procedure and making sure it produces the correct output. |

Protected Mode External Procedure Servers

Protected mode external procedures, including secure mode procedures, run under a separate process from the database. External procedures set up to require external security run in a subtype of protected mode called secure mode. The system sets up separate processes, or *servers*, to support secure mode procedures (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details).

The following series of events occurs the first time a secure mode external procedure is invoked in a session. The process is essentially identical for protected mode external procedures.

1. The external procedure secure server setup logic looks at the authorization and determines whether a secure server process is already set up for the given OS user authorization.

Because this is the first time the procedure has been invoked in the session, no secure server process has been set up.

2. The authorization is not set up, so the process validates the OS user by making a validation check on the user name and password.

If the logon attempt using the provided authorization information fails, the system returns an error to the session.

3. If the authorization is successful, the system creates the secure server process and the procedure executes.

If this is not the first time the procedure has been executed for the session, the following series of events occurs:

1. The external procedure secure server setup logic reads the authorization and determines that there is already a secure server process set up with that authorization.
2. The procedure is executed using the existing secure server.

If the maximum number of secure servers has been created and a new request comes in for a given authorization for which there is no established secure server, the following series of events occurs:

1. The external procedure secure server logic attempts to find a secure server with the given authorization. It does not find one and determines that the maximum number of secure servers has already been set up.
2. The authorization is not set up, so the process validates the OS user by making a validation check on the user name and password.

If the logon attempt using the provided authorization information fails, the system returns an error to the session.

3. The external procedure secure server logic finds the least used secure server process and terminates it.
4. The new secure server process is created and the procedure is executed.

Special Considerations for Java External Procedures

Java external procedures can only be run in EXECUTE PROTECTED mode. If you specify EXECUTE NOT PROTECTED for a Java procedure, the request aborts and returns an error message to the requestor.

For Java external procedures, the COMPILE option operates identically to external procedures written in C or C++ with the exception that the extension the JAR file references in the Java external procedure EXTERNAL NAME clause (see [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#)) is redistributed to all affected nodes on the system. Redistribution is useful when a JAR file is missing on a given node.

The ONLY clause on the COMPILE option for external procedures written in Java generates preamble files, but does not redistribute their associated JAR file.

AT TIME ZONE Option for External Procedures

When you create an external procedure, Vantage stores the current session time zone for the procedure along with its definition to enable the SQL language elements in the procedure to execute in a consistent time zone and produce consistent results. However, time or timestamp data passed as an input

parameter to the procedure still use the runtime session time zone rather than the creation time zone for the procedure.

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

You can only specify AT TIME ZONE with the COMPILE [ONLY] option, and it must follow the COMPILE [ONLY] specification. Otherwise, the system returns an error to the requestor.

Dictionary Updates and the COMPILE Option for Java Procedures

The following dictionary table updates occur as the result of an ALTER PROCEDURE COMPILE request for a Java external procedure:

1. The corresponding row for a JAR in DBC.TVM is updated with a new version number. The Platform column in DBC.TVM is also updated.
2. The corresponding row for a JAR in DBC.JARS is updated with a new version number.
3. The corresponding row for the current database in DBC.Dbase is updated with an incremented JarLibRevision number.

ALTER PROCEDURE (SQL Form)

Invocation Restrictions

Valid for SQL procedures only.

Not valid inside a procedure body.

Limitations

You cannot use ALTER PROCEDURE to change the DDL definition of an SQL procedure, that is, to REPLACE the procedure. To replace the definition on an SQL procedure, you must submit a REPLACE PROCEDURE request (see [CREATE PROCEDURE and REPLACE PROCEDURE \(SQL Form\)](#)).

Attributes Changed by ALTER PROCEDURE (SQL Form)

ALTER PROCEDURE can alter the following attributes of the recompiled SQL procedure.

- Platform.
This is an implicit change and cannot be specified by an ALTER PROCEDURE request.
- TDSP version number.
This is an implicit change and cannot be specified by an ALTER PROCEDURE request.
For information about procedure version numbers, see [HELP PROCEDURE](#).
- Creation time zone.
This is an explicit change that you specify using the AT TIME ZONE option.

You can also change one or all of the following attributes:

- SPL to NO SPL.
You cannot change NO SPL back to SPL.
- WARNING to NO WARNING and vice versa.

Attributes Not Changed by ALTER PROCEDURE (SQL Form)

The creator and the immediate owner of a procedure are not changed after recompilation.

ALTER PROCEDURE (SQL Form) also does not change the following attributes of the procedure being recompiled.

- Session mode
- Creator character set
- Creator character type
- Default database
- Privileges granted to the procedure

ALTER PROCEDURE (SQL Form) Rules

- If a procedure is recompiled by an ALTER PROCEDURE request with NO SPL option, the source text of the procedure is deleted from the system and the procedure cannot be recompiled again.
- You can alter a procedure in the same session mode in which the procedure was originally created. Procedures created in ANSI session mode cannot be recompiled in Teradata session mode and vice versa.
- An ALTER PROCEDURE request cannot be specified inside a procedure body.
- If you do not specify any compile-time options with an ALTER PROCEDURE request, the options with which the procedure was previously created, replaced, or recompiled apply. If some compile-time options are specified with an ALTER PROCEDURE request, the unspecified options default to the existing options of the procedure.
- You must specify COMMIT after an ALTER PROCEDURE request is executed in ANSI mode transactions, as true for all DDL statements.
- In Teradata session mode, ALTER PROCEDURE must be the last request in a transaction.
- All the statements other than DML, DDL, and DCL statements within an SQL procedure are validated syntactically and semantically during the execution of an ALTER PROCEDURE request.

For DML, DDL, and DCL statements within the procedure body, the validation includes syntactic validation and name resolution but does *not* include the following.

- Access privilege checks
- Data type compatibility
- If the execution of an ALTER PROCEDURE request fails, the existing SQL create text for the procedure is retained.
- Errors or warnings generated during the recompilation of a procedure are reported as part of the SUCCESS or OK response. The activity count in the parcel is set to the total number of errors and warnings.

AT TIME ZONE Option for SQL Procedures

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

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

You can only specify AT TIME ZONE with the COMPILE [ONLY] option, and it must follow the COMPILE [ONLY] specification. Otherwise, the system returns an error to the requestor.

Locks and Concurrency

When an ALTER PROCEDURE request is performed, the system places an EXCLUSIVE lock on the corresponding procedure table in the database. The procedure is unavailable to other users during the recompilation time.

ALTER TABLE

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

ALTER TABLE (Basic Table Parameters)

ALTER TABLE and Table Version Numbers

Any change to the structure of a table or join index increments its version number.

You cannot perform any of the following actions over a DDL request that alters table structure.

- Cluster restore
- Single AMP restore
- Rollforward or rollback of a permanent journal

Rules and Restrictions for the Number of Changes Made in One ALTER TABLE Statement

The rules for number of changes that can be made within a single ALTER TABLE statement are as follows.

- Only one change can be made per column per ALTER TABLE statement.
- Only one CHECK alteration is permitted per ALTER TABLE statement, and you cannot mix CHECK and non-CHECK alterations on the same table for the same ALTER TABLE statement.

Rules and Restrictions for Modifying Nonpartitioned NoPI Tables

ALTER TABLE supports nonpartitioned NoPI tables in the same manner as tables with a primary index or primary AMP index, except:

- You cannot add a permanent journal to a nonpartitioned NoPI table or to a column-partitioned table definition.
- You cannot add an identity column to a nonpartitioned NoPI table definition.
- To alter a NoPI table to have partitioning, the table must be empty and the partitioning must include column partitioning.
- To alter a NoPI table to have a primary index or primary AMP index, the table must be empty. To alter a table with a primary index to have no primary index, the table must be empty. To alter a table with a primary AMP index to have no primary index and not be partitioned, the table must be empty.

About Referential Integrity Error Tables for Column-Partitioned Tables

This topic does not apply to the user-defined error tables described in [Rules and Restrictions for Modifying Data Tables Defined With an Error Table](#).

When you add a new referential constraint to a table, Vantage generates a referential integrity error table. For a column-partitioned child table, this error table is a NoPI table with neither column nor row partitioning.

The referential integrity error table for a row-partitioned child table has the same primary index and row partitioning as the row-partitioned child table.

Rules and Restrictions for Modifying NUMBER Columns

The following rules and restrictions apply to modifying NUMBER columns in a table. These rules and restrictions apply to modifying the precision and scale of exact NUMBER columns with modifying their rows.

- You can increase the precision of exact NUMBER columns.
- You can increase both the precision and scale of exact NUMBER columns as long as you increase the precision by as much as you increase the scale.
- You can change an exact NUMBER column to an approximate NUMBER column or approximate NUMBER column with scale as long as you do not decrease the scale.
- You can change both an exact NUMBER column and an approximate NUMBER column with scale to an approximate NUMBER column without scale.
- You cannot decrease the precision of an exact NUMBER column unless the table is not populated with rows.
- You cannot decrease the scale of an exact NUMBER column unless the table is not populated with rows.
- You cannot increase the scale of a NUMBER column unless you also increase its precision by at least as much.
- You cannot modify an approximate NUMBER column to be either an exact NUMBER column or an approximate NUMBER column with scale.
- You cannot modify an exact NUMBER column to be an approximate NUMBER column with a decrease in scale.

Rules and Restrictions for Modifying Queue Tables

There are a number of restrictions on queue tables, as detailed in the following list.

- You cannot change a queue table into a non-queue table.
- You cannot modify the first column defined for a queue table to be something other than a user-defined QITS column (see [QITS Column](#)).
- You cannot drop the QITS column from a queue table.
- You cannot modify the QITS column by adding either of the following attributes.
 - UNIQUE
 - PRIMARY KEY

Because of this restriction, you cannot add a simple UPI on the QITS column.

Similarly, you cannot modify the QITS column to make it a simple UPI.

- You cannot add either of the following constraints to a queue table definition.
 - FOREIGN KEY ... REFERENCES
 - REFERENCES

- You cannot modify any column of a queue table to have a LOB data type (see *Teradata Vantage™ - Data Types and Literals*, B035-1143).
- You cannot add a permanent journal to the definition for a queue table.
- You cannot add a reference to queue table columns in a REFERENCES clause for *any* table.

Otherwise, all column- and table-level constraint clauses are valid within queue table definitions with the exception of UNIQUE and PRIMARY KEY constraints not being valid attributes for the QITS column.

- You cannot modify a queue table definition to have a partitioned primary index.

Rules and Restrictions for Modifying Data Tables Defined With an Error Table

This topic does not apply to the referential integrity error tables described in [About Referential Integrity Error Tables for Column-Partitioned Tables](#).

Several rules apply to altering a table that is associated with an error table. See [CREATE ERROR TABLE](#).

The system maintains the compatibility between data tables and their associated error tables by disallowing *any* of the operations in the following list when an error table is defined for a data table.

- You cannot make any of the following changes to a data table definition if the table is associated with an error table.
 - Add a column
 - Drop a column
 - Alter its primary index
 - Change its fallback protection
- You *can* modify the partitioning of a partitioned data table that has an associated error table.

Rules and Restrictions for Large Object Data Types

Vantage supports the following large object (LOB) data types.

- Binary large object (BLOB)
- Character large object (CLOB)

The rules for specifying these types in an ALTER TABLE request are different from the rules that apply to other data types. Rules for specifying LOB data types in an ALTER TABLE statement are as follows.

- You can increase the maximum size for a BLOB or CLOB column definition, but you cannot decrease that size.

To increase the maximum size for a BLOB or CLOB column definition, use the ADD *column_name* option, specify the name of the column whose LOB data type you want to increase, and Vantage modifies the size of the type to the value you specify. For example, suppose you want to double the maximum size of a CLOB column named *standard_error* in the table named *performance_data*

from 524,288,000 characters to 1,048,576,000 characters. You can use the following ALTER TABLE request to increase the size of the CLOB data type for `standard_error` to 1,048,576,000 characters.

```
ALTER TABLE performance_data
ADD standard_error CLOB(1048576000);
```

Note:

Vantage knows that column *standard_error* already exists in *performance_data*, so it interprets the ADD option to mean that *standard_error* is to be modified, not added.

- You can only alter the following column attributes for a BLOB or CLOB column.
 - NULL
 - NOT NULL

You cannot change the attribute from NULL to NOT NULL if there are nulls in any rows in the table for that column.

 - TITLE
 - FORMAT

If you change the `tdlocaledef.txt` file and issue a `tpareset` command, the new format string settings affect only those tables that are created after the reset. Existing table columns continue to use the existing format string in `DBC.TVFields` unless you submit an ALTER TABLE request to change it.
- You can add BLOB and CLOB columns to a maximum of 32 per base table definition.
- If a table already has 32 BLOB or CLOB columns or a mixture of both, you cannot add another BLOB or CLOB column to it using the same ALTER TABLE request that drops one of the 32 existing BLOB or CLOB columns.

Instead, you must drop an existing BLOB or CLOB column with one ALTER TABLE request and then add the new BLOB or CLOB column using a separate ALTER TABLE request.
- You can drop BLOB or CLOB columns from a base table.

If you drop all BLOB and CLOB columns from a base table, that table is no longer bound to any BLOB or CLOB restrictions.
- A constraint definition can neither be defined for nor reference a BLOB or CLOB column.
- You cannot specify expressions or referenced columns with BLOB or CLOB data types in a start or end expression for a primary index range.

ALTER TABLE Support For UDTs

You can use ALTER TABLE to do any of the following things with UDT columns.

- Add a UDT column to a table.

A LOB UDT column counts as one LOB column even if the UDT is a structured type with multiple LOB attributes. The limit of 32 LOB columns includes both predefined type LOB columns and LOB UDT columns.

Note:

A UDT must have both its ordering and its transform functionality defined before you can add a column typed with that UDT to a table definition.

If you attempt to add such a column to a table definition without first defining its ordering and transform characteristics, the system returns an error to the requestor.

Be aware that each UDT column you add to a table definition subtracts approximately 80 bytes from the available space in the table header. The total number of UDT columns that can be defined for a table is dependent on the presence of other features that also consume space in the table header such as indexes, compression, and so on.

In the absence of any other optional features that occupy table header space, the upper limit on the number of UDT columns that can be defined, assuming a fat table header is approximately 1600 columns. See *Teradata Vantage™ - Database Design*, B035-1094 for information about thin and fat table headers and the respective table header space taken up by various database features.

- Drop a UDT column from a table.
- Modify the attributes of a UDT column.

You can only modify the following set of attributes for a UDT column.

- [NOT] NULL
- FORMAT

The FORMAT string must be valid for the external type of the UDT (see [CREATE TRANSFORM and REPLACE TRANSFORM](#)).

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

- TITLE
- NAMED
- DEFAULT NULL
- You cannot modify the following attributes for UDT columns.
 - CASESPECIFIC
 - CHARACTER SET
 - [CHECK] CONSTRAINT
 - COMPRESS
 - DATE
 - DEFAULT *numeric_value*

- TIME
 - UPPERCASE
 - USER
 - WITH DEFAULT
- You cannot modify a table to have a column attributed with any of the following set of constraints to reference a UDT column.
 - [CHECK] CONSTRAINT
 - FOREIGN KEY ... REFERENCES
 - If a table is partitioned, its partitioning expressions cannot reference UDT columns.

Recommended Null Handling Technique For UDT Columns

If you intend to support null attributes or if you write a map ordering routine that can return nulls, then any CREATE TABLE or ALTER TABLE statements that specify that type should specify the NOT NULL attribute for that UDT column.

If nulls are permitted in the column and either the map ordering routine or the system-generated observer method for the UDT can return nulls, the results returned for identical requests can vary from query to query.

The ordering routine for a UDT determines whether or not column values are equal and also determines the collation order for sort operations.

If you do not specify the NOT NULL attribute for UDT columns, then the following objects may be treated equally.

- A column null.
- A structured type that contains null attributes and whose map or observer routine returns nulls.

Sometimes a column null is returned in the result set and other times the non-null structured type that contains null attributes is returned in the result set.

The following simple example indicates how you should specify the NOT NULL attribute for UDT columns in a table definition.

```
CREATE TABLE udtTable (
  id          INTEGER,
  udtColumn   myStructUdtWithNullAttributes NOT NULL);
```

Dropping a Column and Statistics

Before you can drop a column using an ALTER TABLE statement, you must drop any single-column or multi-column statistics on the column.

Changing a Column Data Type and Statistics

If you change a column data type using an ALTER TABLE request, all the statistics on a base table that reference the column are no longer used. Recollecting the statistics on the modified column makes them usable again.

ALTER TABLE and FALLBACK

When an ALTER TABLE request adds FALLBACK, the operation generates a full-table scan. To add FALLBACK to a table, all AMPs must be online and operational.

When a table or join index is partitioned, its fallback rows are partitioned the same as its primary data rows. If the primary data rows have a primary index, the fallback data rows have the same primary index. If the primary data rows do not have a primary index, the fallback data rows do not have a primary index.

You can change the FALLBACK properties of a global temporary table definition if and only if there are no materialized instances of that table anywhere in the system. However, when an instance of a global temporary table is materialized, Vantage creates it without fallback irrespective of its definition unless there is a down AMP in the system. In that case, and that case only, the table is materialized with fallback if it has been defined to do so.

To enable the file system to detect all hardware read errors for tables and join indexes, set CHECKSUM to ON.

ALTER TABLE, FALLBACK, and Read From Fallback

Fallback is very important when a system needs to reconstruct data from fallback copies if a hardware read error occurs when it attempts to read the primary copy of the data. When a read error occurs in this case, the file system reads the fallback copy of the rows and reconstructs a memory-resident image of them on their home AMP. This is referred to as Read From Fallback. See *Teradata Vantage™ - Database Design*, B035-1094.

Without this feature, the file system fault isolation logic would abort the transaction and, depending on the error, possibly mark the table as being down. See [SET DOWN and RESET DOWN Options](#). 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 data block cannot be repaired, Read From Fallback substitutes an error-free fallback copy of the corrupt rows each time the read error occurs.

To avoid the overhead of this substitution, you must rebuild the primary copy of the data manually from the fallback copy using the Table Rebuild utility. For information about Table Rebuild, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Transient Journaling

Transient journaling supports rollback and backup operations.

To preserve the restore capability of transient journaling, changing only journaling options with ALTER TABLE retains the version number of the table. Other ALTER TABLE options change the structure of tables and thus require incrementing the table version number.

The transient journaling options include:

- WITH JOURNAL TABLE
- JOURNAL
- AFTER JOURNAL

Activating Permanent Journaling

If a table is created in a database that does not have permanent journaling activated, you can activate its journaling with ALTER TABLE. To do this, you must alter the table to include a journaling option such as AFTER JOURNAL or BEFORE JOURNAL or both. As soon as the ALTER TABLE request completes successfully, permanent journaling is activated for that table.

To determine which tables in your database are journal tables, use the following query.

```
SELECT DBC.dbase.databasename (FORMAT 'X(15)'),
       DBC.tvn.tvnname (FORMAT 'X(25)')
FROM DBC.tvn, DBC.dbase
WHERE DBC.dbase.databaseid=DBC.tvn.databaseid
AND   DBC.tvn.tablekind='j'
ORDER BY 1,2;
```

To determine which databases and users in your database currently have a default journal table defined for them, use the following query.

```
SELECT d.databasename (TITLE 'Database'), TRIM(dx.databasename)
       ||'|'||TRIM(t.tvnname)(TITLE 'Journal')
FROM DBC.dbase AS d,
     DBC.TVM   AS t,
     DBC.dbase AS dx
WHERE d.journalid IS NOT NULL
AND   d.journalid <> '00'xb
AND   d.journalid = t.tvmid
AND   t.databaseid = dx.databaseid
ORDER BY 1;
```

To determine which tables in your database are currently being journaled, use the following query.

```
SELECT TRIM(Tables_DB)||'.'||TableName (TITLE 'Table',
      CHARACTER(26)), 'Assigned To' (TITLE ' '), TRIM(journals_db)
      ||'.'||JournalName (TITLE 'Journals', CHARACTER(26))
FROM DBC.JournalsV
ORDER BY 1,2;
```

You can also determine which tables in your database are currently being journaled using the following query that has a somewhat different syntax.

```
SELECT TRIM(d.databasename)||'.'||TRIM(t.tvpname) (FORMAT
      'x(45)', TITLE 'Table'), TRIM(dj.databasename)
      ||'.'||TRIM(tj.tvpname) (TITLE 'Journal')
FROM DBC.TVM AS t,
      DBC.TVM AS tj,
      DBC.dbase AS d,
      DBC.dbase AS dj
WHERE t.journalid IS NOT NULL
AND t.journalid <> '00'xb
AND t.journalid = tj.tvmid
AND d.databaseid = t.databaseid
AND dj.databaseid = tj.databaseid
ORDER BY 1;
```

Also see [CREATE DATABASE](#), [CREATE USER](#), [MODIFY DATABASE](#), and [MODIFY USER](#).

Local Journaling

See [CREATE TABLE](#) for information about local journaling.

FREESPACE PERCENT

Specify this option to modify the free space percentage for a table.

Note:

Not all database operations honor the freespace percentage you specify in an ALTER TABLE request. See the table in the topic [FREESPACE PERCENT](#) for lists of which operations honor the specified freespace percentage and which do not.

Note:

Do *not* use the Ferret utility PACKDISK command to change this value once you have created a table or have modified its FREESPACE PERCENT with an ALTER TABLE request.

Instead, submit an ALTER TABLE request to change the free space percent value for the table and then immediately afterward submit a PACKDISK command that specifies the same free space percent value you set with that ALTER TABLE request (see the documentation for the Ferret utility in *Teradata Vantage™ - Database Utilities*, B035-1102 for more information and instructions for running PACKDISK).

The reason for this is as follows: if you manually submit a PACKDISK command and specify a free space percentage on the command line, PACKDISK packs or unpacks the cylinders for the table to match the specified free space percentage. However, if the free space percentage you specify differs from the value specified for the table when it was created or most recently altered, the database starts a new AutoCylPack task immediately after PACKDISK completes, and this AutoCylPack operation nullifies the free space percentage you had just specified with PACKDISK to realign it with the free space percentage specified when you created or last altered the table.

MERGEBLOCKRATIO

The MERGEBLOCKRATIO option provides a way to combine existing small data blocks into a single larger data block during full table modification operations for permanent tables and permanent journal tables. This option is not available for volatile and global temporary tables. The file system uses the merge block ratio that you specify to reduce the number of data blocks within a table that would otherwise consist mainly of small data blocks. Reducing the number of small data blocks enables Vantage to reduce the I/O overhead of operations that must read and modify a large percentage of the table. It is not possible to define a general block size that would be considered to be small in this context. The exact block size that fits this description is somewhat subjective and can differ based on the I/O performance that occurs at different sites.

For tables that are frequently modified by inserting new data, the average block size varies between 50% and 100% of their maximum supported multirow block size. This maximum supported size is defined as either the table-level attribute DATABLOCKSIZE (see [CREATE TABLE](#)) or the system-level block size for the table as defined in the DBS Control record if you have not established a value for the DATABLOCKSIZE attribute for a table.

Note that when you delete rows from a table, the blocks that previously held the deleted rows can become smaller than the sizes in the defined range, and their new sizes can vary considerably from your user-defined block size.

An extreme example of a block size distribution that benefits from having its data blocks merged together is a table whose blocks were created with an average size of 64 KB, but which have gradually shrunk or split over time to an average size of only 8 KB. This example is extreme because it is uncommon for requests to access every data block in a table. If block splits were common for this table, its number of data blocks can have increased by an order of magnitude.

If the rows in the table could be repackaged into blocks whose size were closer to the original average block size, you would probably experience improved response times for queries that read most or all of the blocks in the table because the number of logical or physical reads (or both) would then be reduced by an order of magnitude as well.

Merging blocks automatically augments the existing functionality of the ALTER TABLE statement DATABLOCKSIZE option, with or without the IMMEDIATE option. Although this option lessens the problem of tables with small blocks, there are drawbacks:

- Resolving the small data blocks problem requires a DBA to manually execute new SQL requests.
- You must also determine which tables and block sizes are causing performance problems before you can submit the SQL requests necessary to resolve the problem. For example, table data block size statistics are available using the SHOWBLOCKS command of the Ferret utility or the equivalent SQL interface using the CreateFsysInfoTable and PopulateFsysInfoTable macros. For information on the Ferret utility, see *Teradata Vantage™ - Database Utilities*, B035-1102. For information on the CreateFsysInfoTable and PopulateFsysInfoTable macros, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.
- The operations performed by the problem resolving SQL request require an EXCLUSIVE table-level lock, which then blocks concurrent update activity on the table.
- If you specify the IMMEDIATE option, some data blocks in the table may become smaller as soon as the table is updated. If you do not specify the IMMEDIATE option, some data blocks in the table might take a long time to grow larger again if they are not updated for some time. See [IMMEDIATE DATABLOCKSIZE](#).

The merge block ratio approach does not have these drawbacks and offers the following enhanced functionality:

- Runs automatically without DBA intervention.
- Does not require the analysis of block size histograms.
- Does not require any specific AMP-level locking.
- Continuously searches for small data blocks to merge together even when some of those blocks are not being updated.

For more information about performance-related aspects of the MERGEBLOCKRATIO option, see [Performance Aspects of Merging Data Blocks](#).

The size threshold at which small data blocks are merged can be controlled either using the MERGEBLOCKRATIO option or by changing the setting of the MergeBlockRatio DBS Control flag. It is also possible to deactivate the merging of small data blocks by changing the setting of the DisableMergeBlocks field of the DBS Control record. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

MINIMUM DATABLOCKSIZE

This specification changes the data block size for a table to the minimum possible value for your system. The option sets the maximum data block size for blocks that contain multiple rows to the minimum legal value of 21,504 bytes (42 sectors) for systems running with large cylinders or 9,216 bytes (18 sectors) for systems running without large cylinders.

IMMEDIATE DATABLOCKSIZE

The only reason to change the data block size for a table is to enhance performance. Repacking the data blocks of large tables is a time-consuming process, so specifying the IMMEDIATE option for large tables substantially increases the amount of time required for the ALTER TABLE request to complete.

Field upgrades of systems do not change the data block size from 64 KB to 127.5 KB until a block splits. To take immediate advantage of the performance enhancements offered by the 127.5 KB block size, you must force the data block size change directly.

To upgrade the data block size from 63.5 KB to 127.5 KB, perform one of the following ALTER TABLE requests on every table in every database.

```
ALTER TABLE database_name.table_name, DEFAULT DATABLOCKSIZE
        IMMEDIATE;

ALTER TABLE database_name.table_name, DATABLOCKSIZE = 127.5 KBYTES
        IMMEDIATE;
```

If you do not specify the IMMEDIATE keyword, the definition for DATABLOCKSIZE is set to 127.5 KB, but the size increase does not occur until rows are inserted into the newly defined table.

Because of the extra time required to process requests with the IMMEDIATE option, you should plan to convert your data block sizes during non-peak hours.

When an ALTER TABLE request that specifies the IMMEDIATE option aborts or is aborted by the user, the repacking might be incomplete; that is, some data blocks are of their original size, while others are of the newly specified size.

The DATABLOCKSIZE value returned in response to a SHOW TABLE request is the value specified in the most recently entered ALTER TABLE or CREATE TABLE request.

If no DATABLOCKSIZE specification is specified in the ALTER TABLE request, then the data block size is not changed and no data blocks are repacked.

BLOCKCOMPRESSION

Use this option to change the current temperature-based block compression state of a table to a new state.

For information about how the BLOCKCOMPRESSION options work for tables, see [BLOCKCOMPRESSION](#).

Typically for a table with block compression set to AUTOTEMP, some blocks in the table are compressed and some are not. If you change the block compression for a table from AUTOTEMP mode to MANUAL or NEVER, some blocks in the table remain compressed and some do not.

A table is fully functional when it is in this state, but it has inconsistent block compression. If you change the block compression option for a table to MANUAL, when you modify the compressed blocks of the table, the newly created blocks are compressed. If the table was changed to NEVER, when you modify the compressed blocks of the table, the file system decompresses the newly created blocks. If you modify the noncompressed blocks of the table and have changed the BLOCKCOMPRESSION option from AUTOTEMP to either MANUAL or NEVER, the cylinders of newly created data blocks for a table remain noncompressed.

The following table explains what you must do to make the block-level compression for such a table consistent. For information about how to use the Ferret utility, see *Teradata Vantage™ - Database Utilities*, B035-1102.

| IF the block-level compression for the table changes from AUTOTEMP to this state ... | Use the following Ferret utility commands to make its block-level compression consistent ... |
|--|--|
| MANUAL | <ul style="list-style-type: none"> • COMPRESS • UNCOMPRESS |
| NEVER | UNCOMPRESS |

A table is fully functional when it is in a mixed block-level compression state, but its block compression *is* inconsistent. If you submit an ALTER TABLE request to change the BLOCKCOMPRESSION option for a table to MANUAL, when you modify the block-level compressed blocks of the table, the newly created blocks are compressed. If the table was changed to NEVER, when you modify the compressed blocks of the table, the file system decompresses the newly created blocks. If you modify the noncompressed blocks of the table and have changed the BLOCKCOMPRESSION option to either MANUAL or NEVER, newly created blocks remain noncompressed.

The best practice is not to use temperature-based block-level compression for a table that requires block-level compression consistency for the entire table.

You can combine multi-value compression, algorithmic compression, 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.

Adding Columns To a Table

The following rules and restrictions apply to adding columns to any table.

- You cannot modify a column more than once in an ALTER TABLE request.
- When a column is added to a base table that already contains data, the system generally needs to access each row of the table to add a field for the new column. The following principles apply when adding a column to a base table that already contains rows.

| IF this phrase ... | IS ... | THEN all rows initially ... |
|-------------------------|----------------------------------|--|
| DEFAULT | specified for the new column | contain the specified constant values in the field. |
| WITH DEFAULT | | contain the system default in the field. |
| DEFAULT WITH DEFAULT | not specified for the new column | are null for the column, and you cannot specify the NOT NULL phrase. |

- A column that is defined as NOT NULL can only be added to an empty base table if it has no DEFAULT or WITH DEFAULT phrase.

- You cannot change the data type, nullability, or name of an indexed column.
- You cannot change the data type, nullability, or name of a partitioning column for a partitioned table.
- You cannot add an identity column to an existing base table, nor can you add the identity column attribute to an existing column.
- You can add BLOB and CLOB columns to a maximum of 32 per base table. See [Rules and Restrictions for Large Object Data Types](#).
- You cannot add a CHARACTER, VARCHAR, or CLOB column with a server character set of KANJI1 to a table. Otherwise, the system returns an error to the requestor.

Adding Columns to a Column-Partitioned Table

The following rules and restrictions apply to adding new columns to a column-partitioned table.

The column cannot currently exist and the system adds it as a new column partition consisting of that single column.

The system adds a group of columns as a new column partition that consists of those columns.

KANJI1 Server Character Set Not Allowed

You cannot add columns with a server character set of KANJI1 to a column-partitioned table.

You should instead use the UNICODE server character set.

Single Modification to a Column Per ALTER TABLE statement

You cannot modify a column more than once in a single ALTER TABLE statement.

INTO Clause and Column-Partitioned Tables

You can only specify an INTO clause for an ADD clause of an ALTER TABLE statement if the object to be altered is a column-partitioned table.

If you do not specify an INTO option and a single column that is not delimited by parentheses is being added that does not currently exist for a column-partitioned table, the column is added as a new column partition consisting of that column. The system assigns a partition number to the new column partition that is not currently in use. The system determines the partition format for the new column partition and defines it with autocompression.

Whether or not you specify the INTO clause option, the system adds a new column as the last column, with respect to an * (ASTERISK character) that is used to select all columns from the table.

You cannot use an ALTER TABLE statement to modify the definition of a join index by adding columns, column partitions, or both.

An INTO clause cannot specify a column name that does not exist in the table.

You cannot add a new column and specify it in an INTO column in the same ALTER TABLE statement.

If you specify an INTO clause for a column or a group of columns, the column set being added becomes part of the column partition containing the column specified by *column_name* in the INTO clause.

You cannot add a column to a column partition that is already defined for that partition.

You cannot add a column to a column partition that is already defined in another column partition.

Column Partition Number

When you add columns to a column partition, Vantage assigns the altered column partition a new column partition number.

Adding columns to a column partition does not affect the number of defined column partitions (including the 2 internal use column partitions), the number of column partitions that can be added, the maximum number of column partitions, or the maximum column partition number.

Adding a Single-Column Partition Without Using INTO

If you do not specify an INTO clause and you add a single column that does not currently exist for a column-partitioned table, the column being added is added as a new column partition consisting of only that column.

Vantage assigns the new column partition a column partition number that is not currently in use. If no column partition number is available, the system returns an error to the requestor.

The format for the new column is system-determined and defined with autocompression by default.

You can specify either AUTO COMPRESS or NO AUTO COMPRESS for a new column partition added to a table if the specification is not preceded by INTO.

Adding a Group of Columns as a Partition Without Using INTO

If you do not specify an INTO clause but you add a group of one or more columns delimited by parentheses:

- The object being altered must be a column-partitioned table. You cannot alter a column-partitioned join index to add columns or column partitions.
- The columns in the group must not currently be defined for the table.
- The group of columns is added as a new column partition consisting of those columns and assigns a new column partition number.

If there are no column partition numbers available, the system returns a message to the requestor.

- If you specify COLUMN format, Vantage stores one or more column partition values in a physical row called a container to compress row headers.

If you specify a grouping with ROW format, Vantage stores only one column partition value is stored in a physical row as a subrow, and the row headers are not compressed. This is equivalent to the format that is normally used by Vantage.

If you specify neither COLUMN nor ROW format explicitly or if you specify SYSTEM format explicitly, Vantage determines whether to use COLUMN or ROW format for the column partition.

A column partition value consists of the values of the columns in the column partition for a specific table row.

- If you specify either `AUTO COMPRESS` or `NO AUTO COMPRESS`, Vantage applies or does not apply the appropriate autocompression to the specified physical rows. However, it does apply any user-specified compression and, for column partitions with `COLUMN` format, row header compression, to the physical rows.

Default Format of Column Partitions

The choice of default format is based on the size of the column partition value for the added column partition and other factors such as whether a column partition value for the column partition has fixed or variable length and whether the column partition is a single-column or multicolumn partition.

As a general rule, `COLUMN` format is assigned to narrow column partitions and a `ROW` format to wide column partitions.

You can either submit a `HELP COLUMN` statement or query an appropriate data dictionary view to determine the format chosen for an added column partition. You can also specify an explicit format for a column partition.

If you add a column set to a column partition and that partition has system-determined `COLUMN` or `SYSTEM` partition format, Vantage redetermines the column partition format, basing its choice of format on the size of a column partition value for the altered set of columns in the column partition and other factors such as whether a column partition value for the partition has fixed or variable length.

If you add columns to a column partition and the partition has a user-specified `COLUMN`, `ROW`, or `SYSTEM` format, the system does not change the format for the altered column partition.

Dropping and Adding Column Partitions in the Same ALTER TABLE Statement

If you drop and add column partitions in the same `ALTER TABLE` statement, the system drops those column partitions before adding any new column partitions.

Number of Defined Column Partitions

If you add a column partition, the number of defined column partitions, which always includes 2 column partitions for internal use, is incremented by one and the number that can be added is decremented by 1.

The maximum number of column partitions and the maximum column partition are not affected.

Column Partition COLUMN, ROW, or SYSTEM Format

An individual column partition either has `COLUMN`, `ROW`, or `SYSTEM` format. It cannot have a mix of any formats. However, different column partitions of a column-partitioned object can have different formats.

The column partitions of a column-partitioned table can have all `COLUMN` format, all `ROW` format, all `SYSTEM` format, or a mix of `COLUMN`, `ROW`, or `SYSTEM` formats. The column partitions of a column-partitioned table or join index can be all containers, all subrows, or a mix of containers and subrows.

A column partition number cannot be less than 1 or greater than the maximum column partition number for the table or join index. Column partition numbers might not correspond to the order in which the column partitions were defined for an object.

Regardless of whether you specify an INTO clause, the system always adds a column as the last column for an ASTERISK (*) specification used to select all of the columns from the table.

Adding Columns to a Column Partition in a Column-Partitioned Table

The following rules apply to adding columns to a column partition in a column-partitioned table.

- A column partition number cannot be less than 1 or exceed the maximum column partition number of the table.

Column partition numbers might not correspond to the order that the column partitions were defined.

- While you can use both row and column partitioning for the same table, you cannot mix both formats in the same partitioning level. A column partition either has COLUMN format or it has ROW format.
- If a column to be added is already defined for that column partition, the system returns an error to the requestor.
- If a column to be added is already defined in another column partition, the system returns an error to the requestor.
- If you add columns to a column partition that has user-specified COLUMN, ROW, or SYSTEM format, the format is retained for the altered column partition.
- If you add columns to a column partition, the altered column partition is assigned to a new column partition number.
- Adding columns to a column partition does not affect the number of defined column partitions, which includes the 2 internal use column partitions, the number of column partitions that can be added, the maximum number of column partitions, or the maximum column partition number.
- Vantage bases its choice of system-determined COLUMN or ROW format for an added column partition on the size of the column partition value for the added column partition and other factors such as whether a column partition value for the column partition has fixed or variable length and whether the column partition is a single-column or multicolumn partition.

Vantage generally determines a narrow column partition (defined as 256 or fewer bytes) to have COLUMN format and a wide column partition to have ROW format.

You can use HELP COLUMN requests or retrieve the appropriate rows using a data dictionary view to determine the system-column partition form that Vantage chose for a column partition. See *Teradata Vantage™ - Data Dictionary*, B035-1092 for lists of the views provided by Teradata.

- If you both add and drop column partitions in the same ALTER TABLE request, Vantage drops the specified column partitions before it adds any new column partitions.
- If you add a new column partition to a table, Vantage increments the number of defined column partitions, including the 2 internal use column partitions, by 1, and decrements the number of partitions that can be added by 1.

The maximum number of column partitions and the maximum column partition for the table are not affected by adding a new column partition.

Adding Columns to a Normalized Table

The following rules and restrictions apply to adding columns to a normalized table.

- A table can have only one NORMALIZE column.
- You can normalize an unnormalized table by altering its definition to specify the *ADD normalize_option* definition.
- Adding a new column to a normalized table does not automatically add that column to the *normalize_ignore_columns* list.
- Vantage renormalizes a normalized table if you add a column to the ignore column list.
- When Vantage normalizes a previously unnormalized table, the value for the ignore columns is non-deterministic when multiple rows are normalized as one.
- You can use an ALTER TABLE request to alter the *normalize_ignore_columns* column list by specifying an *ADD normalize_ignore_columns* option. This includes the complete normalize clause and the ignore columns.
- You cannot add and drop normalization from a table within the same ALTER TABLE request.

The following rules and restrictions apply to adding a NORMALIZE option to an existing table.

- The column you specify for the NORMALIZE option must have either a Period data type or a pseudo Period column.
- You cannot specify a TRANSACTIONTIME column as a NORMALIZE column.
- If you do not specify an explicit normalization condition, the default is ON MEETS OR OVERLAPS.
- If an altered table that is to be normalized contains columns with a BLOB, CLOB, JSON, or XML data type, those columns must be specified in the *normalize_ignore_column_name* list for the NORMALIZE option.
- Vantage validates both UNIQUE and PRIMARY KEY constraints in the altered table with normalized rows. If a normalized row violates a UNIQUE or PRIMARY KEY constraint, the system returns an error to the requestor.
- Vantage validates CHECK constraints for an altered table on a row inserted into a normalized table, and if the constraint is violated, the system returns an error to the requestor.

This action prevents a security issue that can occur if a constraint is specified on the beginning or end of a normalized Period column. In this case the input row violates the CHECK constraint but the normalized row does not. This situation cannot occur with UNIQUE constraints.

For information about dropping columns from a normalized table, see [Dropping Columns from a Normalized Table](#).

Maximum Number of Columns in Table

A base table can contain a maximum of 2,048 columns. Up to 32 columns can be defined with a LOB data type. See [Rules and Restrictions for Large Object Data Types](#).

A maximum of 2,560 columns can be defined for a base table over its lifetime. Dropping columns does not affect this rule.

If new columns need to be added that would exceed this number, you must create a new table. You can use the SHOW TABLE statement and INSERT ... SELECT statements to create and load the spin-off table.

Changing Column Attributes: NOT NULL or NULL

When changing columns from NULL to NOT NULL, or from NOT NULL to NULL, refer to the following table:

| Column Attribute | Composite Changes |
|------------------|-------------------|
| NULL | Not permitted. |
| NOT NULL | Permitted. |

The following rules define how columns defined with NULL or NOT NULL attributes can be altered:

| This column type ... | When defined as ... | Can be altered to this column type ... |
|----------------------|---------------------|---|
| Indexed | NULL | NULL only |
| | NOT NULL | NOT NULL only |
| Unindexed | NULL | NOT NULL If and only if the column does not contain nulls. |
| | NOT NULL | NULL |

Modifying Column Data Types or Multivalue Compression

You can compress nulls and as many as 255 distinct values per column. You can compress an unlimited number of columns per table. This value is constrained by the maximum system row length because compressed values are added to the table header row for each column.

| Type of Multivalue Compression | Description |
|--------------------------------|---|
| Single-valued | Default is null if no value is specified explicitly. |
| Multivalued | <ul style="list-style-type: none"> there is no default and all values must be specified explicitly; however, if a column is nullable, then null compression is in effect even if it is not specified explicitly. you can specify the values you want to compress in any order for a column. you can only specify a value once in a multivalue compression list for a column. |

You can add a new column with multivalued compression to an existing table, add multivalued compression to an existing column, or you can drop compression from an existing column by specifying the NO COMPRESS attribute.

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

Columns defined with the COMPRESS attribute cannot participate in fast path INSERT ... SELECT operations, so if you execute an INSERT ... SELECT request on a target table that has multivalue compressed columns, the Optimizer does not specify fast path optimization for the access plan it creates.

Usually, the performance cost of not being able to take advantage of the fast path INSERT ... SELECT is more than offset by the performance advantages of multivalue compression.

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

- Identity
- LONG VARCHAR
- BLOB
- CLOB

If the data type of any column in the new table is not compatible with the value of the corresponding field in the existing table, individual INSERT requests must be used to load each row.

If all the new data types are compatible with all the existing values (for example, only the COMPRESS attribute is being changed), you can use an INSERT ... SELECT request to copy all the rows in a single request. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Also see [Rules for Adding or Modifying Multivalue Compression for a Column](#).

See *Teradata Vantage™ - Database Design*, B035-1094 for more detailed information about multivalue compression and its applications for performance optimization and disk capacity savings.

Adding, Modifying, or Dropping Algorithmic Compression

The following rules apply to using an ALTER TABLE request to add, modify, or drop algorithmic compression for a column.

- You can use an ALTER TABLE request to add a new table column that has only algorithmic compression or that has both value and algorithmic compression.
- You can use an ALTER TABLE request to modify the algorithmic compression characteristics of an existing column only if the table contains no data.
- You cannot use an ALTER TABLE request to add algorithmic compression for a column that has a structured UDT data type, including a column that use embedded services UDFs for compression.
- When you specify a UDF name for algorithmic compression, you must also specify its containing database or user.
- You cannot specify algorithmic compression for a VALIDTIME or TRANSACTIONTIME column that has a Period data type.
- If a table is populated with data, you cannot modify its column compression attributes if either the current or planned compression attribute specifies algorithmic compression.

The following table summarizes the supported compression modification cases.

| Current Compression | Modified Compression | | | |
|----------------------------|----------------------|-----------------|------------------|----------------------------|
| | None | Multivalue Only | Algorithmic Only | Multivalue and Algorithmic |
| None | yes | yes | no | no |
| Multivalue Only | yes | yes | no | no |
| Algorithmic Only | no | no | no | no |
| Multivalue and Algorithmic | no | no | no | no |

- You must specify a compression algorithm and a decompression algorithm for any column specified to have algorithmic compression.

You can specify multivalue compression and algorithmic compression for a column in any order.

- When you specify multivalue compression and algorithmic compression for the same column, Vantage applies algorithmic compression only to values that are not specified with multivalue compression.
- If you specify only algorithmic compression for a column, there is no limit on the size of the specified data type.
- If you specify both algorithmic and multivalue compression for a column, the size of the data type is restricted to the same data type limitations as those for multivalue compression.
- Support for algorithmic compression is restricted to the following data types.

- BLOB
- BYTE
- CHARACTER

Vantage implements the GRAPHIC data type as CHARACTER CHARACTER SET GRAPHIC.

- CLOB
- Geospatial
- VARBYTE
- VARCHAR
- VARGRAPHIC
- JSON
- XML
- Period
- All distinct BLOB-based, CLOB-based, and XML-based UDT types

You can also compress nulls for data in the following types of columns.

- Distinct UDT
- ARRAY/VARRAY
- Period types, but not derived Period types

- You cannot specify algorithmic compression for a column that is a component of the primary index for a table.
- You can specify algorithmic compression for a column that is a component of a secondary index for a table.
- You cannot specify algorithmic compression for a column in a standard referential integrity relationship.
- You can specify algorithmic compression for a column in a Batch referential integrity relationship and for a column that is a component of a Referential Constraint.
- You can specify algorithmic compression for columns in a permanent base table and for columns in a global temporary table.
- Vantage automatically compresses nulls in a column that specifies algorithmic compression.
- The dictionary table column *DBC.TVFields.CompressValueList* contains the names of the algorithmic compression and algorithmic decompression UDFs specified for a column as well as the compression multivalue for the column if it specifies both algorithmic and multivalue compression.

If the size of *CompressValueList* for a table column exceeds 8,192 characters, the system returns an error to the requestor.

- For the rules that apply to writing UDFs to support algorithmic compression and decompression, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- Teradata provides external UDFs for algorithmic compression and decompression. See *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

Adding, Modifying, or Dropping Block-Level Compression

You cannot use an ALTER TABLE statement to add, modify, or drop the block-level compression characteristics of a table or join index. You must use the Ferret utility to change the block-level compression of a table or join index. See *Teradata Vantage™ - Database Utilities*, B035-1102. For a description of the restrictions on specifying block-level compression for volatile and global temporary tables, see [Block-Level Compression and Tables](#).

Data Type and Attribute Definitions

The data type defined for a column can be changed only if the new type does not require a change to the existing data. The rules for changing the data type of a column are listed in [Rules for Changing Column Data Types](#). Also see [Procedure to Change Column Data Types](#).

For an existing column, the following rules are true for data types and attributes with the exception of LOB data types.

You can specify either of the following:

- FORMAT, TITLE, DEFAULT, or WITH DEFAULT phrases.
- Changes in data type attributes or in data types in cases where the change does not require rewriting existing rows.

The following table provides examples of valid and non-valid type and attribute definition changes.

| Changing this form ... | TO this form ... | IS ... |
|------------------------|------------------|-----------|
| CASESPECIFIC | NOT CASESPECIFIC | valid |
| NOT CASESPECIFIC | CASESPECIFIC | valid |
| CASESPECIFIC | UPPERCASE | not valid |
| | | |
| VARCHAR(6) | VARCHAR(12) | valid |
| VARCHAR(12) | VARCHAR(6) | not valid |

For BLOB and CLOB data types, you can only alter the following column attributes.

- NULL
- NOT NULL
- TITLE
- FORMAT

No other data type attributes are permitted for BLOB or CLOB data.

You cannot specify a character data set of KANJI1 for CHARACTER, VARCHAR, or CLOB data types.

Changing a Data Type Attribute

The display attributes FORMAT and TITLE can be altered at any time and a new DEFAULT declared.

The following rule applies to this class of table modifications: you can declare either the CASESPECIFIC or the NOT CASESPECIFIC attribute for character columns.

Rules for Adding or Modifying Multivalue Compression for a Column

The following rules apply to adding or modifying rows with multivalue compression.

- For single-valued compression, the default for the compress value is NULL.
- For multivalued compression, there is no default compress value.
All values in the multivalue must be specified explicitly.
- Nulls in a nullable column are always compressed automatically, even if not explicitly specified.
- You can compress as few as 0 or as many as all but the primary index column set of a table. You cannot compress partitioning column values when the primary index or table is row partitioned.
- The maximum number of distinct values that can be compressed per column is 255.
- You cannot modify or add the definition for a new column results in the table row header exceeding the maximum row length.
- You can modify an existing column to have multivalued compression.

- In addition to the direct method of modifying an existing column to use multivalued compression, there are two workarounds for cases where you want to convert an existing column to support multivalued compression.
 - The procedure for the first workaround is:
 1. Use an ALTER TABLE request to add a new column with the desired multivalued compression.
 2. Populate the new column by copying the data from the column it is designed to replace.
 3. Use ALTER TABLE to drop the old column.
 4. Use ALTER TABLE to rename the new column to match the name of the dropped column.
 - The procedure for the second workaround is:
 1. Use an ALTER TABLE request with the desired multivalued compression defined where needed.
 2. Populate the new table by copying the data from the old table using an INSERT ... SELECT request.
 3. Use a DROP TABLE request to drop the old table.
 4. Use an ALTER TABLE request to rename the new table to match the name of the dropped table.
- You can drop compression from an existing column or make values uncompressible for a new column by specifying the NO COMPRESS option.

| IF the specified column ... | THEN you can use NO COMPRESS to ... |
|-----------------------------|---|
| already exists | drop the compression from the specified column. |
| does not already exist | make the column values not compressible for the specified column. |

- The effect of adding the COMPRESS attribute to an existing column depends on the specific definition of the COMPRESS clause, as is explained by the following cases. Note that you cannot specify null compression if the specified column is defined with the NOT NULL attribute.

- To compress nulls only for the specified column.

```
ALTER TABLE table_name
  ADD column_name COMPRESS;
```

This is equivalent to the following request.

```
ALTER TABLE table_name
  ADD column_name COMPRESS NULL;
```

- To compress nulls and a specified constant value for the specified column.

```
ALTER TABLE table_name
  ADD column_name COMPRESS my_constant;
```

If *column_name* already exists, and a compression multivalue set is already defined for it, then the newly specified compression multivalue set replaces the existing set.

- To compress nulls and the specified list of constant values for the specified column.

```
ALTER TABLE table_name
  ADD column_name COMPRESS (constant_list);
```

If *column_name* already exists, and a compression multivalue set is already defined for it, then the newly specified compression multivalue set replaces the existing set.

- Specifying NO COMPRESS when adding a new column has the same effect as not specifying COMPRESS for that column. The column cannot be defined with multivalue compression within the same request that adds the new column.
- Specifying NO COMPRESS when modifying an existing column drops the COMPRESS attribute from that column.
- The ALTER TABLE ADD COMPRESS syntax supports the following tables and columns.
 - Loaded and empty tables of all types.
 - Global Temporary tables.
 - Secondary index columns unless the column is the PRIMARY KEY or FOREIGN KEY in a referential constraint, in which case it cannot be compressed.
 - Base table columns on which a join index is defined.

If you do this, the system returns a warning message advising you that you must recreate the join index to reflect the compression changes.

- Base table columns on which a hash index is defined.
- Modification of multiple columns in a single ALTER TABLE request. For example, the following ALTER TABLE request is valid.

```
ALTER TABLE table_ex
  ADD column_1 COMPRESS (0,100, 200),
  ADD column_2 COMPRESS ('Female','Male'),
  ADD column_3 COMPRESS ('Teradata');
```

- Specification of changes to multiple options within the same ALTER TABLE request. For example, the following ALTER TABLE request is valid.

```
ALTER TABLE table_ex
  ADD column_1 COMPRESS (0,100, 200),
  DROP column_2,
```

```
ADD column_3 COMPRESS ('Teradata'),
ADD column_4 INTEGER;
```

- Changing an noncompressed column to have multivalue compression. For example,

```
ALTER TABLE table_ex
ADD column_4 COMPRESS (0, 1000, 10000);
```

- Changing a compressed column to not being compressed. For example,

```
ALTER TABLE table_ex
ADD column_4 NO COMPRESS;
```

- Adding compressed values to an existing multivalue compression. This can only be done by *replacing* the existing compression multivalue for a column. For example, suppose *column_3* in *table_ex* already compresses the following set of values: WalMart, JCPenney, Kmart.

The following ALTER TABLE request adds Harrahs to the list of compressed values for *column_3*.

```
ALTER TABLE table_ex
ADD column_3
COMPRESS ('WalMart', 'JCPenney', 'Kmart', 'Sears', 'Harrahs');
```

- Dropping compressed values from an existing compression multivalue list. This can only be done by *replacing* the existing compression multivalue list for a column. For example, suppose *column_3* in *table_ex* already compresses the following set of values: WalMart, JCPenney, Kmart, Sears, Harrahs.

The following ALTER TABLE request drops Harrahs from the list of compressed values for *column_3*.

```
ALTER TABLE table_ex
ADD column_3
COMPRESS ('WalMart', 'JCPenney', 'Kmart', 'Sears');
```

- The ALTER TABLE MODIFY COMPRESS syntax does not support the following features.
 - A compression multivalue list that exceeds the maximum size of 8,192 characters as defined by the column *DBC.TVField.CompressValueList*.
 - A compression multivalue list that causes the table header to exceed its maximum size of 128 kilobytes.
 - A compression multivalue list that causes the Transient Journal to exceed its maximum size of 65,535 bytes.
 - A compression multivalue list with values that are incompatible with the specified data type for the column. An example of this would be attempting to add a character string to a multivalue compression for a column typed as INTEGER.
 - A compression multivalue that exceeds the maximum number of 255 unique values for a column.

- NULL compression when the column has a NOT NULL attribute.
- A compression value or multivalue that contains a character that is not in the character set for the current session.
- A compression value that duplicates a value that is already in the compression multivalue. An example of this would be attempting to add the string 'WalMart' twice to a compression multivalue for a CHARACTER column.

When this occurs, the request aborts and the system returns an error to the requestor.

- Modifying the compression multivalue characteristics of the same column more than once in the same ALTER TABLE request.

When this occurs, the request aborts and the system returns an error to the requestor.

- Modifying both the multivalue compression characteristics and the characteristics of a constraint in the same ALTER TABLE request.

When this occurs, the request aborts and the system returns an error to the requestor.

- Modifying a primary index column, whether nonpartitioned or partitioned, to add multivalue compression because the columns of a primary index cannot be value-compressed.

When this occurs, the system returns an error to the requestor.

- Adding multivalue compression to a column that is a member of the partitioning column set for a row-partitioned table.

When this occurs, the system returns an error to the requestor.

- Modifying any column that is part of a referential integrity constraint to add multivalue compression.

When this occurs, the request aborts and the system returns an error to the requestor.

- Modifying an identity column to add multivalue compression because you cannot value compress identity columns.

When this occurs, the request aborts and the system returns an error to the requestor.

- Modifying a column that has any of the following data types to add multivalue compression.
 - LONG VARCHAR
 - BLOB
 - CLOB
 - A UDT, no matter what its underlying data type (or, if a structured UDT, data type set) contains.

When this occurs, the request aborts and the system returns an error to the requestor.

- The expanded compress values for a column are stored in the data dictionary.

If the text exceeds the 8,192 character maximum size for DBC.TVFields.CompressValueList, then the ALTER TABLE request fails.

- The list of compressed values for a nullable column can be null or one or more distinct constant values.

- The list of compressed values for a non-nullable column cannot include nulls.
- You cannot specify the same distinct value more than once in a compressed multivalue.

Adding and Dropping CHECK Constraints

The following rules apply to using CHECK constraints with the ALTER TABLE statement.

- The following table explains rules for ADD and MODIFY.

| THIS form ... | IS allowed only if <i>column_name</i> does ... |
|---|--|
| ADD <i>column_name</i> CHECK (<i>boolean_condition</i>) | <i>not</i> already have a constraint. |
| MODIFY | have a constraint. |

Note:

Only one CHECK modification can be made per ALTER TABLE request and that CHECK and non-CHECK alterations cannot be combined within a single ALTER TABLE request.

- To drop all unnamed column level CHECK constraints on *column_name*:

```
ALTER TABLE table_name DROP column_name CHECK
```

- To drop all unnamed table level CHECK constraints on the table name:

```
ALTER TABLE table_name DROP CHECK
```

- To drop a named CHECK constraint, use either of the following statements:

```
DROP CONSTRAINT name CHECK
DROP CONSTRAINT name
```

- The following form is valid only if a constraint with *name* already exists in the table.

```
MODIFY CONSTRAINT name CHECK (search condition)
```

This also applies to the named constraints defined as part of the column definition because those constraints are handled as named table-level constraints.

- To ensure maximum system performance, there is a limit of 100 table-level constraints that can be defined for any table.

A combination of table-level, column-level, and WITH CHECK on view constraints can create a constraint expression that is too large to be parsed for INSERT and UPDATE requests.

- CHECK constraints are not supported for global temporary tables.
- Details of adding and dropping constraints with ALTER TABLE are explained in the following table.

| IF you execute this statement ... | THEN you add/drop the following number of unnamed table level CHECK constraints ... |
|-----------------------------------|---|
| ALTER TABLE ... DROP CHECK | all. |
| anything else | one per ALTER TABLE request, regardless of constraint type. |

- A CHECK constraint can neither be defined for nor reference a LOB column.

Dropping Columns From a Table

The following rules and restrictions apply to dropping columns from a table:

- You cannot drop all of the columns from a table.
- When you drop a column, the database deletes the field corresponding to the dropped column in every row in the table.
- You cannot drop indexed columns from a table without first dropping the index on those columns.

The following set of procedures explain what you must do to drop an indexed column from a table.

| To drop a column on which this type of index is defined ... | Follow this procedure ... |
|---|---|
| Primary: the table has no rows. | <ol style="list-style-type: none"> 1. Use the primary index or primary AMP index modification syntax of the ALTER TABLE statement to modify the table definition to create a new primary index or primary AMP index. 2. Use ALTER TABLE <i>table_name</i> DROP <i>column_name</i> to drop the column set that was the former primary index or primary AMP index. |
| Primary: using the CREATE TABLE AS statement | <ol style="list-style-type: none"> 1. Copy the table into a newly defined table defined with a different primary index or primary AMP index using the CREATE TABLE AS syntax. 2. Drop the original table. 3. Rename the new table. |
| Primary: legacy method | <ol style="list-style-type: none"> 1. Create a new table with the correct primary index or primary AMP index. 2. Copy the data into the new table. A simple way to do this is to use an INSERT ... SELECT request. See <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146. 3. Drop the original table. 4. Rename the new table. |
| <ul style="list-style-type: none"> • Secondary • Hash • Join | <ol style="list-style-type: none"> 1. Drop the index. 2. Drop the index columns. 3. Define a new index on the correct columns. |

- You cannot drop a partitioning column for a row-partitioned table without first modifying the partitioning to exclude that column from the partitioning set. Otherwise, the database returns an error.

You cannot drop a column on which a primary, primary AMP index, secondary, hash, or join index is defined.

- You cannot drop columns that are referenced in the UPDATE OF clause of a trigger.
- You can drop the identity column from an existing table.
- You can also drop only the identity column *attribute* from an identity column in an existing table, retaining the column and its data.
- You can drop LOB columns from a base table without restrictions. If you drop all the LOB columns from a table, the table is not bound by the LOB restrictions. See [Rules and Restrictions for Large Object Data Types](#).
- You cannot drop the QITS column from a queue table.

Dropping Columns from a Column-Partitioned Table

The following rules and restrictions apply to dropping columns from a column-partitioned table.

- A column partition number cannot be less than 1 or greater than the maximum column partition number for the table or join index. Column partition numbers might not correspond to the order that the column partitions were defined.
- You cannot drop all of the columns of a column-partitioned table other than the 2 internal use partitions, the system returns an error to the requestor.
- You cannot drop the 2 internal use column partitions from a table.
- If you drop all of the columns from a column partition, the following things occur.

- The column partition is dropped.
- The column partition number from the dropped column partition becomes available for adding another column partition.

The system decrements the number of defined column partitions by 1 and increments the number of column partitions that can be added by 1.

The maximum number of column partitions and the maximum column partition number for the table are unaffected.

- If you add multiple column partitions are in the same ALTER TABLE request that deletes column partitions, the system drops the specified column partitions before adding the new column partitions.
- You can drop all the existing columns from a column partition, but the system does not drop the partition if you also add new columns to the column partition in the same ALTER TABLE request.
- If you drop columns from a column partition and there are other columns in the altered column partition:
 - If you drop a column set from a column partition and that column partition has system-determined column partition format, the system redetermines the column partition format based on the size of a column partition value for the remaining set of columns in the column partition and other factors such as whether a column partition value for the column partition has fixed or variable length.

Vantage generally determines a narrow column partition (defined as 256 or fewer bytes) to have COLUMN format and a wide column partition to have ROW format.

You can use HELP COLUMN requests or retrieve the appropriate rows using a data dictionary view to determine the system-column partition form that Vantage chose for a column partition. See *Teradata Vantage™ - Data Dictionary*, B035-1092 for lists of the views provided by Teradata.

- If the column partition from which the column set was dropped has user-specified COLUMN, ROW, or SYSTEM format, Vantage does not change the format for the altered column partition.
- Vantage assigns a different column partition number to the altered column partition.
- The number of defined column partitions, including the 2 internal use column partitions, the number of column partitions that can be added, the maximum number of column partitions, and the maximum column partition number are not affected by these operations.

Dropping Columns from a Normalized Table

The following rules and restrictions apply to dropping columns from a normalized table.

- You can drop any column from a normalized table, including ignore columns.
- Dropping a normalized Period column also drops the NORMALIZE clause from the table definition.
- Vantage renormalizes a normalized table if you drop a column that is not specified in the ignore column list from the table or when a column is added to the ignore column list.
- You can use an ALTER TABLE request to alter the *normalize_ignore_columns* column list by specifying the DROP NORMALIZE option. This include the complete list - normalize clause and the ignore columns.
- You cannot add and drop normalization from a table within the same ALTER TABLE request.

For information about adding columns to a normalized table, see [Adding Columns to a Normalized Table](#).

Renaming Table Columns

You can rename table columns except when the:

- *new_column_name* matches any existing column name in the target table.
- Column is an index field.
- Column is part of any type of referential constraint.
- Column is a partitioning column for a partitioned primary index.
- Column is referenced in the UPDATE OF clause of a trigger.

Adding or Dropping PRIMARY KEY and UNIQUE Constraints

ALTER TABLE ... ADD PRIMARY KEY or UNIQUE constraint requires the following privileges:

- DROP TABLE on the table or its database
- INDEX on the table

The following rules apply to adding and dropping PRIMARY KEY and UNIQUE constraints.

- You can add a PRIMARY KEY or UNIQUE constraint that specifies the same columns as a unique primary index (UPI).
- You can add a PRIMARY KEY or UNIQUE constraint that specifies the same columns as a primary AMP index (PA).
- You can add a PRIMARY KEY or UNIQUE constraint that specifies the same columns as a unique secondary index (USI).
- You can add a PRIMARY KEY or UNIQUE constraint that specifies the same columns as a nonunique primary index (NUPI) if the NUPI includes all the partitioning columns or there are no partitioning columns. The constraint is enforced by a unique secondary index.
- You cannot add a PRIMARY KEY or UNIQUE constraint to the (Queue Insertion TimeStamp) QITS column of a queue table.
- You cannot define a PRIMARY KEY or UNIQUE constraint for a LOB column.
- Use the DROP INDEX statement to drop unnamed UNIQUE constraints.
- You can drop a named PRIMARY KEY or UNIQUE constraint that specifies the same columns as a unique primary index (UPI).
- You can drop a named PRIMARY KEY or UNIQUE constraint that specifies the same columns as a primary AMP index (PA).

Adding a PRIMARY KEY or UNIQUE Constraint to a Table With a NUPI

For a table with a nonunique primary index (NUPI), you can only add a PRIMARY KEY or UNIQUE constraint on the same set of columns as the primary index column list if a unique secondary index (USI) is not explicitly defined on the same column set as the primary index. A PRIMARY KEY or UNIQUE constraint implicitly defines a USI on the same set of columns as those defining the primary index column list.

The system creates a USI on the specified column set, and the system can then use the index to validate that the primary index columns are unique. You should only use this alteration as a temporary solution for making a NUPI unique. When you alter a table in this way, the system returns a warning message and the CREATE TABLE SQL text returned by a SHOW TABLE request is not valid.

You should immediately alter the table using the MODIFY PRIMARY clause to change its primary index from a NUPI to a UPI. The system drops the USI that was defined on the NUPI because it is no longer needed to enforce the uniqueness of the primary index.

Adding or Dropping Standard Referential Integrity Constraints

Referential integrity is a rule that prevents rows in a child table from being orphaned from the rows they refer to in a parent table. When a referential integrity relationship is established between a primary or alternate key for a parent table and a foreign key in a child table, the system does not permit you to update or delete the referenced parent table key without first updating or deleting rows containing the foreign key in the child table.

The following rules apply to adding and dropping referential integrity constraints.

- The user issuing the ALTER TABLE request containing a REFERENCES option must have the REFERENCES privilege on the referenced table or on all specified columns of the referenced table.

- You can only add or drop one foreign key reference and it cannot contain any options other than to add or drop the foreign key.
- The columns in the referenced parent table must be defined uniquely, either as the primary key for the table, with a UNIQUE column attribute, or as a USI.

This rule is *not* mandatory for Referential Constraints, where the candidate key acting as the primary key for the referenced table in the constraint need not be explicitly declared to be unique. See [Referential Constraints](#).

- The foreign key columns in the referencing child table must be identical to the key in the referenced table.
- You cannot define a referential constraint on a LOB column in a child table, nor can you define a referential constraint to a LOB column in the parent table.
- You cannot add either of the following constraints to a queue table definition.
 - FOREIGN KEY ... REFERENCES
 - REFERENCES
- You cannot add either of the following constraints to a non-queue table definition if the target table for that reference is a queue table.
 - FOREIGN KEY ... REFERENCES
 - REFERENCES
- You cannot compress either key value in a referential relationship. The system does not compare column-level constraints for the referentially related columns.
- Referential integrity is not supported for global temporary, trace, queue, or volatile tables.

See *Teradata Vantage™ - Database Design*, B035-1094 for more information about general aspects of referential integrity and its maintenance.

The following process describes the error handling processes involved in adding any new referential constraint, whether it checks referential integrity or not, to a table.

1. The system generates an error table when the new constraint is added to the table. This is a different error table than the one you create to log minibatch bulk loading errors (see [CREATE ERROR TABLE](#)).
 - Its name is the name of the target table suffixed with the appropriate reference index number.
A different reference index number is assigned to each foreign key constraint defined for the table. You can determine reference index numbers using the *RI_Distinct_ChildrenV* or *RI_Distinct_ParentsV* system views (see *Teradata Vantage™ - Data Dictionary*, B035-1092).
 - Its columns and primary index are the same as those of the target table. If the primary index is partitioned, then the partitioning columns in the parent and child tables must also be the same.
2. The error table is created under the same user or database as the table being altered.

| IF ... | THEN the system ... |
|---|---------------------|
| a table with the same name as that generated for the error table already exists | returns an error. |

| IF ... | THEN the system ... |
|---|---|
| rows in the referencing table contain values in the foreign key columns that cannot be found in any row of the referenced table | does not return an error. Instead, a row is inserted into the error table for each such row found in the target table. Use these rows to determine the corrections that must be made. |

You are responsible for correcting values in the referenced or referencing tables so that full referential integrity exists between the two tables.

You are also responsible for maintaining the error table, including deleting it after any errors have been corrected.

Adding or Dropping Batch Referential Integrity Constraints

Batch referential integrity is similar to standard referential integrity because it enforces the referential integrity constraints it defines. This constraint differs from standard referential integrity with respect to when it is validated and how non-valid rows are treated. The rules for adding or dropping batch referential integrity constraints are identical to those documented by [Adding or Dropping Standard Referential Integrity Constraints](#).

Adding or Dropping Referential Constraints

Referential Constraints provide the Optimizer with an efficient method for producing better query plans that take advantage of defined, but unenforced, referential relationships.

Other than not enforcing referential integrity and not requiring the candidate key for the referenced table in the constraint to be explicitly declared to be unique, the rules for Referential Constraints (Referential Constraints are sometimes called soft referential integrity constraints.) are identical to those documented by [Adding or Dropping Standard Referential Integrity Constraints](#). See [Referential Constraints](#) for additional information.

Maintaining Foreign Key Integrity

Vantage verifies that the structure of columns defined as foreign keys, or referenced by foreign keys, is not changed in any manner that violates the rules for the definition of a foreign key constraint.

An ALTER TABLE (or DROP INDEX) request attempting to change the structure of such a column aborts and causes the system to return an error to the requestor.

Vantage verifies that a table referenced by another is not dropped. Before dropping the table, you must perform an ALTER TABLE request to drop the foreign key reference.

Referential Integrity When a Table Is Inconsistent

While a table is marked as inconsistent, no updates, inserts, or deletes are permitted. The table is fully usable only when the inconsistencies are resolved. This restriction is true for both hard and soft (Referential Constraint) referential integrity constraints.

It is possible that the user either intends to or must revert to a definition of a table which results in an inconsistent reference on that table.

After the child table is archived, the parent table can be dropped.

When the child table is restored, the parent table no longer exists. The normal ALTER TABLE DROP FOREIGN KEY request does not work, because the parent table references cannot be resolved.

You can use the DROP INCONSISTENT REFERENCES option to remove these inconsistent references from a table.

The syntax is as follows.

```
ALTER TABLE database_name.table_name DROP INCONSISTENT REFERENCES
```

You must have DROP privilege on the target table of the request to perform this option, which removes all inconsistent internal indexes used to establish references.

Rules for Changing Column Data Types

The table on the following pages illustrates the rules for changing a data type from the existing type to a new type.

Notice that there is not a direct correspondence between whether a source type can be explicitly cast to a target type (see *Teradata Vantage™ - Data Types and Literals*, B035-1143) and whether you can change an existing column data type to a new type using an ALTER TABLE request.

You cannot *change* the character set attribute of an existing column using ALTER TABLE requests. For example, once a column has been defined as LATIN, you cannot change it to UNICODE.

| Old Data Type | New Data Type | Restrictions on Changes |
|--|--|--|
| All | All | Cannot change the data type for a column specified in a primary or secondary index. |
| <ul style="list-style-type: none"> CHARACTER(<i>n</i>), UC CHARACTER(<i>n</i>) | <ul style="list-style-type: none"> CHARACTER(<i>n</i>) CHARACTER(<i>n</i>), CS | Lengths must remain identical. You can only change a case-specific setting. |
| CHARACTER | CHARACTER(<i>n</i>) | Not allowed if $n > 1$. |
| CHARACTER | DATE | Value trimmed from leading and trailing blanks and handled like a string literal in the declaration of the DATE string literal. If conversion is not possible, the system returns an error. |

| Old Data Type | New Data Type | Restrictions on Changes |
|--|--|---|
| CHARACTER | TIME [WITH TIME ZONE] | Value trimmed from leading and trailing blanks and handled like a string literal in the declaration of the TIME string literal. If conversion is not possible, the system returns an error. |
| CHARACTER | TIMESTAMP [WITH TIME ZONE] | Value trimmed from leading and trailing blanks and handled like a string literal in the declaration of the TIMESTAMP string literal. If conversion is not possible, the system returns an error. |
| CHARACTER | INTERVAL | Value trimmed from leading and trailing blanks and handled like a string literal in the declaration of the INTERVAL string literal. If conversion is not possible, the system returns an error. |
| <ul style="list-style-type: none"> CHARACTER VARCHAR | PERIOD | Not allowed. |
| VARCHAR(<i>m</i>), UC | VARCHAR(<i>n</i>) | Cannot decrease the maximum length. Cannot change from no UC to UC. |
| GRAPHIC | CHARACTER(<i>n</i>) CHARACTER SET GRAPHIC | Not allowed if <i>n</i> > 1. |
| VARGRAPHIC(<i>m</i>) | VARCHAR(<i>n</i>) CHARACTER SET GRAPHIC | Cannot decrease the maximum length. |
| BYTE | BYTE(<i>n</i>) | Not allowed if <i>n</i> > 1. |
| VARBYTE(<i>m</i>) | VARBYTE(<i>n</i>) | Cannot decrease the maximum length. |
| Exact NUMERIC | INTERVAL | INTERVAL must have a single field only and its numeric value must be in the range permitted for an INTERVAL value. Otherwise, the system returns an error. |
| INTEGER | DATE | Cannot make this change because an INTEGER column could include non-valid DATE values. |
| DECIMAL(<i>n</i> ,0) | INTEGER | Can change only if <i>n</i> is in the range 5 - 9, inclusive. |
| DECIMAL(<i>n</i> ,0) | BIGINT | Can change only if <i>n</i> is in the range 10 - 18, inclusive. |
| DECIMAL(<i>n</i> ,0) | SMALLINT | Can change only if <i>n</i> is 3 or 4. |
| DECIMAL(<i>n</i> ,0) | BYTEINT | Can change only if <i>n</i> = 1 or 2. |

| Old Data Type | New Data Type | Restrictions on Changes |
|---|--------------------------------|--|
| DECIMAL(<i>n</i> , <i>f</i>) | DECIMAL(<i>m</i> , <i>f</i>) | Can change only as follows. <ul style="list-style-type: none"> • $m \geq n$, no change in <i>f</i> • $n = 1$ or 2, $m < 3$ • $n = 3$ or 4, $m < 5$ • $n = 5$ to 9, $m < 10$ • $n = 10$ to 15, $m \leq 18$ |
| DATE | INTEGER | No restrictions. |
| DATE | DATE | No restrictions. |
| DATE | CHARACTER | No restrictions. Result might not be in ANSI date format. |
| DATE | TIME [WITH TIME ZONE] | No restrictions. |
| DATE | TIMESTAMP [WITH TIME ZONE] | Year, month, and day are taken from source date value. Hour, minute, and seconds are set to 0. If target specifies TIMESTAMP WITH TIME ZONE, the time zone fields are taken from the explicit or implicit values for the source. |
| DATE | PERIOD | Not allowed. |
| TIME [WITH TIME ZONE] | TIME [WITH TIME ZONE] | If target is TIME WITH TIME ZONE, then time zone displacement value is added. Otherwise, value is adjusted to the current time zone displacement for the session. |
| <ul style="list-style-type: none"> • TIME [WITH TIME ZONE] • TIMESTAMP [WITH TIME ZONE] • INTERVAL | CHARACTER(<i>n</i>) | <i>n</i> must be \geq the length of the value. If $n >$ the value, then trailing blanks are added. If $n <$ the value, an error is reported. |
| <ul style="list-style-type: none"> • TIME [WITH TIME ZONE] • TIMESTAMP [WITH TIME ZONE] • INTERVAL | VARCHAR | Same as conversion to CHARACTER except no trailing blanks are added when $n >$ value. |
| TIME [WITH TIME ZONE] | TIMESTAMP [WITH TIME ZONE] | No restriction. Year, month, and day are set to the values for CURRENT_DATE. If target also specifies TIME ZONE, the time zone values are taken from the explicit or implicit values for the source. |
| TIME | PERIOD | Not allowed. |
| TIME WITH TIME ZONE | TIME | No restriction, but TIME ZONE value is removed. |

| Old Data Type | New Data Type | Restrictions on Changes |
|---|--|--|
| <ul style="list-style-type: none"> • TIMESTAMP • TIMESTAMP WITH TIME ZONE | DATE | Result is year, month, and day from TIMESTAMP value after adjustment for TIME ZONE, if present. |
| TIMESTAMP [WITH TIME ZONE] | TIMESTAMP [WITH TIME ZONE] | No restriction. If source and target differ on value for WITH TIME ZONE, conversion is required, which can change the values in the result. |
| TIMESTAMP [WITH TIME ZONE] | TIME | Result is taken from hour, minute, and second fields. If target is TIME WITH TIME ZONE, then time zone displacement is added. Otherwise value is adjusted to the current time zone displacement of the session. |
| TIMESTAMP [WITH TIME ZONE] | PERIOD | Not allowed. |
| INTERVAL (<i>n</i>) | INTERVAL (<i>m</i>) | Can change only if $m \geq n$ and is YEAR-MONTH- or DAY-TIME-compatible. You cannot mix YEAR-MONTH intervals with DAY-TIME intervals. |
| PERIOD | <ul style="list-style-type: none"> • CHARACTER(<i>n</i>) • VARCHAR(<i>n</i>) | Not allowed. |
| PERIOD | PERIOD | Not allowed. |
| PERIOD | DATE | Not allowed. |
| PERIOD | TIME [WITH TIME ZONE] | Not allowed. |
| PERIOD | TIMESTAMP [WITH TIME ZONE] | Not allowed. |
| UDT | Not allowed. | Not allowed. You cannot convert a column UDT data type to any other data type using an ALTER TABLE request. |
| <ul style="list-style-type: none"> • ARRAY (one-dimensional) • VARRAY (one-dimensional) | Not allowed. | Not allowed. You cannot convert a one-dimensional column ARRAY data type to any other data type using an ALTER TABLE request. |
| <ul style="list-style-type: none"> • ARRAY (multidimensional) • VARRAY (multidimensional) | Not allowed. | Not allowed. You cannot convert a multidimensional column ARRAY data type to any other data type using an ALTER TABLE request. |

Procedure to Change Column Data Types

To make a change to a column data type that affects existing column data, use the following procedure.

1. Create a new table with a different name that contains the changed data type attributes.
2. Populate the new table using an INSERT ... SELECT request.
3. Catalog the privileges of the old table before step 4. Use the following syntax.

```
SELECT username, accessright, grantauthority, columnname,
       allnessflag
FROM dbc.allrightsV
WHERE tablename = 'table_name'
AND   databasename = 'database_name';
```

4. Drop the old table.
5. Rename the new table with the name of the old table.

As an example, use the following sequence of requests to expand the data type attribute for the name column from 12 to 14 characters.

```
CREATE TABLE temp, FALLBACK (
  EmpNo INTEGER NOT NULL FORMAT ZZZZ9,
  Name  CHARACTER(14) NOT NULL,
  ...
  HCap  BYTEINT FORMAT 'Z9')
UNIQUE PRIMARY INDEX (index_name) ;
```

6. Once the table is created, populate it, drop the old *employee* table, and rename the temporary table.

The following example shows how this is done.

```
INSERT INTO temp
  SELECT *
  FROM employee;
DROP TABLE employee;
RENAME TABLE temp TO employee;
```

A different name, temp, is used in recreating the *employee* table because the employee table already exists. If you submit a CREATE TABLE request for an existing table, you receive an error.

To facilitate recreating a table according to Step 1, you can display the CREATE TABLE DDL text for the table by submitting a SHOW TABLE request.

For example, the following request displays a reconstruction of the CREATE TABLE DDL text for table *dept_em*.


```
SHOW TABLE dept_em;
```

If you are using BTEQ, you can change the SQL text (that is, change the table name and index) using BTEQ edit commands and submit the new table definition using the BTEQ SUBMIT command.

7. Grant the same privileges to the *employee* table again, as they were lost when the table was dropped. Use the same privileges cataloged in step 3.

Increasing the length of VARCHAR or VARBYTE does not change the current format. Instead, you should consider including a new FORMAT phrase in the request.

ALTER TABLE provides many extensions not offered by ANSI SQL. For example, changing column attributes in ANSI SQL is restricted to setting or dropping a default clause, which is not the case for SQL in the database.

Effect of Changing Column Data Type on Statistics

If you change the data type of a column, Vantage no longer uses the statistics that reference the column. Recollecting the statistics on the modified column makes them usable again.

ALTER TABLE (MODIFY Option)

Redefining the Primary, Primary AMP, or Partitioning for a Table

To redefine the PRIMARY, PRIMARY AMP, NO PRIMARY and PARTITION BY or NOT PARTITIONED clauses for a table, use one of the following methods:

- Use the ALTER TABLE MODIFY PRIMARY or PRIMARY AMP syntax to modify the primary index or primary AMP index for the table. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144, [General Rules for the MODIFY PRIMARY Clause](#), and [Modifying the Partitioning of a Table or Join Index](#).
- To change the partitioning for a table, you can specify MODIFY without specifying PRIMARY, PRIMARY AMP, or NO PRIMARY. The PRIMARY, PRIMARY AMP, or NO PRIMARY keywords are only required to modify the primary index or primary AMP index of a table.
- Create a new table with the primary index, primary AMP index, or partitioning defined properly and then use an INSERT ... SELECT to copy the data from the old table to the new table. See [Redefining Primary Index, Primary AMP Index, or Partitioning without ALTER TABLE](#).
- Use the CREATE TABLE AS syntax to:
 1. Create a new table with the primary index or primary AMP index and partitioning, if required.
 2. Copy the rows and, if desired, the statistics from the old table into the new table.

See [CREATE TABLE \(AS Clause\)](#).

You can use ALTER TABLE to perform the following modifications to a primary index, primary AMP index, or partitioning for a table. See [General Rules for the MODIFY PRIMARY Clause](#), [Modifying the Partitioning of a Table or Join Index](#), and [Rules For Altering a Partitioning For a Table](#).

| To perform this operation ... | Specify ... |
|--|---|
| Change a NUPI to a UPI | MODIFY UNIQUE PRIMARY or PRIMARY AMP in the MODIFY PRIMARY clause. |
| Change a UPI to a NUPI | MODIFY NOT UNIQUE PRIMARY or PRIMARY AMP in the MODIFY PRIMARY clause. |
| Change a partitioned object to a nonpartitioned object | NOT PARTITIONED in the MODIFY PRIMARY clause. The table must be empty. |
| Change a nonpartitioned object to a partitioned object | a PARTITION BY clause in the MODIFY PRIMARY clause. The table must be empty. |
| Add or drop partitioning expression ranges | an ADD RANGE or DROP RANGE clause within the Primary Index Change Options clause. See Modifying Partitioning Using the ADD RANGE and DROP RANGE Options . |
| Add or drop primary index or primary AMP index columns | MODIFY PRIMARY clause. The table must be empty. |

| To perform this operation ... | Specify ... |
|---|--|
| Validate partitioning by regenerating table headers and correcting any errors in row partitioning | a REVALIDATE PRIMARY INDEX clause within the REVALIDATE Options clause. See General Rules and Restrictions for the REVALIDATE Option . |

If you alter a table to have a UPI and there is already a USI defined on the same column set, the system drops the USI automatically during the ALTER TABLE operation.

A primary index cannot contain any BLOB, CLOB, ARRAY, VARRAY, Period, or Geospatial columns.

You cannot change the primary index for an error table. See [CREATE ERROR TABLE](#).

Redefining Primary Index, Primary AMP Index, or Partitioning without ALTER TABLE

To redefine the primary index or partitioning for a table without using an ALTER TABLE request, see [Procedure to Change Column Data Types](#). Refer to one of the following methods.

First method

1. Copy the table into a newly defined table defined with a different primary index or primary AMP index (or without a primary index) and populate it using the CREATE TABLE AS syntax. See [CREATE TABLE \(AS Clause\)](#).
2. Catalog the privileges on the old table. See [Procedure to Change Column Data Types](#).
3. Drop the original table. See [DROP TABLE](#) and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
4. Rename the new table. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
5. Grant privileges on the new table. See *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

Second method

1. Create a new table with a different name that specifies the new index.
2. Populate the new table using an INSERT ... SELECT request. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
3. Catalog the privileges on the old table. See [Procedure to Change Column Data Types](#).
4. Drop the original table. See [DROP TABLE](#) and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
5. Rename the new table with that of the old table. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
6. Grant privileges on the new table. See *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

Altering a Primary Index or Primary AMP Index Using ALTER TABLE

The following operations are fairly quick:

- Altering an empty table.
- Altering a table with a primary index or primary AMP index to have a unique or nonunique primary index.
- Altering a table to change the primary index or primary AMP index name.

Dropping or Adding New Ranges or Partitions to a Row-Partitioned Table

Schedule operations that alter table partitions and partition ranges to avoid impact on your production workload. When dropping or adding new ranges or partitions to a row-partitioned table, refer to the following performance considerations:

- When dropping or adding new ranges or partitions in a populated table, the operation can be fairly quick because the rows remain in the retained ranges and partitions.
- There is a small amount of overhead if dropped ranges and partitions are populated, and still further overhead if any referential integrity constraints are defined on the table.
- There is additional overhead if new ranges are added which are populated with NO RANGE or UNKNOWN partitions or rows in dropped ranges that must be moved to the added ranges.
- You must update any secondary, join, or hash indexes on the table.

Updating secondary, join, and hash indexes can be lengthy operations, depending on several factors, including the size of the table and indexes and the number of rows deleted or moved.

- There is additional overhead if the deleted rows are inserted into a save table. The amount of overhead depends on the number of rows that must be inserted into the save table and the other standard performance issues associated with inserting rows into a table.
- If a table is defined with a NO RANGE partition, specifying a WITH DELETE or WITH INSERT INTO clause in an ALTER TABLE statement has no effect.

Rows from deleted partitions and rows whose partition number evaluates to a value other than 1 through 65,535 for 2-byte partitioning or 1 through 9,223,372,036,854,775,805 for 8-byte partitioning are retained in the NO RANGE partition rather than being moved to the target table specified in the WITH DELETE or WITH INSERT INTO clause.

For additional information, see [Purpose and Behavior of the NO RANGE and UNKNOWN Partitions](#) and [Rules For Altering a Partitioning For a Table](#).

General Rules for the MODIFY PRIMARY Clause

Refer to the following rules for the MODIFY PRIMARY INDEX or MODIFY PRIMARY AMP INDEX clause. For partitioning expressions, see [Rules For Altering a Partitioning For a Table](#).

- You must specify `MODIFY PRIMARY INDEX` or `MODIFY PRIMARY AMP INDEX` when you are altering the component columns of the primary index or primary AMP index for a table.

To modify only the partitioning for a table, you can specify `MODIFY` without `PRIMARY INDEX` or `PRIMARY AMP INDEX`.

- You cannot alter the primary index or primary AMP index for any of the following database objects using an `ALTER TABLE` statement.
 - Global temporary tables
 - Volatile tables
 - Join indexes
 - Hash indexes
- You cannot modify the primary index, primary AMP index, or partitioning for a table and other table definitions within a single `ALTER TABLE` statement because the operations are mutually exclusive.
- You cannot alter a populated table with a primary index or primary AMP index to have a different set of primary index or primary AMP index columns.
- If you alter a table to have a unique primary index and there is currently a unique secondary index on the same column set, the database automatically drops the unique secondary index.
- At least one specified option must change the definition of the primary index or the system returns an error.

Specifying different partitioning expressions for an existing partitioned table or join index is considered a change even if the results of the new expressions are identical to those produced by the former partitioning expressions.

- If you do not specify a column set for the primary index or primary AMP index, the existing column set for the primary index or primary AMP index is retained.
- If the resulting primary index or primary AMP index of a modification is unique and if there is also an existing USI on the same column set, whether explicitly or implicitly defined, the USI and any associated `PRIMARY KEY` or `UNIQUE` constraint is dropped because it is redundant.
- If the resulting primary index if a modification is nonunique and the previous primary index was implicitly defined by a `PRIMARY KEY` or `UNIQUE` constraint, the system drops the constraint.
- The primary index, primary AMP index, or partitioning expression of a table or join index cannot be defined on columns that have any of the following data types.
 - BLOB
 - CLOB
 - JSON
 - XML
 - Period
 - Derived Period
 - ARRAY/VARRAY
 - Geospatial
 - Distinct or structured UDTs that contain BLOB, CLOB, or XML columns.

All other Teradata SQL data types are valid.

- If there is a FOREIGN KEY reference to or FOREIGN KEY constraint on either the current or resulting set of primary index columns, then you can only change the name of the index. Otherwise, the request aborts and the system returns an error to the requestor.
- If you specify a column set for the primary index or primary AMP index of a populated table or join index, the column set must be identical to the existing column set for the primary index or primary AMP index.

To specify a column set for the primary index or primary AMP index that differs from the current column set for the primary index or primary AMP index, the table must be empty.

- You cannot alter a populated table with primary index that currently has a NUPI to have a UPI on the same column set unless a USI is also defined on that column set.
- If you specify a column set for the primary index of a populated table or join index that differs from a current primary index that is defined implicitly by a PRIMARY KEY or UNIQUE constraint, the system drops that PRIMARY KEY or UNIQUE constraint.
- If the current primary index for a table or join index is defined implicitly by a PRIMARY KEY or UNIQUE constraint and you use ALTER TABLE to specify a PARTITION BY clause, the system drops that PRIMARY KEY or UNIQUE constraint.
- You cannot modify the primary index of a queue table to partition it.

Queue tables cannot have partitioned primary indexes or primary AMP indexes.

Changing the Name of a Primary Index or Primary AMP Index

The following rules apply to changing the name of a primary index or primary AMP index:

- The *index_name* cannot be the same as the name of any secondary index or constraint defined for the table or join index.
- If you specify *index_name*, that is the name of the index.
- If you specify NOT NAMED, the index has no name.
- If you specify neither an *index_name* nor a NOT NAMED phrase, then no change is made to the current name.

MODIFY PRIMARY Option

The following rules apply to the MODIFY PRIMARY INDEX or MODIFY PRIMARY AMP INDEX option:

- If you specify MODIFY PRIMARY and the table currently does not have a primary index, you must specify a column list to indicate the columns to be included in the new primary index.

If you do not, the system returns an error to the requestor.

- You cannot alter a populated table with a primary-index or primary AMP index to have a different set of index columns. Otherwise, the system returns an error to the requestor.
- If you specify UNIQUE PRIMARY INDEX, then the new primary index for the table is a UPI.

A primary index can be unique only if all the partitioning columns, if any, are also included in the column set that defines the primary index.

You cannot specify `UNIQUE PRIMARY INDEX` unless at least one of the following conditions is true for the current primary index for the table.

- The current primary index is a UPI.
- The primary index is defined implicitly by a `PRIMARY KEY` or `UNIQUE` constraint.
- An existing USI is defined on the same column set as the new UPI.

The existing USI can be specified either explicitly or implicitly by a `PRIMARY KEY` or `UNIQUE` constraint.

- The table is not populated with rows.

Otherwise, the system returns an error to the requestor.

- If you specify `NOT UNIQUE PRIMARY INDEX`, the new primary index for the table or join index is a NUPI.
- If you neither specify `UNIQUE PRIMARY INDEX` nor `NOT UNIQUE PRIMARY INDEX`, then Vantage does not change the uniqueness of the primary index for the table or join index.
- You cannot specify `ALL` with `MODIFY PRIMARY`.

MODIFY NO PRIMARY Option

The following rules apply to the `MODIFY NO PRIMARY` option.

- You can specify `MODIFY NO PRIMARY` without altering partitioning and the table currently has a column-partitioned primary index or primary AMP index.
- If you specify `MODIFY NO PRIMARY` for a SET table, Vantage also alters the table to be `MULTISET`.
- If you specify `MODIFY NO PRIMARY` with `DROP RANGE#n`, `ADD RANGE#n` clauses, or both, and the table currently has a partitioned primary index without column partitioning, the system returns an error to the requestor.
- You can specify an `ADD` clause for column partitioning, a `RANGE_N` partitioning expression, or a `CASE_N` partitioning expression.

If a row partitioning is not based on a `RANGE_N` or `CASE_N` partitioning expression, you cannot specify an `ADD` clause. Otherwise, the system returns an error to the requestor.

Rules for Modifying Populated and Unpopulated Tables and Join Indexes

The following rules apply to altering the partitioning of a populated table or join index.

- You cannot alter a populated partitioned table to have no partitioning.
- You cannot alter a populated nonpartitioned table to have partitioning.
- You cannot alter the partitioning of a populated partitioned table to have a new partitioning using a `PARTITION BY` clause.

- You cannot alter the partitioning of a populated partitioned table to have a new partitioning with DROP RANGE clauses, ADD RANGE clauses, or both unless the new partitioning expressions are defined using DROP RANGE#L *n* and ADD RANGE#L *n* clauses that do not drop ranges other than from the beginning or end of the range sequence with at least one remaining range, and then does not add ranges between the resulting beginning and end of the ranges.

If the table has multilevel partitioning and the modification of a partitioning expression includes dropping the NO RANGE or UNKNOWN partitions, the partitioning expression must not have been modified previously.

The resulting number of partitions for a level must be between 1 and the maximum defined for that level.

Modifying the Partitioning of a Table or Join Index

The following rules and restrictions apply to modifying partitioning.

- You cannot alter a table with 2-byte partitioning to have 8-byte partitioning.
- You cannot alter a table with 8-byte partitioning to have 2-byte partitioning even if the number of combined partitions decreases to be fewer than 65,536.
- If you specify neither PARTITIONED BY nor NOT PARTITIONED, Vantage does not change the partitioning of the specified table.
- You cannot partition a secondary or hash index.
- If a table does not need to be unpopulated for a particular ALTER TABLE statement, the modified table and new partitioning must meet the conditions defined in [MODIFY PRIMARY Option](#) and [MODIFY NO PRIMARY Option](#) except that 8-byte partitioning remains 8-byte partitioning even if the new number of combined partitions decreases to be fewer than 65,536 and a 2-byte partitioning remains 2-byte partitioning. See [Rules for Modifying Populated and Unpopulated Tables and Join Indexes](#).

If the new partitioning defines more than a maximum of 65,535 combined partitions for a 2-byte partitioned table, the table must be unpopulated, and the rule documented in the previous paragraph does not apply. In this case, the new partitioning for an unpopulated table would be 8-byte partitioning.

- If you specify a secondary index in an ALTER TABLE request that is followed by a COMMA character, which is then followed by a PARTITION BY clause, the PARTITION BY clause applies to the table, not to the secondary index.
- If you specify a PARTITION BY clause by itself in an index list, not as the last item specified in the index list, you must also add a COMMA character following the PARTITION BY clause.
- You cannot specify a PARTITION BY clause more than once by itself in an index list.
- If you specify a PARTITION BY clause by itself in an index list, you cannot also specify it in an index definition for NO PRIMARY, PRIMARY, or PRIMARY AMP.
- You cannot specify a COLUMN partitioning level more than once in a PARTITION BY clause.
- Vantage implicitly rewrites a partitioning expression as follows.
 - A TIME(*n*) without time zone literal is modified to a TIME(*n*) WITH TIME ZONE literal by including the current session time zone displacement, if one exists, or the time zone displacement based

on the current session time zone string, the local value of the literal, and the `CURRENT_DATE` at UTC (that is, at time zone `+00:00`).

- A `TIMESTAMP(n)` without a time zone literal is modified to be a `TIMESTAMP(n) WITH TIME ZONE` literal by including the current session time zone displacement, if one, or the time zone displacement based on the current session time zone string and the local value of the literal.
- An `AT LOCAL` clause is modified to be an `AT` simple expression clause that specifies a character literal.

If the current session time zone is a time zone displacement, the character literal is the current session time zone displacement as a character literal with format `'+hh:mm'` if positive and `'-hh:mm'` if negative; if the current session time zone is a time zone string, the character literal is the current session time zone string.

- For `CURRENT_DATE` or `DATE` without an `AT` clause, the resolved current date defined is at the current session time zone.

For `CURRENT_DATE` or `DATE` with an `AT [TIME ZONE]` simple expression clause, the resolved current date is at the specified time zone.

- If evaluation of the new partitioning expression for a row or column in the table causes evaluation errors (such as divide by zero), changes to the table (and also to the save table, if any) are rolled back and the partitioning expression is not changed.
- A new partitioning expression for a level becomes the partitioning expression for that level after it is successfully altered.

If there is a subsequent attempt to insert or update a row or column of a row-partitioned table such that the partitioning expression for that row or column does not result in a value between 1 and the number of partitions defined for that level, an error occurs for the insert or update.

- If a table or join index has both column partitioning and row partitioning, modifying the row partitioning for a populated table or join index has a performance impact. The performance impact is because it can be more costly to move a table row from one row partition to another.
- You must disable the triggers specified in the following table before you can perform an `ALTER TABLE` request that modifies or revalidates a table that is defined with triggers.

| ALTER TABLE Request Modifies or Revalidates the Row Partitioning and Includes this Option | Disable These Triggers |
|--|---|
| WITH DELETE | All delete triggers on <i>table_name</i> . After the <code>ALTER TABLE</code> request completes, you can re-enable the disabled triggers. |
| WITH INSERT | <ul style="list-style-type: none"> ◦ All delete triggers on <i>table_name</i>. ◦ All insert triggers on <i>save_table</i>. After the <code>ALTER TABLE</code> request completes, you can re-enable the disabled triggers. |

The following general rules apply to the MODIFY PRIMARY, MODIFY NO PRIMARY, and PARTITIONED BY clauses.

- If you do not specify MODIFY PRIMARY or MODIFY NO PRIMARY, but do specify a PARTITION BY clause, the default is NO PRIMARY INDEX even if you specify a PRIMARY KEY or UNIQUE constraint, regardless of the setting of the DBS Control field PrimaryIndexDefault. For information about DBS Control and the PrimaryIndexDefault field, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- If you do not specify MODIFY PRIMARY, MODIFY NO PRIMARY, or a PARTITION BY clause, the default primary index, primary AMP index, or default MODIFY NO PRIMARY for the table or join index is determined by the setting of the DBS Control field PrimaryIndexDefault. For information about the DBS Control PrimaryIndexDefault field, see *Teradata Vantage™ - Database Utilities*, B035-1102.

The following rules apply to the system-derived columns PARTITION and PARTITION#L *n*.

- You cannot specify the system-derived columns PARTITION or PARTITION#L *n*, where the value of *n* ranges from 1-15 inclusive for 2-byte partitioning and from 1 - 62 inclusive for 8-byte partitioning, in any of its forms in the definition of a new partitioning expression.
- Any change to how a table or join index is partitioned affects the values for the system-derived columns PARTITION and PARTITION#L *n* for most, if not all, rows in the table or join index.

The following rules apply to the NOT PARTITIONED clause.

- If you specify NOT PARTITIONED and the existing table or join index is partitioned, its partitioning is modified to be a nonpartitioned if the table or join index is not populated with rows.
If the table or join index is populated, the system returns an error to the requestor.
- If you specify NOT PARTITIONED and the existing table or join index has a nonpartitioned primary index, Vantage does not change the partitioning of its primary index.
- If you specify neither PARTITIONED BY nor NOT PARTITIONED, the partitioning of the primary index is not changed.

WITH INSERT and WITH DELETE Null Partition Handling Clauses

The following rules apply to the use of the WITH INSERT INTO *save_table* and WITH DELETE null partition handling clauses when modifying or reevaluating the row partitioning of a table or join index.

If You Do Not Specify a Null Partition Handling Clause

If you do not specify a WITH INSERT INTO *save_table* or WITH DELETE null partition handling clause, a new partitioning expression for all rows currently in the table must generate a value between 1 the number of partitions defined for that level, after the partitioning expression value is cast to INTEGER or BIGINT, if it is not already typed as INTEGER, BIGINT, or CHARACTER.

Otherwise, the system returns an error to the requester.

If an Existing Row Violates a New Partitioning Expression

If an existing row violates a new partitioning expression and you have not specified a WITH DELETE or WITH INSERT INTO *save_table* null partition handler clause, then you cannot apply the new partitioning expression.

Rows that violate the implied partitioning CHECK constraint, including those whose partitioning expression evaluates to null, are not allowed in the table.

WITH DELETE Null Partition Handling Clause

If you specify a WITH DELETE null partition handling clause, any row for which a new partitioning expression does not generate, after casting to INTEGER or BIGINT if it is not already typed as INTEGER, BIGINT, or CHARACTER, a value between 1 and the number of partitions defined for that level is deleted.

Otherwise, the system returns an error to the requestor.

WITH INSERT INTO *save_table* Null Partition Handling Clause

If you specify a WITH INSERT INTO *save_table* null partition handling clause, Vantage deletes any row for which a new partitioning expression does not result in a value between 1 and the number of partitions defined for that level and inserts it into *save_table*.

The following rules apply to this clause.

- *save_table* cannot be the same table as *table_name* or the system returns an error to the requestor.
- *save_table* must have the same number of columns with compatible data types as *table_name*, or the system returns an error to the requestor.

Rollback in Case of Errors

If evaluating the new partitioning expression for a row in the table causes evaluation errors to occur, Vantage rolls back any changes made to the table and to *save_table* and does not change the partitioning.

If errors occur when inserting into *save_table*, the following rollback behavior occurs, depending on the session mode.

| Session Mode | Block of Work Rolled Back |
|--------------|---------------------------|
| ANSI | Request. |
| Teradata | Transaction. |

New Partitioning

A new partitioning for a level does not become the partitioning for that level until it is successfully altered.

If you later attempt to insert or update a row of a row-partitioned table such that a partitioning expression for that row does not result in a value between 1 and the number of partitions defined for that level, the system returns an error to the requestor.

Changing Row Partitioning on a Populated Table

There is a negative performance impact if the table or join index has both column partitioning and row partitioning if the row partitioning is changed on a populated table because it is often costly to move a table row from one row partition to another.

Disable Triggers Before Modifying or Revalidating Table

You must disable the triggers specified in the following table before you can perform an ALTER TABLE request that modifies or revalidates the table.

| ALTER TABLE Clause to Modify or Revalidate Row Partitioning | Triggers to Disable |
|---|--|
| WITH DELETE | All delete triggers on <i>table_name</i> . After the ALTER TABLE request completes, re-enable the disabled triggers. |
| WITH INSERT INTO <i>save_table</i> | <ul style="list-style-type: none"> • All delete triggers on <i>table_name</i>. • All insert triggers on <i>save_table</i>. After the ALTER TABLE request completes, re-enable the disabled triggers. |

RANGE_N and CASE_N Functions

General Guidelines for RANGE_N and CASE_N Functions

The following rules apply when using the RANGE_N and CASE_N functions to create a partitioning expression.

- For a table or join index with 2-byte partitioning not already at its maximum combined partition number, the maximum number of partitions for the first level is increased to the largest value that does not cause the maximum combined partition number to exceed 65,535.

If there is at least one level with an explicit ADD clause, there is at least one level that consists solely of a RANGE_N function with BIGINT data type, or there is column-partitioning, this is repeated for each of the other levels, from the second level to the last.

- The partitioning expression is rewritten to be CAST to INTEGER data type if it does not solely consist of a RANGE_N function and it does not already have INTEGER or BIGINT data type. If it cannot be CAST to INTEGER or BIGINT data type, the system returns an error to the requestor.
- The number of defined partitions for a row partitioning level is the number of row partitions specified by the RANGE_N or CASE_N function used to define the row partitioning for the level or 65,535 if the level is not based on a RANGE_N or CASE_N function.

RANGE_N Functions

The following conditions are true if a new partitioning expression consists solely of a RANGE_N function.

- The number of ranges defined must be less than or equal to 65,535 for 2-byte partitioning or less than or equal to 9,223,372,036,854,775,807 for 8-byte partitioning. Otherwise, the system returns an error to the requestor.

For single-level partitioning, the total number of partitions defined for such a partitioning expression can be up to 65,535 for 2-byte partitioning and up to 9,223,372,036,854,775,807 for 8-byte partitioning if both the NO RANGE and UNKNOWN partitions are specified.

For a RANGE_N function with INTEGER data type, the number of partitions defined must be less than or equal to 65,535. Otherwise, the system returns an error to the requestor.

For a RANGE_N function with BIGINT data type, the number of partitions defined must be less than or equal to 9,223,372,036,854,775,807. Otherwise, the system returns an error to the requestor.

- A range is combined with the previous range if the following conditions are all true.
 - The previous range is not specified with a range start of *.
 - The previous range is a multiple of its EACH range size if specified. That is, the last expanded range is not smaller than specified in the EACH clause.
 - The end of the previous range is just prior the start of this range. A timestamp range end value specifying any of 59.999999 truncated to the precision of the test value through 61.999999 seconds truncated to the precision of the test value, inclusive, is considered to be just prior to the start of the next minute.
 - The size (ignoring leap seconds) of the range (or its EACH range size if specified) is the same for both ranges.

The combined range has an EACH clause which is the same as the EACH clause of the prior range or, if it does not have an EACH clause, the combined range has an EACH clause with a range size equal to the size of the prior range.

- For a partitioning expression for single-level partitioning defined using only a RANGE_N function with INTEGER data type, the total number of partitions defined must be less than or equal to 2,147,483,647.

If the number of partitions exceeds 2,147,483,647, the system returns an error to the requestor.

For 2-byte single-level partitioning, the RANGE_N function cannot define more than 65,533 range partitions. Otherwise, the partitioning would be 8-byte partitioning.

For 2-byte single-level partitioning, the total number of partitions defined can be as many as 65,535 if you also specify both the NO RANGE and UNKNOWN partitions.

- If a partitioning expression is defined using only a RANGE_N function with BIGINT data type, the number of ranges defined must be less than or equal to 9,223,372,036,854,775,805.

If the number of ranges exceeds 9,223,372,036,854,775,805, the system returns an error to the requestor.

For single-level partitioning, the total number of partitions defined for such a partitioning expression can be as many as 9,223,372,036,854,775,807 if you also specify both the NO RANGE and UNKNOWN partitions.

CASE_N Functions

The following conditions are true if a partitioning expression consists only of a CASE_N function.

- The number of partitions defined must be less than or equal to 2,147,483,647. Otherwise, the system returns an error to the requestor because the CASE_N function has INTEGER data type and 2,147,483,647 is the largest possible INTEGER value.
- The number of partitions defined must be less than or equal to 2,147,483,647 because values returned by the CASE_N function have INTEGER data type.

If the number of partitions exceeds 2,147,483,647, the system returns an error to the requestor.

Modifying Partitioning Using the ADD RANGE and DROP RANGE Options

General Rules and Guidelines

The following rules and guidelines apply to modifying table partitioning by adding ranges to and dropping ranges from partitioning expressions using the ADD RANGE and DROP RANGE options. For additional rules that apply to character partitioning expressions, see [Rules for Modifying Populated and Unpopulated Tables and Join Indexes](#).

Unless otherwise indicated, these rules apply to both row-partitioned tables and join indexes and to the row partitioning levels of column-partitioned tables and join indexes.

These rules only allow modification of RANGE_N partitioning expressions that are already defined for the table. You cannot use these options to change the number of partitioning levels for a table, nor can you use them to drop or add partitioning levels.

You cannot use an ALTER TABLE ... ADD RANGE[#L n] or an ALTER TABLE ... DROP RANGE[#L n] request to alter the partitioning of a table from partitioned to nonpartitioned.

Rows can be moved from their current partition to any of the following other partitions.

- Newly added range partitions.
- The NO RANGE partition.
- The NO RANGE OR UNKNOWN partition.

Similarly, rows can be moved from the NO RANGE partition to newly added range partitions.

To delete the rows of a partition for the situation where they have other partitions they could be moved into (such as NO RANGE, NO RANGE OR UNKNOWN, or a newly added partition), use a DELETE request to remove them from the table before you alter its partitioning expressions.

There are two general guidelines for adding and dropping ranges:

- Do not overlap new ranges with dropped ranges. For example, do not DROP 2 ranges and ADD one range that covers both of the original 2 ranges, or DROP up to all but 1 range and ADD back new ranges.
- DROP RANGE ... WITH DELETE and DROP RANGE ... WITH INSERT INTO *save_table* do not necessarily delete rows in the dropped ranges.

Changing Partitions Using the PARTITION BY Clause

To change the number of partitioning levels or to modify a partitioning level that is not based only on a RANGE_N function, the table cannot be populated with rows and you must use the PARTITION BY clause to specify the desired partitioning levels and expressions. See [Rules for Modifying Populated and Unpopulated Tables and Join Indexes](#).

To change the number of partitioning levels or to modify a partitioning expression that is neither based only on a RANGE_N function nor based only on a CASE_N function, you must use the PARTITION BY clause in an ALTER TABLE request to specify all the desired partitioning levels and expressions and the table must be empty. See [Rules for Modifying Populated and Unpopulated Tables and Join Indexes](#).

This type of partitioning expression is only valid for single-level partitioning. All multilevel partitioning expressions must be based only on a RANGE_N function or only on a CASE_N function. The two functions can be mixed within a partitioning expression, but you can only specify each of them once per partitioning level of the partitioning expression.

If that is not possible, or if the reorganization of partitions would cause too many rows to be moved from their current partition to a different partition, you can use a CREATE TABLE request to create a new table with the partitioning you want, then use a MERGE or INSERT ... SELECT request with error logging to move the rows from the source table into the newly created target table with the desired partitioning.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for information about the INSERT ... SELECT and MERGE statements and their LOGGING ERRORS option, and [CREATE ERROR TABLE](#) for information about error tables.

Note the following rules exist for modifying partitioning:

- You cannot specify NO PRIMARY INDEX without altering partitioning for a table without a primary index.
- You cannot specify NO PRIMARY INDEX with NOT PARTITIONED for a table without a primary index, primary AMP index, or partitioning.
- You cannot specify NO PRIMARY INDEX without altering partitioning for a table that has a partitioned primary index without column partitioning.
- You cannot specify NO PRIMARY INDEX with a DROP RANGE#n clause, an ADD RANGE#n clause, or both, for a table that has a partitioned primary index without column partitioning.
- You cannot alter the partitioning of a populated partitioned table to have a new partitioning with DROP clauses, ADD clauses, or both unless the new partitioning expressions are defined using DROP RANGE[#L n] and ADD RANGE[#L n] clauses that do not drop ranges other than from the beginning or end of the range sequence with at least one remaining range, and then does not add ranges between the resulting beginning and end of the ranges.

If the table has multilevel partitioning and the modification of a partitioning expression includes dropping the NO RANGE [OR UNKNOWN] or UNKNOWN partitions, the partitioning expression must not have been modified previously.

The resulting number of partitions for a level must be between 1 and the maximum defined for that level.

- You cannot ADD or DROP partitioning expressions that are based on the CASE_N function.

To modify a partitioning expression that is based on the CASE_N function, you must use the MODIFY [PRIMARY INDEX] option to redefine the entire PARTITION BY clause (see [Redefining the Primary, Primary AMP, or Partitioning for a Table](#), [General Rules for the MODIFY PRIMARY Clause](#)), [Modifying the Partitioning of a Table or Join Index](#), and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

- You can alter the row partitioning of a table or join index by explicitly specifying new partitioning expressions only if the table is empty (see [Rules for Modifying Populated and Unpopulated Tables and Join Indexes](#)).

Using DROP RANGE

The following table lists conditions that are true for partitioning when you specify DROP RANGE:

| If... | Then... | Otherwise, |
|---|---|--|
| you specify DROP RANGE without a partitioning level number in the first alter partitioning expression, | it is equivalent to DROP RANGE#L <i>n</i> where <i>n</i> is equal to the number of the first partitioning level that is defined using only a RANGE_N partitioning expression. | if there is no such level, the system returns an error to the requestor. |
| you specify DROP RANGE without specifying a level number in other than the first alter partitioning expression, | DROP RANGE is equivalent to DROP RANGE#L <i>n</i> , where <i>n</i> is equal to the number of the next partitioning level that is specified using only a RANGE_N partitioning expression after the partitioning level for the preceding alter partitioning expression. | if there is no such level, the system returns an error to the requestor. |
| you specify DROP RANGE[#L <i>n</i>] in other than the first alter partitioning expression, | the value for <i>n</i> must be at least 1 greater than the default or specified level associated with the preceding alter partitioning expression. | the system returns an error to the requestor. |
| you specify the default or specified level for is DROP RANGE#L <i>n</i> , | the value for <i>n</i> must not exceed the number of partitioning levels currently defined for the table. | the system returns an error to the requestor. |
| you drop a range or partition, | it does not necessarily cause rows to be deleted. | affected rows could be moved to an added range or to the optional NO RANGE [OR UNKNOWN] or UNKNOWN partitions. |
| you specify a DROP RANGE WHERE <i>partition_conditional_expression</i> for both multilevel row-partitioned tables and multilevel column-partitioned tables, | the specification must reference the system-derived column PARTITION#L <i>n</i> , where the value for <i>n</i> ranges from 1 - 62, inclusive, and is the same as the default or specified level of the corresponding DROP RANGE[#L <i>n</i>] and <i>no other columns</i> . | the system returns an error to the requestor. |
| the table definition specifies an explicit column named | you cannot specify PARTITION#L 1 in the DROP RANGE WHERE <i>partition_</i> | the system returns an error to the requestor. |

| If... | Then... | Otherwise, |
|---|---|---|
| PARTITION#L1 for both multilevel row-partitioned tables and multilevel column-partitioned tables, | <i>conditional_expression</i> specification of a DROP RANGE[#L <i>n</i>] clause. | |
| you specify a WITH DELETE clause for a multilevel partitioning expression, | there must be a DROP RANGE[#L <i>n</i>] in at least one of the alter partitioning expressions. | the system returns an error to the requestor. |
| a single-level row-partitioned table definition specifies an explicit column named PARTITION, | you cannot specify PARTITION in the WHERE <i>partition_conditional_expression</i> specification of a DROP RANGE WHERE clause. | the system returns an error to the requestor. |
| you specify a WITH INSERT clause for a multilevel partitioning expression, | there must be a DROP RANGE[#L <i>n</i>] in at least one of the alter partitioning expressions. | the system returns an error to the requestor. |
| working in a single-level row-partitioned table, | a DROP RANGE WHERE clause conditional expression must reference either the system-derived column PARTITION or the system-derived column PARTITION#L1, but not both, and no other columns. | the system returns an error to the requestor. |

Using ADD RANGE

The following table lists conditions that are true for partitioning when you specify ADD RANGE:

| If... | Then... | Otherwise, |
|--|--|--|
| you specify an ADD RANGE expression without also specifying a level in the same ALTER TABLE request as a DROP RANGE[#L <i>n</i>] expression in an alter partitioning expression, | the operation is equivalent to an ADD RANGE#L <i>n</i> expression, where the value for <i>n</i> is the same as the default or specified level number of that DROP RANGE[#L <i>n</i>]. | |
| you specify an ADD RANGE expression without a level and without a DROP RANGE[#L <i>n</i>] expression in the first alter partitioning expression for a multilevel partitioned table, | that ADD RANGE expression is equivalent to ADD RANGE#L <i>n</i> , where the value for <i>n</i> is equal to the number of the first partitioning level that is defined only with a RANGE_N partitioning expression. | if there is no such level, the system returns an error to the requestor. |
| you specify an ADD RANGE expression without specifying a level and without also specifying a DROP RANGE[#L <i>n</i>] expression in other than the first alter partitioning expression for a multilevel partitioned table, | that ADD RANGE expression is equivalent to ADD RANGE#L <i>n</i> , where the value of <i>n</i> is equal to the level number of the next partitioning level that is defined using only a RANGE_N partitioning expression after the | if there is no such level, the system returns an error to the requestor. |

| If... | Then... | Otherwise, |
|--|---|--|
| | partitioning level for the preceding alter partitioning expression. | |
| you specify an ADD RANGE expression with a DROP RANGE[#L <i>n</i>] in an alter partitioning expression, | it is equivalent to ADD RANGE[#L <i>n</i>], where <i>n</i> is the same as the default or specified level of that DROP RANGE[#L <i>n</i>]. | |
| you specify an ADD RANGE[#L <i>n</i>] clause with a DROP RANGE[#L <i>n</i>] clause in an alter partitioning expression, | the value for <i>n</i> must specify the same level as the default or specified level of that DROP RANGE[#L <i>n</i>] expression. | the system returns an error to the requestor. |
| you specify an ADD RANGE clause without a DROP RANGE[#L <i>n</i>] in the first alter partitioning expression of a multilevel partitioning expression, | it is equivalent to ADD RANGE[#L <i>n</i>], where <i>n</i> is equal to the number of the first partitioning level that is defined solely with a RANGE_N partitioning expression. | the expression is equivalent to ADD RANGE[#L <i>n</i>], where the value of <i>n</i> is one greater than the level associated with the preceding alter partitioning expression. If there is no such level, the system returns an error to the requestor. |
| you specify an ADD RANGE[#L <i>n</i>] expression without also specifying a DROP RANGE[#L <i>n</i>] expression in other than the first alter partitioning expression of a multilevel partitioning expression, | the value for <i>n</i> must be at least one greater than the default or specified level associated with the preceding alter partitioning expression. | the system returns an error to the requestor. |
| you specify an ADD RANGE[#L <i>n</i>] expression for a multilevel partitioning expression, | the value for <i>n</i> must not exceed the number of partitioning levels currently defined for the table. | the system returns an error to the requestor. |

Using DROP RANGE or ADD RANGE

The following conditions are true if you specify DROP RANGE or ADD RANGE:

- You cannot use an ALTER TABLE ... ADD RANGE[#L *n*] or an ALTER TABLE ... DROP RANGE[#L *n*] request to alter the partitioning of a table from partitioned to nonpartitioned.
- DROP RANGE and ADD RANGE modify an existing partitioning expression to create a new partitioning expression for the table.

Vantage deletes rows only if they do not satisfy this new partitioning expression.

If the new partitioning expression specifies the NO RANGE option, then no rows are deleted.

- If you specify DROP RANGE#L *n* or ADD RANGE#L *n*, the partitioning at level *n* must be a partitioning expression that consists only of a RANGE_N function that does not compare character or graphic data. Otherwise, the system returns an error to the requestor.

- If you specify a DROP RANGE#L *n* or ADD RANGE#L *n* expression for a multilevel partitioning expression, the partitioning at level *n* must be a partitioning expression that is specified using only a RANGE_N function that does not compare character or graphic data. Otherwise, the system returns an error to the requestor.
- If you specify a DROP RANGE or ADD RANGE expression for a partitioned table without also specifying a level number, the partitioning at the default level for the options must not compare character or graphic data. Otherwise, the system returns an error to the requestor.
- If you specify a DROP RANGE#L *n* or ADD RANGE#L *n* expression without a level number for a multilevel partitioning expression, the partitioning at the default level for the options must not compare character or graphic data. Otherwise, the system returns an error to the requestor.
- If you specify DROP RANGE[#L *n*] or ADD RANGE[#L *n*] for a multilevel partitioning expression, the corresponding partitioning expression at the default or specified level of this DROP RANGE[#L *n*] or ADD RANGE[#L *n*] must be defined only using a RANGE_N function that does not compare character or graphic. You cannot define the expression with a BLOB, CLOB, or BIGINT data type. Otherwise, the system returns an error to the requestor.

Using Range Expressions

The following rules are true for using Range Expressions.

- Each range specified for a start_expression or end_expression must conform to the rules of a RANGE_N function (see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for details).

You must obey the following rules.

- A range value must be compatible in data type with the test expression that defines the corresponding existing partitioning expression for the table.
- The ranges must be in sequential order and cannot overlap one another.
- The last range of a range expression sequence must specify an ending range and be preceded by the keyword AND.
- Not all data types are not supported for the test expression or in the start and end expressions. See *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for details.
- You can specify either an OR UNKNOWN clause or an UNKNOWN clause for a range expression sequence, but not both.

If you specify both, the system returns an error to the requestor.

- Dropping a range or partition does not necessarily cause rows to be deleted. Affected rows could be moved to an added range or to the optional NO RANGE [OR UNKNOWN] or [UNKNOWN] or UNKNOWN partitions.
- The resulting new partitioning expression for a level must define at least one range. Otherwise, the system returns an error to the requestor.

For multilevel partitioning, the resulting new partitioning expressions must each define at least two partitions. Otherwise, the system returns an error to the requestor.

| If a new partitioning expression corresponds to this level ... | Then ... |
|--|--|
| 1 | the number of partitions defined must be less than or equal to the previously defined maximum. |
| 2 or higher | it must define the <i>same</i> number of partitions. |

Using ALTER Table Request

An ALTER TABLE request that specifies a DROP RANGE[#L *n*] expression, an ADD RANGE[#L *n*] expression, or both, is equivalent to the following generalized ALTER TABLE request syntax expressed in pseudo-BNF format.

```
ALTER TABLE table_name
  MODIFY [ [ [NOT] UNIQUE ] PRIMARY INDEX [ index_name | NOT NAMED ]
  [ (primary_index_column_list) | NO PRIMARY INDEX ]
  PARTITION BY {
    new_partitioning_level |
    (new_partitioning_level [, new_partitioning_level] ...)
  }
  [ null_partition_handler ];
```

table_name

index_name

primary_index_column_list

new_partitioning_level

Same as in the original ALTER TABLE request.

null_partition_handler

Refers to a WITH INSERT or WITH DELETE clause in the original ALTER TABLE request.

The number of new partitioning levels is the same as the number of existing partitioning levels for the table.

Each new partitioning level is the same as the existing partitioning for that level except as noted in the following points:

| If you... | Then... | Otherwise, |
|--|---|------------|
| specify DROP RANGE[#L <i>n</i>] WHERE <i>partition_conditional_expression</i> for that level, | the new partitioning expression for that level does not include the specified existing ranges or partitions where the partition conditional expression evaluates to TRUE. | |

| If you... | Then... | Otherwise, |
|---|--|------------|
| specify DROP RANGE[#L n] <i>alter_ranges</i> for that level, | the new partitioning expression for that level does not include the specified ranges and partitions in the alter ranges. | |
| specify ADD RANGE[#L n] <i>alter_ranges</i> for that level, | the new partitioning expression for that level includes the addition of the new ranges and partitions merged in order with the existing ranges after first applying the DROP RANGE[#L n] clause if one is specified. | |
| reduce the number of defined partitions for a level to 0 after dropping and adding any partitions, | the system returns an error to the requestor. | |
| reduce the number of defined partitions for a level by <i>r</i> partitions after dropping partitions, adding partitions, or both, and the level previously specified an ADD option, | Vantage increases the value for ADD by <i>r</i> . | |
| reduce the number of defined partitions for a level by <i>r</i> partitions after dropping partitions, adding partitions, or both, and the level did not previously specify an ADD option, | Vantage adds an ADD <i>r</i> specification for the level. | |
| increase the number of defined partitions for a level by <i>i</i> partitions after dropping partitions, adding partitions, or both, and the level previously specified an ADD option with a value greater than <i>i</i> , | Vantage decreases the specified value for ADD by <i>i</i> . | |
| increase the number of defined partitions for a level by <i>i</i> partitions after dropping partitions, adding partitions, or both, and the level previously specified an ADD option with a value equal to <i>i</i> , | Vantage removes the ADD option for that level. | |
| increase the number of defined partitions for a level by <i>i</i> partitions after dropping partitions, adding partitions, or both, and the level previously specified an ADD option with a value less than <i>i</i> either explicitly or by default, | the system returns an error to the requestor. | |

| If you... | Then... | Otherwise, |
|---|---|---|
| increase the number of defined partitions for a level by <i>i</i> partitions after dropping partitions, adding partitions, or both, and the level did not previously specify an ADD option either explicitly or by default, but one or more partitioning levels do specify an ADD option either explicitly or by default, | the system returns an error to the requestor. | |
| increase the number of defined partitions in a multilevel partitioning expression for a level for other than level 1 by <i>i</i> partitions after dropping partitions, adding partitions, or both, none of the levels specifies an ADD option, and the table is populated with rows, | the system returns an error to the requestor. Note: You can only perform this operation on an unpopulated table. | |
| use a new partitioning expression, | the new partitioning expression for a level must specify at least one range. | the system returns an error to the requestor. This error occurs under the following circumstances: <ul style="list-style-type: none"> • When only one partition is defined for a row partitioning level with an ADD 0 or with no ADD specification AND • The maximum number of partitions is not increased to the largest value that would not increase the partitioning from 2-byte partitioning to 8-byte partitioning OR • If the table or join index already has 8-byte partitioning, the maximum number of partitions is not increased to the largest value that does not cause the maximum combined partition number to exceed 9,223,372, 036,854, 775,807 AND |

| If you... | Then... | Otherwise, |
|-----------|---------|---|
| | | <ul style="list-style-type: none"> The maximum is not increased to at least 2. |

Best Practices for Adding and Dropping Partitioning Ranges from a Partitioning Expression

The following guidelines are recommended best practices for maintaining the partitioning ranges of a partitioned table.

As with any processes, you must tune the best practices listed below to suit your individual processes, query workloads, and performance criteria.

- Have a well-defined, preferably automated, process for scheduling and issuing ALTER TABLE requests to add and drop partitioning ranges.
- Collect and maintain a current, monthly PARTITION statistics report which tracks the frequency of ALTER TABLE requests for adding and dropping partitions.
- Run and compare periodic EXPLAIN reports. This report is useful for determining the effects of your current partitioning on the performance of individual queries from your standard workloads.
- Based on your collected PARTITION statistics and scheduling process, define enough *future ranges* to minimize the frequency of ALTER TABLE requests for adding and dropping partitions. You should keep the number of “future” ranges at any one time to less than 10 percent of the total number of defined partitions.

Note:

A *future range* is a range over date set of values which has not yet occurred at the time the partition is added. It is used to minimize the frequency of ALTER TABLE requests for adding and dropping partitions.

- To ensure the ALTER TABLE request does not fail as a result of infrequency, at least monthly make the necessary range drops and additions.
- Ensure that you drop any old partitions no longer needed, especially if queries in the workloads accessing the row-partitioned table do frequent primary index accesses and joins, and the complete partitioning column set is not included in the primary index definition.

COLUMN Option for Column Partitioning

General Guidelines for Column Partitioning

The following rules apply to the COLUMN specification for column partitioning:

- You cannot specify a column more than one time within a column partition.

- You cannot specify a column to be in more than one column partition.
- A column partition value consists of the values of the columns in the column partition for a specific table row.
- When the COLUMN or ROW format is system-determined, either because you did not explicitly specify COLUMN or ROW format or because you explicitly specified SYSTEM format, the database bases its choice on the size of a column partition value for the column partition and other factors such as whether a column partition value for the column partition has fixed or variable length and whether the column partition is a single-column or multicolumn partition.

As a general rule, a narrow column partition is determined to have COLUMN format and a wide column partition is determined to have ROW format.

The database considers a column partition to be narrow when its size is about 256 bytes or less. Anything else is judged to be wide.

The database estimates the size of a variable-length column in a column partition as its (maximum length / 3) + 2. This value depends on other factors that might make it smaller or larger, and is also subject to change if a more appropriate value is determined.

This size is chosen based on the number of noncompressed column partition values that could fit into a container, and this depends on how large containers are allowed to grow.

To be narrow, a large enough number of column partition values can be packed into a container to get row header compression benefits that offset any negative impacts from packing values into a container and retrieving values from a container.

You can use HELP COLUMN requests or retrieve the information using an appropriate data dictionary view to determine the format that the database selects for a column partition.

- A column partition has COLUMN format, ROW format, or SYSTEM format, but never a mix of formats. Different column partitions of a column-partitioned table or join index can have different formats. The column partitions of a column-partitioned table can have all COLUMN format, all ROW format, all SYSTEM format, or a mixture of formats.

Using COLUMN Grouping in the PARTITION BY Clause

The following rules apply to specifying column grouping for a COLUMN specification in the PARTITION BY clause.

| If... | ...Then | Otherwise, |
|--|--|---|
| you specify COLUMN in the PARTITION BY clause for a table or join index, | the table or join index is column-partitioned by definition. | |
| you specify MODIFY NO PRIMARY and you specify a PARTITION BY clause, | you must specify a COLUMN partitioning level. | The database returns an error to the requestor. |

| If... | ...Then | Otherwise, |
|--|---|---|
| you specify non-group and group column partitions in the column group list, | the database defines a column partition for each non-group and group column partition. | |
| you do not specify a column grouping for a COLUMN clause, | the database defines a column partition for each column and column group specified in the column list for a CREATE TABLE request, or in the select list for a CREATE JOIN INDEX request. | |
| you specify ALL BUT with the column grouping specification, | the database defines a single-column partition with autocompression and a system-determined format of either COLUMN or ROW, depending on the column characteristics, for each column not specified in the column group list. | |
| you do not specify ALL BUT with the column grouping specification, | the database groups any columns not specified in the column group list into one column partition with autocompression and a system-determined format of either COLUMN or ROW, depending on the column characteristics. | |
| you specify COLUMN for a column partition, | the database stores one or more column partition values in a physical row called a container using COLUMN format. | |
| you specify ROW for a column partition, | the database stores only one column partition value in a physical row as a subrow. Subrow, or ROW, format is the standard way rows are stored. | |
| you specify SYSTEM for a column, | <p>the database determines the column partition format based on the size of a column partition value for the remaining set of columns in the column partition and other factors such as whether a column partition value for the column partition has fixed or variable length.</p> <p>Note: The database generally determines a narrow column partition to have COLUMN format and a wide column partition to have ROW format.</p> | |
| you want the database to determine the format for a column partition, | specify SYSTEM. | you can explicitly specify COLUMN or ROW format if you want a specific format for a column partition. |
| you explicitly specify AUTO COMPRESS or NO AUTO COMPRESS for a column partition, | the database either does or does not apply autocompression for physical rows according to the specification. | |

| If... | ...Then | Otherwise, |
|---|---|------------|
| | The database does apply any user-specified compression and, for column partitions with COLUMN format, row header compression. | |
| you want to determine the format that the database selects for a column partition, | use HELP COLUMN requests or retrieve the information using an appropriate data dictionary view. | |
| you specify neither COLUMN nor ROW for a column partition or if you specify SYSTEM, | the database determines whether to use COLUMN or ROW format depending on the column characteristics. | |

Rules for the MODIFY For Multilevel Partitioning

The general rules for MODIFY and MODIFY PRIMARY also apply to the multilevel case with the following exception. See [General Rules for the MODIFY PRIMARY Clause](#).

- If you specify PARTITION BY, the system accepts the request and alters the table to have a partitioning with the specified new partitioning expressions as long as one of the following is true.
 - The table is empty.
 - The new partitioning expressions are defined by DROP RANGE[#Ln] and ADD RANGE[#Ln] options that do not drop ranges other than from the beginning or end of the ranges, with at least one remaining range, and then does not add ranges between the resulting beginning and end of the ranges.

In addition, if the table has multilevel partitioning and the modification of a partitioning expression includes dropping the NO RANGE [OR UNKNOWN] or UNKNOWN partitions, that partitioning expression must not have previously been modified or has only been modified such that dropping the RANGE[OR UNKNOWN] or UNKNOWN partitions would not cause an internal partition number to be skipped in the sequence of internal partition numbers for the ranges.

- Vantage derives a new table-level partitioning CHECK constraint from a new set of partitioning expressions for a table or join index and replaces the previous partitioning CHECK constraint.
- When you alter the partitioning of a partitioned database object, Vantage derives a new partitioning CHECK constraint from the new set of partitioning expressions for each partitioned table and replaces the previous partitioning CHECK constraint. See [Partitioning CHECK Constraints for Partitioned Tables With 2-Byte Partitioning](#) and [Partitioning CHECK Constraint for Partitioned Tables with 8-Byte Partitioning](#) for more information about the partitioning CHECK constraints that Vantage derives for row-partitioned and column-partitioned tables and join indexes, respectively.

If an existing row violates the new partitioning expressions and you have not specified a WITH DELETE or WITH INSERT clause, then you cannot apply the new partitioning.

- If an existing row violates a new partitioning expression and you have not specified a WITH DELETE or WITH INSERT INTO *save_table* null partition handler clause, then you cannot apply the new partitioning expression.

Rows that violate the implied partitioning CHECK constraint, including those whose partitioning expression evaluates to null, are not allowed in the table.

Rules for the MODIFY Options for Character Partitioning

The rules for the MODIFY options for non-character partitioning in [General Rules for the MODIFY PRIMARY Clause](#) and [Rules for the MODIFY For Multilevel Partitioning](#) also apply to character partitioning and to the character row partitioning levels of a column-partitioned table or join index.

The following rules are specific to character row partitioning levels:

- You can specify a partitioning expression that involves character or graphic comparisons such as =, >, ≤, BETWEEN, or LIKE.
- A character row partitioning expression cannot specify Kanji1 or KanjiSJIS columns or constant expressions.
- A character row partitioning expression must be based on one or more columns from the table or deterministic expressions based on the columns from the table or join index the partitioning expression is written for.
- The expressions and referenced columns in the specification of the character partitioning expression must not have CLOB data types.
- The expressions specified in a character row partitioning expression must not contain any of the following functions.
 - Aggregate functions
 - Built-in functions
 - Grouped row OLAP functions
 - UDFs of any kind
 - HASHAMP function
 - HASHBAKAMP function
 - RANDOM function
- The expressions specified in a row partitioning expression must not contain any of the following system-derived columns, operators, or SELECT subqueries.
 - ROWID
 - PARTITION
 - PARTITION#L *n*
 - Set operators
 - Subqueries

Rules For Altering a Partitioning For a Table

General Guidelines

The following general rules and observations apply to the MODIFY option when you alter the partitioning of row-partitioned tables.

For more information, see [Modifying Partitioning Using the ADD RANGE and DROP RANGE Options](#), [Rules for Altering the Partitioning Expression for Multilevel Partitioning](#), and [Modifying Partitioning Using the ADD RANGE and DROP RANGE Options](#).

See also [Partitioned and Nonpartitioned Primary Indexes](#) under CREATE TABLE.

- If the data type for the result of *partitioning_expression* is not INTEGER, BIGINT, or CHARACTER, then the value is implicitly cast to the INTEGER type.
If the result type cannot be cast to INTEGER, the system returns an error to the requestor.
- If *partitioning_expression* for a partitioning level specifies only a RANGE_N function with INTEGER data type, the number of ranges defined for the function must be less than or equal to 2,147,483,647. See the description of the RANGE_N function in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.
- If *partitioning_expression* for a partitioning level specifies only a RANGE_N function with BIGINT data type, the number of ranges defined for the function must be less than or equal to 9,223,372,036,854,775,805 for 8-byte partitioning or 65,533 for 2-byte partitioning. See the description of the RANGE_N function in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.
- If you do not specify the RANGE_N or CASE_N functions to define a partitioning level, the number of defined partitions for a row partitioning level is the number of row partitions specified by the RANGE_N or CASE_N function, or 65,535.

Using the ADD Clause

The following rules are true when using the ADD Clause.

- If you specify an ADD clause, the maximum number of partitions for a partitioning level is the number of defined partitions for that level plus the value of the INTEGER constant specified by the ADD clause.
If this maximum exceeds 9,223,372,036,854,775,807 for 8-byte partitioning or 65,535 for 2-byte partitioning, the system returns an error to the requestor.
- If you do not specify an ADD clause for a partitioning level and it is the only level of partitioning for the table, the maximum number of partitions for that level, including the two partitions reserved for internal use, is 65,534.
- If the following things are true for a column-partitioned table or join index:
 - you do not specify an ADD clause for the column partitioning level
 - the table is also row-partitioned
 - at least one of the row partitioning levels does not specify an ADD clause,

the maximum number of partitions for the column partitioning level is the number of column partitions you define plus 10. The default in this case is ADD 10.

- If the following things are true for a column partitioning level:
 - you do not specify an ADD clause for the column partitioning level and the table is also row-partitioned
 - all of the row partitioning levels specify an ADD clause
 - using the number of column partitions defined plus 10 as the maximum number of column partitions, the table or join index has 2-byte partitioning,

the maximum number of partitions for the column partitioning level is the largest value that does not cause the partitioning to be 8-byte partitioning.

Otherwise, the maximum number of partitions for the column partitioning level is the largest value that does not cause the limit to exceed 9,223,372,036,854,775,807.

If there is no such largest value, the system returns an error to the requestor.

- If the following things are true for a partitioned table:
 - the row partitioning level does not specify an ADD clause
 - the number of defined row partitions is the current maximum for this and any lower row partitioning level without an ADD clause – a table has 2-byte partitioning,

the maximum number of partitions for each row partitioning level is the largest value that does not cause the partitioning to become 8-byte partitioning.

Otherwise, the maximum number of partitions for the level is the largest value that does not cause the combined partition number to exceed 9,223,372,036,854,775,807 for 8-byte partitioning or 65,535 for 2-byte partitioning.

- You can specify ADD 0 for a partitioning level to specify that the maximum number of partitions for this level is the same as the number of defined partitions in the following cases:
 - for a column partitioning level; this is useful if you want to override the default of ADD 10 so that other levels can have more partitions
 - for a row partitioning level; this is useful if you want a lower level that does not specify the ADD clause to have any excess partitions
- The maximum number of partitions for a row partitioning level must be at least 2.

If it is not, the system returns an error to the requestor.

This error occurs when only 1 partition is defined for a row partitioning level with an ADD 0 or with no ADD option and the maximum is not increased to at least 2.

- The following table summarizes to which partitioning levels Vantage adds any excess combined partitions.

| IF the partitioning is ... | AND ... | THEN as many leftover combined partitions as possible are added to ... |
|----------------------------|---|---|
| single-level | | the first and only row or column partitioning level. If an ADD value is specified, Vantage overrides it. |
| multilevel | all the row partitioning levels have an ADD clause, but there is a column partitioning level without an ADD clause | the column partitioning level, which does not need to be the first partitioning level. |
| | a column partitioning level and at least one of the row partitioning levels does not have an ADD clause, including the case where none of the row partitioning levels have an ADD clause specified | the first row partitioning level without an ADD clause after using a default of ADD 10 for the column partitioning level. This is repeated for all of the other row partitioning levels without an ADD clause in level order. |
| | a column partitioning level has an ADD clause and at least one of the row partitioning levels does not have an ADD clause | the first row partitioning level without an ADD clause. This is repeated for all of the other row partitioning levels without an ADD clause in level order. |
| | there is no column partitioning level and at least one of the row partitioning levels does not have an ADD clause, including the case where none of the row partitioning levels has an ADD clause specified | |
| | all the partitioning levels have an ADD clause or after applying leftover combined partitions as defined in the next column of this table | the first row or column partitioning level and Vantage overrides the ADD clause for the first partitioning level if one is specified. If there at least one level with an explicit ADD clause, at least one level that consists solely of a RANGE_N function with BIGINT data type, there is column partitioning, or the partitioning is 8-byte, then this is repeated for each of the other levels from the second level to the last. |

Row Partitioning

The following rules are true for row partitioning.

- If a new partitioning expression is defined with a NO RANGE partition, by definition this partition contains any row that does not fall into an explicitly defined partitioning expression value range (see the description of the RANGE_N function in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for an explanation of the function of the NO RANGE partition).

As a result, once a range partition is dropped, any row that had been assigned to that partition must then be assigned to the NO RANGE partition.

- If evaluation of a new partitioning expression for a row causes evaluation errors (such as divide by zero), then Vantage rolls back any changes made to the table or to *save_table* and does not modify the partitioning.
- The new partitioning expressions become the effective partitioning expressions for the table as soon as it is altered successfully.
- If there is a subsequent attempt to insert or update a row of a row-partitioned table or a column-partitioned table that also has row partitioning such that the partitioning expression for that row does not generate, after casting to INTEGER if it is not already typed as INTEGER or CHARACTER, a value between 1 and 65,535, then the system returns an error for the insert or update operation.
- Increasing the number of active row partitions for a partitioned table might degrade the performance of primary index accesses and joins, but allows for finer row partition elimination (see *Teradata Vantage™ - Database Design*, B035-1094 for details).

The general usage rules for altering a character-row-partitioned table are the same as the rules for non-character row-partitioned tables only for the following cases:

- The table has no rows and the partitioning is changed by specifying a new partitioning expression. If the table is populated with rows, the system returns an error to the requestor.
- The table is character-row-partitioned with one or more row partitioning levels that do not specify character data comparisons.

You can only add or drop non-character row partitioning levels for the following cases:

- The partitioning levels are defined with a RANGE_N function
- The session collation must be identical to the table or join index collation
- The session transaction semantics are the same as those that were in effect when the table or join index was created

Otherwise, the system returns an error to the requestor.

- You can only alter character row partitioning levels that are defined using a RANGE_N function.
- You can only alter character row partitioning levels to add or drop a NO RANGE partition, an UNKNOWN partition, or both.

Also see [Rules for Altering the Row Partitioning for Character Partitioning](#).

Using CASE_N

The usage rules for using CASE_N partitioning expressions in a character-row-partitioned table created with an ALTER TABLE statement are the same as those for a non-character-row-partitioned table with the following exceptions:

- To change the character-based row partitioning of a table using an ALTER TABLE ... MODIFY ... PARTITION BY statement, the table must be empty.

This applies to row-partitioned tables and column-partitioned tables that also specify one or more character row partitioning levels in their partitioning expression.

- If the new partitioning expression is for a character-row-partitioned table, the session collation in effect during the ALTER TABLE statement becomes the collation of the modified table.
- You cannot alter the partitioning of a character-row-partitioned table using ADD RANGE or DROP RANGE clauses for CASE_N expressions.

The system determines the case sensitivity of character column references and literals, which also affects comparison evaluation, based on the session default or any explicit CAST expression in the CREATE TABLE statement when the table is created.

You can also explicitly assign columns to be CASESPECIFIC or NOT CASESPECIFIC, and you can also CAST constant expressions with those qualifiers.

| FOR this session mode ... | Vantage uses the following default case specificity ... |
|---------------------------|---|
| ANSI | CASESPECIFIC |
| Teradata | NOT CASESPECIFIC |

A CASE_N partitioning expression can specify the UPPERCASE column attribute and the following functions.

- CHAR2HEXINT
- INDEX
- LOWER
- MINDEX
- POSITION
- TRANSLATE
- TRANSLATE_CHK
- TRIM
- UPPER
- VARGRAPHIC

Modifying Row Partitioning Defined With an Updatable DATE, CURRENT_DATE, or CURRENT_TIMESTAMP

Vantage provides a special version of the ALTER TABLE statement for modifying the row partitioning of a table (or WHERE clause of a join index definition) defined with an updatable DATE, updatable CURRENT_DATE, or updatable CURRENT_TIMESTAMP built-in function called ALTER TABLE TO CURRENT (see [ALTER TABLE TO CURRENT](#)).

ALTER TABLE TO CURRENT is designed specifically to reconcile the values of these functions when they are specified in a partitioning expression of a table or join index or in the WHERE clause of a join index.

The rules for ALTER TABLE TO CURRENT requests are identical for row partitioning and column partitioning; however, there can be a negative performance impact if the table or join index has column

partitioning in addition to row partitioning because it is often costly to move a row from one row partition to another.

See the table in [Comparing ALTER TABLE TO CURRENT and ALTER TABLE ... REVALIDATE](#) for a comparison of the capabilities of ALTER TABLE ... REVALIDATE PRIMARY INDEX requests with the ALTER TABLE TO CURRENT statement.

Rules for Adding and Dropping Ranges in a Character Partitioning Expression

The following rules apply to the ADD RANGE and DROP RANGE options for character partitioning expressions.

- ALTER TABLE ... DROP RANGE WHERE is only supported for partitioning expressions that are defined without character or graphic comparisons.
- If you specify the ADD RANGE or DROP RANGE options for a non-character partitioning expression of a multilevel partitioned table, or on the special partitions of a RANGE_N character partitioning expression, and at least one of the partitioning expressions has character partitioning, then the session collation must be the same as the table collation.

Otherwise, the database returns an error to the requestor.

- If the table collation is MULTINATIONAL or CHARSET_COLL, the same collation or character set must be installed and its definition must be the same as was in effect when the table was created.

Otherwise, the database returns an error to the requestor.

Rules for Altering the Row Partitioning for Character Partitioning

See also [Partitioning Expressions Using on a CASE_N or RANGE_N Character Column](#).

The following rules and restrictions apply in addition to all other rules for using ALTER TABLE requests to modify row-partitioned tables:

You can use ALTER TABLE requests to modify character partitioning only as described below.

PARTITION BY Expression on a Table Not Populated with Rows

The table is not populated with rows and the partitioning is changed by specifying a new PARTITION BY expression.

Partitioning Levels that Do Not Specify Character Data Comparisons

The character partitioning is multilevel, with at least one, and possibly several, partitioning levels that do not specify character data comparisons.

You can alter those non-character expression partitioning levels that consist only of a RANGE_N function using the ADD RANGE or DROP RANGE options in the same way as you would for a non-character partitioning.

The session collation must be the same as the table collation, and the session mode must also be the same as was in effect when the table or join index was created.

For this example, suppose you have created the following row-partitioned *orders* table in Teradata session mode in database *df2* with ASCII collation.

```
CREATE SET TABLE df2.orders, NO FALLBACK, NO BEFORE JOURNAL,
  NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1)
                  CHARACTER SET UNICODE NOT CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21)
                  CHARACTER SET UNICODE NOT CASESPECIFIC,
  o_comment       VARCHAR(79)
                  CHARACTER SET UNICODE NOT CASESPECIFIC)
PRIMARY INDEX OrdPI ( o_orderkey )
PARTITION BY (RANGE_N(o_orderpriority BETWEEN 'high'
                      AND 'highest',
                      'low'
                      AND 'lowest',
                      'medium'
                      AND 'medium',
                      NO RANGE OR UNKNOWN) )
UNIQUE INDEX (o_orderkey);
```

The following example shows the output of a request for this table against the DBC.IndexConstraints view using BTEQ. For the definition of this view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

```
SELECT *
FROM DBC.IndexConstraints
WHERE TableName = 'Orders'
AND   DatabaseName = 'DF2';

*** Query completed. One row found. 11 columns returned.
*** Total elapsed time was 1 second.
```

The DBC.IndexConstraints.CharSetID column contains the code of the same column name from DBC.CharTranslations representing the character set used as the collation when the corresponding table uses CHARSET_COLL collation.

The DBC.IndexConstraints.SessionMode column identifies the session mode, either ANSI or Teradata, in effect when the associated character partitioning was created or last modified.

The following report is an example of what you might see if you were to use BTEQ to run the previous SELECT example request against DBC.IndexConstraints.

| | |
|---------------------|---|
| DatabaseName | DF2 |
| TableName | Orders |
| IndexName | OrdPI |
| IndexNumber | 1 |
| ConstraintType | Q |
| ConstraintText | CHECK ((RANGE_N(o_orderpriority BETWEEN 'high' AND 'hi |
| ConstraintCollation | A |
| CollationName | ASCII |
| CreatorName | DF2 |
| CreateTimeStamp | 2015-08-20 13:02:30 |
| CharSetID | 127 |
| SessionMode | T |

Partitioning Levels that Use Only a RANGE_N Function

The partitioning specifies character row partitioning levels that consist only of a RANGE_N function. You can only alter those levels to ADD or DROP the special NO RANGE and UNKNOWN partitions.

Row-partitioned Table Attributes You Can Change

You can change the following attributes of a row-partitioned table using ALTER TABLE statements.

- Table name
- The component primary index or primary AMP index columns if the table has a primary index, and the partitioning expression, but only for an unpopulated table
- If the table is primary-indexed, you can change its UPI to a NUPI.
- If the table is primary-indexed, you can sometimes change its NUPI to a UPI.

Table Must Be Unpopulated

To change the character partitioning of a table using an ALTER TABLE ... MODIFY PARTITION BY statement, the table must be unpopulated with rows.

Session Collation for the Modified Table

If the new partitioning also specifies character partitioning, the session collation in effect at the time the ALTER TABLE statement is submitted becomes the collation for the modified table.

Partitioning-related optimizations such as row partition elimination and deferred row partition delete operations involving the character row partitioning levels are restricted to SQL requests having the same session collation as was in effect when the table was created.

Vantage still uses row partition elimination on the non-character row partitioning levels of a character multilevel partitioning when collations do not match.

Partitioning CHECK Constraints

Vantage derives a partitioning CHECK constraint from the partitioning expressions. See [Partitioning CHECK Constraints for Partitioned Tables With 2-Byte Partitioning](#) and [Partitioning CHECK Constraint for Partitioned Tables with 8-Byte Partitioning](#). The text for this index CHECK constraint must not exceed 16,000 characters, or roughly 2,000 partitions.

Character partitioning typically constrains the number of partitions you can define to a much smaller number than you can define for non-character partitioning.

The effective limit for character partitioning is roughly 2,000 partitions because of the 16,000 character limit on partitioning CHECK constraints. For very large tables, the most effective character partitioning in most cases is as part of a multilevel partitioning expression. Combining predicates on multiple partitioning levels might then reduce the ratio of combined partitions that need to be read from the rough limit of $\frac{1}{2,000}$ for a single-level character partitioning to up to the maximum number of partitions that can be defined for a table.

Using ADD RANGE or DROP RANGE for Multilevel Partitioning

You can alter character partitioning using the ADD RANGE or DROP RANGE options for multilevel partitioning if the partitioning levels being added or dropped obey one or all of the following rules.

- Use only RANGE_N partitioning.
- Do not make character data comparisons.
- Specify one or both of the following special partitions.
 - NO RANGE
 - UNKNOWN

The following rules apply:

| IF the ... | THEN ... |
|---|--|
| session collation does not match the table collation | the request aborts and the system returns an error to the requestor. |
| table collation is MULTINATIONAL and the installed MULTINATIONAL collation has changed since the table was created | |
| table collation is CHARSET_COLL and the definition of the character set in <i>DBC.Translation</i> has changed since the table was created | |
| session mode does not match the session mode that was in effect when the table was defined | |

If any of these errors occur, you must perform an ALTER TABLE ... REVALIDATE request to revalidate the partitioning.

Rules for Altering the Partitioning Expression for Multilevel Partitioning

The following rules apply only to altering the partitioning expression for multilevel partitioning.

Use a RANGE_N or CASE_N Function

Each new partitioning expression must be defined using only a RANGE_N function or a CASE_N function. It is not required that the partitioning expressions for a row-partitioned table or that the row partitioning levels for a column-partitioned table or join index be all of one form or the other, only that each individual partitioning expression must be constructed from either a RANGE_N or a CASE_N function.

Maximum Number of Partitions

The product of the number of partitions defined by each new partitioning expression cannot exceed 9,223,372,036,854,775,807 for 8-byte partitioning or 65,535 for 2-byte partitioning.

For 8-byte partitioning, the maximum number of new partitioning expressions is 62. This is because $2^{63} = 9,223,372,036,854,775,808$, which is one larger than the valid upper limit on the number of combined partitions for a table or join index.

For 2-byte partitioning, the maximum number of new partitioning expressions is 15. This is because $2^{16} = 65,536$, which is one larger than the valid upper limit on the number of combined partitions for a table or join index with 2-byte partitioning.

Note:

$2^{63} - 1 = 9,223,372,036,854,775,807$, which is the maximum number of combined partitions for 8-byte partitioning of a table or join index.

$2^{16} - 1 = 65,535$, which is the maximum number of combined partitions for 2-byte partitioning of a table or join index.

The system returns an error to the requestor when only one partition is defined for a row partitioning level with an ADD 0 or with no ADD specification and the maximum is not increased to at least 2.

Minimum of 2 Partitions per Row Partitioning Level

As a general rule, the number of partitions for a row partitioning level must be at least 2, with the following exceptions.

For each row partitioning level that does not specify an ADD clause in level order, the maximum number of partitions for the row partitioning level is as follows.

| Table or join index partitioning | Maximum number of partitions for a row partitioning level |
|----------------------------------|---|
| 2-byte partitioning | Cannot cause the partitioning to be 8-byte partitioning. |
| 8-byte partitioning | Cannot exceed 9,223,372,036,854,775,807. |

The following table defines the maximum number of row partitions for a row partitioning level that define the maximum number of partitions for the first level to the largest value that does not cause the maximum combined partition number to exceed the maximum.

| Table or join index partitioning | Maximum number of partitions for a row partitioning level |
|----------------------------------|--|
| 2-byte partitioning | Cannot cause the maximum combined partition number to exceed 65,535. If there is at least one level with an explicit ADD clause, there is at least one level that consists solely of a RANGE_N function with BIGINT data type, or there is column partitioning, this is repeated for each of the other levels, if any, from the second level to the last. |
| 8-byte partitioning | Cannot cause the maximum combined partition number to exceed 9,223,372,036,854,775,807. This is repeated for each of the other levels from the second level to the last. |

The maximum number of new partitioning expressions is 62.

If more than two partitions are defined at one or more levels, the number of new partitioning expressions can be further limited.

Rules and Restrictions for Modifying Column-Partitioned Tables

Following are the rules and restrictions for modifying column-partitioned tables:

MODIFY NO PRIMARY

If you specify MODIFY NO PRIMARY, you must specify the NOT PARTITIONED option or a PARTITION BY clause.

You can also specify MODIFY NO PRIMARY for nonpartitioned NoPI tables, in addition to column-partitioned tables.

Specifying COLUMN partitioning in a PARTITION BY clause

You cannot specify MODIFY NO PRIMARY and NOT PARTITIONED on a table without a primary index or primary AMP index which is not partitioned.

If you alter a SET table to be a table with MODIFY NO PRIMARY, the system alters the table kind to be MULTiset by default. See [Rules for Modifying Populated and Unpopulated Tables and Join Indexes](#).

The table being modified must not be populated with rows in the following cases:

- Table with a primary index or primary AMP index being modified to have no primary index
- NoPI or column-partitioned table being modified to have a primary index or primary AMP index.
- Partitioned table being modified to have no partitioning.
- Nonpartitioned table being modified to have partitioning.
- Partitioned table being modified to have a new partitioning using a PARTITION BY clause.

New Partitioning Using DROP and ADD RANGE Clauses

To modify a partitioned table to have a new partitioning, you must define the new partitioning expressions using `DROP RANGE#L n` and `ADD RANGE#L n` options that do not drop ranges other than from the beginning or end of the range sequence with at least one remaining range, and do not add ranges between the resulting beginning and end of the ranges.

If the table has multilevel partitioning and the modification of a partitioning expression includes dropping the `NO RANGE [OR UNKNOWN]` or `UNKNOWN` partitions, that partitioning expression must not have previously been modified.

The resulting number of partitions for a level must be between 1 and the maximum defined for that level.

Rules for Retaining Eligibility for Restoring or Copying Selected Partitions

Although you can change some of the characteristics of a table and retain the ability to restore or copy selected partitions to it, other operations on a partitioned table render the table unsuitable for selected partition restorations.

The following table definition changes do not invalidate subsequent restorations of selected partitions for the targeted or referenced tables.

- Altering table-level options that are not related to the semantic integrity of the table, including the following.
 - Fallback protection (see [ALTER TABLE and FALLBACK](#)).
 - Journaling attributes (see [Transient Journaling](#)).
 - Free space percentage.
 - Data block size (see [IMMEDIATE DATABLOCKSIZE](#)).

Also see [CREATE TABLE \(Table Options Clause\)](#) for more information about these options.

- Altering the partitioning expression. See [Rules For Altering a Partitioning For a Table](#) and [CREATE TABLE \(Index Definition Clause\)](#) for more information about these options.
- Altering either column-level or table-level CHECK constraints. See [Adding and Dropping CHECK Constraints](#) and [CREATE TABLE \(Column Definition Clause\)](#) for more information about these options.

When the target and source are different systems, you must repeat each of the previously listed operations on the affected tables of the target system to ensure that the two are kept in synchrony.

The following ALTER TABLE operations *do* make any future restore or copy operations of selected partitions for targeted or referenced tables non-valid.

- Adding or dropping columns (see [Dropping Columns From a Table](#) and [Adding Columns To a Table](#)).
- Altering the definition of existing columns (excluding the exceptions noted in the previous list).
- Adding, dropping, or modifying referential integrity relationships between tables (see [Adding or Dropping Standard Referential Integrity Constraints](#)).

See *Teradata Vantage™ - Database Administration*, B035-1093 for complete warnings, procedures, and other information related to partial restoration of partitions.

ALTER TABLE (REVALIDATE Option)

General Rules and Restrictions for the REVALIDATE Option

The REVALIDATE option regenerates the table headers for a partitioned table or noncompressed join index and optionally verifies and corrects the row partitioning for a table (but not for a join index) if you specify the WITH DELETE or WITH INSERT null partition handling options.

You can specify the REVALIDATE option for nonpartitioned tables and join indexes without also specifying PRIMARY INDEX. This makes revalidation applicable whether or not the table or join index has a primary index.

You must specify the REVALIDATE option without also specifying PRIMARY INDEX for NoPI and primary AMP index tables and join indexes. Otherwise, the database returns an error to the requestor.

The REVALIDATE option also provides the ability to revalidate some data dictionary columns.

REVALIDATE requests do not verify the column partitioning of a column-partitioned table or join index.

If partitioning errors are detected for a table (but not a join index), you can use an ALTER TABLE ... REVALIDATE statement including the WITH clause to correct them.

REVALIDATE also revalidates the following Data Dictionary table columns.

| Table | Column Name | Validation Action Taken |
|----------------------|--|---|
| DBC.TVM | PIColumnCount | Set to the number of primary index columns if the value is currently 0. |
| | PartitioningLevels | Set to the number of partitioning levels if the value is currently 0. |
| DBC.TVFields | PartitioningColumn | Set to Y if the value is currently N. |
| DBC.TableConstraints | <ul style="list-style-type: none"> DefinedCombinedPartitions MaxCombinedPartitions PartitioningLevels | Set to the appropriate values if the value is currently 0. |

After an upgrade, the Data Dictionary columns in the preceding table might not be set correctly for existing tables and join indexes that were created before those columns were added to the Data Dictionary.

Regardless of whether the table or join index has a primary index, you should submit an ALTER TABLE statement with the REVALIDATE option to correct the Data Dictionary column values

You may need to revalidate the table headers for a partitioned table or join index when any of the following conditions exist.

- The table headers for a partitioned table or join index are marked as nonvalid.

This is most often found after an upgrade or after restoring or copying a table to a system that has a different operating system than the source or that runs on different hardware.

- The RoundHalfwayMagUp parameter in the DBS Control record is changed and a partitioning expression specifies DECIMAL operations.

A change in DECIMAL rounding rules can cause the partitioning expression to evaluate differently.

If this occurs for a partitioned join index, drop the index and then recreate it. See [CREATE JOIN INDEX](#).

- The table or join index is restored or copied to another system that has a different RoundHalfwayMagUp setting in its DBS Control record and the partitioning expression for the table or noncompressed join index specifies DECIMAL operations.

A change in DECIMAL rounding rules can cause the partitioning expression to evaluate differently.

- The table or join index is restored or copied to another system and the partitioning expression specifies floating point operations.

Floating point calculations might cause the partitioning expression to evaluate differently on different systems.

- The table or join index is restored or copied to another system that has a different hashing algorithm and the partitioning expression specifies a HASHROW or HASHBUCKET function.

A different hashing function can cause the partitioning expression to evaluate differently.

- The database is upgraded in such a way that the calculation of the partitioning expressions for tables and join indexes changes.
- The CheckTable utility or valid SQL queries indicate rows with incorrect internal partition numbers or rows in the wrong partition for the table or join index.

If this occurs unexpectedly, please report the problem to the Teradata Support Center immediately.

If this occurs for a join index, drop the index and then recreate it. See [CREATE JOIN INDEX](#).

The following rules apply to the REVALIDATE option.

- You cannot specify REVALIDATE PRIMARY INDEX for a table or join index that does not have a primary index.
Otherwise, the database returns an error to the requestor.
- You cannot modify index or partitioning definitions and revalidate a primary index or partitioning within a single ALTER TABLE request because the operations are mutually exclusive.
- You cannot specify a WITH DELETE or WITH INSERT null partition handling clause for a table that does not have row partitioning.

Otherwise, the database returns an error to the requestor.

- You cannot specify a WITH DELETE or WITH INSERT null partition handling clause for a join index because the null partition handling options are only valid for tables.

If you attempt to revalidate a join index using a null partition handling clause, the database returns an error to the requestor.

If you suspect or detect a problem with the rows of a noncompressed join index, you must drop and then recreate it.

- If you specify neither a WITH DELETE nor a WITH INSERT null partition handling clause for the option, then only table headers are regenerated and no error checking or repair is performed.
- If you specify either a WITH DELETE or a WITH INSERT null partition handling clause, then in addition to regenerating the table headers for the table, all the rows in a base table are validated for their partition number and row hash value.
- If you specify a WITH DELETE null partition handling clause, then the database deletes any row for which a partitioning expression does not generate a value between 1 and 65,535 (after casting to INTEGER if the value is not already typed as INTEGER or CHARACTER).
- If you specify a WITH INSERT [INTO] *save_table* null partition handling clause, the database deletes any row for which a partitioning expression does not generate a value between 1 and 65,535 (after casting to INTEGER if the value is not already typed as INTEGER or CHARACTER from the table and inserts it into *save_table*.

The following rules apply to the use of a *save_table* with a null partition handling clause.

- *save_table* cannot be the same table as *table_name* or the system returns an error.
- *save_table* must have the same number of columns with compatible data types as *table_name*, otherwise the system returns an error.
- If errors occur inserting into *save_table*, then the following rollback behavior occurs, depending on the session mode in effect.

| IF the session is in this mode ... | THEN the following block of work is rolled back ... |
|------------------------------------|---|
| ANSI | request. |
| Teradata | transaction. |

- If evaluation of a partitioning expression for a row causes evaluation errors, then the database rolls back any changes made to the table or to *save_table* and the partitioning is not changed.

Specific rollback behavior depends on the session mode in effect.

| IF the session is in this mode ... | THEN the following block of work is rolled back ... |
|------------------------------------|---|
| ANSI | request. |
| Teradata | transaction. |

If this occurs, you should correct the problem using one of the following solutions.

- Make the primary index for a primary-indexed table or noncompressed join index a nonpartitioned primary index.
- Change the partitioning expression to one that does not generate evaluation errors.
- Delete rows that cause evaluation errors.
- Drop the table.

- For the remaining rows in the table, the database updates the internal partition numbers and row hash values as needed and, if the row is not in the correct partition or row hash, it is moved to the correct location.

Rows that are out of order by their assigned ROWID are not corrected and the database returns an error to the requestor.

- The database updates any secondary indexes, join indexes, or hash indexes defined on the table as needed.
- The values of the system-derived PARTITION or PARTITION#L[n] columns might change for the rows in the table.
- If a partitioning expression for a table is affected by a time zone string (see [Daylight Saving Time and Time Zone Strings Specified As Time Zone Strings](#)) and the time zone rules for that string are changed, you must revalidate the table using the WITH INSERT or WITH DELETE null partition handling options.

The following rules apply to the REVALIDATE option for character partitioning.

- Character-based partitioning might need to be revalidated if the following are all true.
 - The table or join index was created with the CHARSET_COLL session collation, in which case the database uses the code point ordering of the session character set in effect when the table was created for the table or join index collation.
 - The collation character set is not one of the following predefined character sets.
 - ASCII
 - EBCDIC
 - UTF-8
 - UTF-16
 - The character set in the *DBC.Translation* table has been modified since the table was created.
- Character-based partitioning might need to be revalidated if the following are both true.
 - The table or noncompressed join index was created with MULTINATIONAL collation.
 - A different MULTINATIONAL collation has been installed in DBC.Collations since the table or noncompressed join index was created.
- Character-based partitioning might need to be revalidated if the character partitioning expression involves one or more Unicode character expressions or literals and the database was backed down to a previous minor release that has a different Unicode character set definition.

The table or join index might need to be revalidated if the later release has code points that are undefined at the earlier release.

- If the character-based partitioning you need to revalidate is defined on a join index, then you must drop and then recreate the index.
- When the system rebuilds the character-based partitioning and moves rows to their correct partitions (assuming that you have specified a null partition handler), it uses the collation associated with the

original table or join index collation in the regenerated partitioning expression, not the session collation in effect when the ALTER TABLE ... REVALIDATE request is submitted.

- The database uses the default case sensitivity for the session mode that was in effect when the table or join index was created or last had its partitioning altered in the regenerated partitioning expression.
- The collation character set in effect when the table or join index was created must exist in DBC.CharTranslations, must be installed, and must have the same numeric code (in the CharSetID column) as it did when the table or noncompressed join index was created or be one of the following predefined character sets.
 - ASCII
 - EBCDIC
 - UTF-8
 - UTF-16

If the character set is no longer installed, the database returns an error to the requestor.

If you cannot reinstall the character set that was in effect when the table or join index was created, then you cannot revalidate that table or join index. Instead, you must submit an appropriate INSERT ... SELECT request to move the rows into a new table in which you can either generate valid partitioning or remove the current partitioning as is appropriate.

- The following table indicates the differences in how the new collation for the character-based partitioning is handled.

| IF the partitioning collation is ... | THEN the database uses ... |
|--------------------------------------|--|
| MULTINATIONAL | the currently installed MULTINATIONAL collation as the new partitioning collation even if it differs from the MULTINATIONAL collation that was installed when the table or join index was created. |
| CHARSET_COLL | the collation character set used in the original partitioning collation for the new partitioning collation. |

Revalidating the Partitioning for Tables and Join Indexes With a HASH Function Partitioning Expression

During a cross platform migration process, the database regenerates the table headers for partitioned tables and join indexes after they are restored or copied to the new system.

If the new system either uses a different hashing algorithm or has 20-bit hash buckets rather than 16-bit hash buckets, a change in the value returned by a hash function can cause the partitioning expression to evaluate differently, and the regenerated table headers can cause incorrect query results to be returned from partitioned tables and join indexes when you specify a partitioning expression using the HASHBUCKET function. For information about the HASHBUCKET function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

You must revalidate the partitioning of any affected tables or join indexes using the following procedure.

1. Identify the partitioned tables and join indexes affected by this problem by running the `pre_upgrade_prep.pl` script prior to the migration. You should then run the `hashbucket_ppi.rpt` script afterward. The `pre_upgrade_prep.pl` script performs a SELECT request that finds and reports any tables or join indexes that must be revalidated after the migration occurs.

If you want to submit a standalone SELECT request to return the rows that must be revalidated, you can submit the following request, which duplicates the request the `pre_upgrade_prep.pl` script submits.

```
SELECT TRIM(DataBasenamei)||'.'||TRIM(TVMNamei), TableKind
FROM   DBC.TableConstraints AS tc JOIN DBC.TVM
      ON tc.TVMID=TVM.TVMID
      JOIN DBC.Dbase
      ON tc.DBaseId=Dbase.DatabaseId
WHERE  ConstraintType = 'Q'
AND    UPPER(TableCheck) LIKE '%HASHBUCKET%'
ORDER BY 1;
```

| TO revalidate this database object ... | GO to this step ... |
|--|---------------------|
| partitioned table | 2. |
| partitioned join index | 5. |

2. Revalidate the rows in each of the data tables identified by `pre_upgrade_prep.pl` (but not the join index rows).

To revalidate data tables, this revalidation requires the null partition handler specified by `save_table` to ensure that rows are stored in the correct partitions.

Note:

You cannot specify a WITH DELETE or WITH INSERT [INTO] `save_table` null partition handler clause for a partitioned join index.

The script uses the following procedure for partitioned tables:

- a. Create a save table using the DDL for the affected partitioned table, but *do not* specify any partitioning or secondary indexes for `save_table`.

This step only applies to revalidating a partitioned *table*. You cannot use a null partition handler to revalidate a partitioned join index.

- b. Submit the following ALTER TABLE request to revalidate the partitioning for `table_name`.

```
ALTER TABLE database_name.table_name
REVALIDATE
WITH DELETE;
```

or

```
ALTER TABLE database_name.table_name
REVALIDATE
WITH INSERT INTO save_table;
```

database_name

Specifies the name of the database or user that contains *table_name*.

table_name

Specifies the name of the affected partitioned table.

save_table

Specifies the name of the table created to save the null partitions that are created when you revalidate the partitioned table.

3. The action taken for this step depends on whether *save_table* is populated with rows or not.

| IF <i>save_table</i> ... | THEN ... |
|----------------------------|--|
| is not populated with rows | the revalidation process for the data table is complete. |
| is populated with rows | you have a choice of how to handle each row in <i>save_table</i> . <ul style="list-style-type: none"> • Redefine the partitioning of the original table so you can reinsert the row. or <ul style="list-style-type: none"> • Save the row in a separate table. |

4. After successfully handling each of the rows in *save_table*, drop the null partition handler save table.
5. Revalidate partitioned join index rows by dropping and then recreating each affected partitioned join index.

ALTER TABLE (Set Down and Reset Down Options)

SET DOWN and RESET DOWN Options

The SET DOWN and RESET DOWN options enable the database to attempt to avoid database crashes, whenever possible, by converting what would otherwise be a crash into a transaction abort and a snapshot dump for a selected set of normally fatal file system error events.

The database flags a table as being down when the maximum number of permissible down regions is exceeded on its data subtable. A table that is flagged down is no longer accessible for DML requests.

To mark only a portion of a table down, the database identifies a range of rows that it makes inaccessible to DML requests. This row range is called a *down region* and is also recorded in the table header. Any queries that do not access the down portions of the subtable continue to execute as they did before the region was flagged. Only requests that attempt to access rows in ranges that have been flagged as down abort their containing transaction.

In case the rollback processing for a transaction abort or system recovery attempts to process a row in a down region, the database ignores the rollback because the affected row set is inaccessible. Because ignoring the rollback creates a data integrity problem if the down region is later manually reset, the database flags down regions to indicate that rollback had been skipped for them.

SET DOWN

If you want to forcibly set a table down, you can submit an appropriate ALTER TABLE ... SET DOWN request. This flags the table as down on all AMPs. You cannot specify any other options when you submit an ALTER TABLE ... SET DOWN request.

RESET DOWN

To re-enable access to a down table, you can submit an appropriate ALTER TABLE ... RESET DOWN SQL request. This removes the down status flag from all AMPs in the system and makes the table available for all DML requests.

When you reset the down status flag for a table by submitting an ALTER TABLE ... RESET DOWN request, Vantage removes the down status flag in the table header pertaining to all the subtables from all the AMPs. The system also deletes the down regions from the table header on each AMP except for regions that were skipped during rollback and recovery processing.

To avoid potential transaction aborts due to file system faults, you should submit your ALTER TABLE ... RESET DOWN requests only after you repair the intermittent hardware errors that caused the table to be flagged as down in the first place. If the intermittent hardware issues that caused data blocks to be marked bad are not resolved beforehand, then a DML request can attempt to access them, triggering another round of file system aborts and flagging regions down. Any DML requests that access rows in data blocks that do not have problems continue to execute successfully.

Once the down region has been flagged, Vantage does not permit it to be reset using an ALTER TABLE ... RESET DOWN request. The only ways to reset such down regions are the following.

- Run the Table Rebuild utility. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- Submit a fast path DELETE ALL request, which effectively ignores the current rows in the table. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

If the down table is later rebuilt or restored from an earlier backup, Vantage resets the down status flag for the various subtables and moves their rows to unmarked data blocks and cylinders from the fallback copies.

ALTER TABLE TO CURRENT

Comparing ALTER TABLE TO CURRENT and ALTER TABLE ... REVALIDATE

The following table compares ALTER TABLE TO CURRENT and ALTER TABLE ... REVALIDATE. See [General Rules and Restrictions for the REVALIDATE Option](#).

| ALTER TABLE TO CURRENT | ALTER TABLE ... REVALIDATE |
|---|--|
| Periodically modify the partitioning. | <ul style="list-style-type: none"> • Updates partitioning information in the table header after an upgrade to a major release. • Corrects any improperly partitioned rows after restoring to a system architecture that is different from the one from which an archive was made. • Corrects improperly partitioned rows caused by a system problem. |
| Resolves the DATE, CURRENT_DATE, and CURRENT_TIMESTAMP to their current values. | Uses the most recently resolved DATE, CURRENT_DATE, and CURRENT_TIMESTAMP values. |
| Updates the partitioning information in the table header. | Updates the partitioning information in the table header. |
| <p>When reconciling the rows of the specified table or join index, Vantage skips any partition when it can determine that all of the rows in the partition would remain in that partition after reconciliation.</p> <p>This assumes that the rows of the table or join index are properly partitioned prior to submitting the ALTER TABLE TO CURRENT request.</p> | <p>When reconciling the rows of the specified table, Vantage scans all of its partitions if you specify a null partition handler.</p> <p>This assumes that the rows of the table might not be properly partitioned before you submitted the current ALTER TABLE ... REVALIDATE PRIMARY INDEX request.</p> <p>You cannot specify this option for join indexes.</p> |
| <p>Reconciles all of the rows of the specified table or join index.</p> <ul style="list-style-type: none"> • If the request specifies a null partition handler, Vantage deletes any rows that cannot be reconciled after saving them to a save table if the request specifies a WITH INSERT INTO <i>save_table</i> null partition handler. <p>You cannot specify this option for join indexes because the rows from a join index cannot be deleted using an ALTER TABLE TO CURRENT request.</p> <ul style="list-style-type: none"> • If the request does not specify a null partition handler, the system aborts the request if an existing row cannot be reconciled. | <p>Only reconciles the rows of the specified table or join index if the request specifies a null partition handler.</p> <p>If you specify a WITH INSERT null partition handler for a table, Vantage deletes any rows that cannot be reconciled after saving them in a save table if the request specifies a WITH INSERT INTO <i>save_table</i> null partition handler.</p> <p>You cannot specify this option for join indexes.</p> |

General Usage Guidelines and Rules for ALTER TABLE TO CURRENT Requests

Although you can specify the `DATE`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP` functions anywhere in a partitioning expression where a date or timestamp constant is valid, you should exercise great care in how you use them.

If you specify multiple ranges using a `DATE` or `CURRENT_DATE` function in one of the ranges, and then later reconcile the partitioning the range specified using `CURRENT_DATE` might overlap one of the existing ranges. If so, reconciliation returns an error to the requestor. If this happens, you must recreate the table with a new partitioning expression based on `DATE` or `CURRENT_DATE`. Because of this, you should design a partitioning expression that uses a `DATE` or `CURRENT_DATE` function in one of its ranges with care.

`DATE`, `CURRENT_DATE`, and `CURRENT_TIMESTAMP` functions in a partitioning expression are most appropriate when the data must be partitioned as one or more Current partitions and one or more History partitions, where the terms Current and History are defined with respect to the resolved `DATE`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP` values in the partitioning expression.

This enables you to reconcile a table or join index periodically to move older data from the current partition into one or more history partitions using an `ALTER TABLE TO CURRENT` request instead of redefining the partitioning using explicit dates that must be determined each time you alter a table using `ALTER TABLE` requests to `ADD` or `DROP` ranges. You should evaluate how a `DATE`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP` function will require reconciliation in a partitioning expression before you define such expressions on a table or join index.

For a join index, reconciliation includes its `WHERE` clause if its predicate specifies a `DATE`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP` function. Again, the actual current date or current timestamp value is always earlier than the resolved `DATE`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP` in the partitioning expression, and it is reconciled based on the newly resolved value after the `ALTER TABLE TO CURRENT` request completes.

Be aware that using an `ALTER TABLE TO CURRENT` request to reconcile rows with newly resolved `DATE`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP` values in their partitioning expressions can be expensive, both because of the time required to scan a table or join index to find rows that need to be reconciled and because of the time required to move or delete rows that need to be reconciled if you specify a `WITH DELETE` or `WITH INSERT` clause with the `ALTER TABLE TO CURRENT` request for a table. You cannot specify a null partition handler for an `ALTER TABLE TO CURRENT` request made on a join index.

If more than a small percentage of table rows need to be reconciled without optimizations, such as whole partition deletions, another form of partitioning might be more appropriate.

When you build your `ALTER TABLE TO CURRENT` requests, you should specify modified functions in your partitioning expressions such as `DATE - INTERVAL '2' DAY`, `CURRENT_DATE - INTERVAL '2' DAY`, or some other appropriate adjustment rather than specifying an unmodified `DATE` or `CURRENT_DATE` function because requests might be submitted in different time zones than the session time zone in which

the ALTER TABLE TO CURRENT request is submitted. Such adjustments ensure that the Optimizer query plans remain the same for the same request regardless of the session time zone.

Specifying a partitioning expression that uses a CURRENT_TIMESTAMP function avoids the time zone issues that occur with partitioning expressions that use a DATE or CURRENT_DATE function, and these partitioning expressions do not require any adjustments to be universally applicable.

The rules for ALTER TABLE TO CURRENT requests are identical for row partitioning and column partitioning; however, there can be a negative performance impact if the table or join index has column partitioning in addition to row partitioning because it is often costly to move a row from one row partition to another.

The following general rules apply to the ALTER TABLE TO CURRENT statement.

- A table or join index that is partitioned using a DATE or CURRENT_DATE function, a CURRENT_TIMESTAMP function, or both in the partitioning definition can be reconciled to a newly resolved date or timestamp using an ALTER TABLE TO CURRENT request, which resolves the values of DATE, CURRENT_DATE, and CURRENT_TIMESTAMP based on their current values and reconciles the partitioning accordingly.
- If you submit an ALTER TABLE TO CURRENT request and the partitioning on the specified table or join index does not specify a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function, the database returns an error to the requestor.

A join index can also specify a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function in its WHERE clause. In that case, the database neither aborts the request nor does it return an error to the requestor.

An error should only occur if neither the partitioning expression nor the WHERE clause of a join index definition specifies a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function.

- When you submit an ALTER TABLE TO CURRENT request, the database evaluates the partitioning expressions using the newly resolved current date or current timestamp value. The rows in the table or join index being reconciled are scanned and adjusted by moving rows into new partitions or by deleting rows if they are no longer associated with a partition as a result of the ALTER TABLE TO CURRENT operation.
 - If you specify a null partition handler for a join index, the database returns an error to the requestor. This happens because you cannot delete rows from a join index using this statement.
 - If you do not specify a null partition handler and there are rows that are invalidated by the newly reconciled partitioning expression, the database returns an error to the requestor.

Because you cannot specify a null partition handler for a join index, partitioning defined on the index using DATE, CURRENT_DATE, or CURRENT_TIMESTAMP must be such that an ALTER TABLE TO CURRENT request does not cause this error. Otherwise, the join index may not reconcile to a new date or timestamp value.

In this case, you must drop the existing join index and then create a new index with an appropriate partitioning definition.

- The database updates all relevant secondary, join, and hash indexes as needed if table rows of the table are moved or deleted during reconciliation.

- If you specify either a WITH DELETE clause or a WITH INSERT clause, you must disable any delete triggers on *table_name*.

You must also disable any insert triggers on *table_name* if you specify a WITH INSERT clause. If you do not disable such insert triggers, then the database returns an error to the requestor.

You must remember to re-enable any such disabled triggers after you have successfully submitted your ALTER TABLE TO CURRENT request.

- If a newly resolved date or timestamp value evaluates to an earlier date or timestamp than the previously reconciled date or timestamp, the database returns an error to the requestor.

While it is possible for such dates or timestamps to occur because of clock drifts, adjustments to the system clock, or session time differences for a date, reconciliation should occur infrequently enough that a clock reset or adjustment does not cause the regression of current date or timestamp values.

The projected usage of reconciliation is based on the assumption that date and timestamp values always progress, so such date and timestamp resolution problems are not expected to occur.

- The latest resolved date and timestamp for a base table or join index is maintained in table DBC.TVM.
- If a newly resolved date evaluates the starting expression (containing the DATE or CURRENT_DATE function) of a RANGE_N function to a partition boundary, the database drops all of the partitions that are earlier than this partition. Otherwise, the database repartitions the entire table using the new partitioning expression.

For example, consider the following CREATE TABLE request submitted on April 1, 2006.

```
CREATE TABLE ppi (
  i INTEGER,
  j DATE)
PRIMARY INDEX(i)
PARTITION BY RANGE_N(j BETWEEN CURRENT_DATE
                      AND   CURRENT_DATE+INTERVAL '1' YEAR -
INTERVAL '1' DAY
                      EACH INTERVAL '1' MONTH);
```

In this example, consider the last resolved date to be April 1, 2006 and assume that, when the value for DATE or CURRENT_DATE is DATE '2006-06-01', you submit an ALTER TABLE TO CURRENT request. The starting expression with the newly resolved DATE or CURRENT_DATE value falls on a partition boundary of the third partition; therefore, the database drops partitions 1 and 2, and the last reconciled date is set to the newly resolved value for DATE or CURRENT_DATE.

Now suppose you submit an ALTER TABLE TO CURRENT request on DATE '2006-06-10'. The starting expression with the newly resolved DATE or CURRENT_DATE value does *not* fall on a partition boundary, so the database scans all of the rows, and repartitions them based on the new partitioning expression. The partition boundary after this request aligns to the tenth day of a month instead of the earlier first day of a month.

- With an updatable DATE or CURRENT_DATE value in a partitioning expression, it becomes possible for a partitioning expression based on a RANGE_N function to become obsolete after some time passes. Exercise great caution specifying RANGE_N functions for such cases, only doing so after you fully understand its implications for reconciliation and its applicability as the value changes each time you reconcile the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP value.

For example, consider the following CREATE TABLE request.

```
CREATE TABLE ppi (
  i INTEGER,
  j DATE)
PRIMARY INDEX(i)
PARTITION BY RANGE_N(j BETWEEN CURRENT_DATE
                      AND    DATE '2008-01-01'
                      EACH INTERVAL '1' MONTH);
```

If you reconcile this table using an ALTER TABLE TO CURRENT request after January 1, 2008, the request aborts and the database returns an error to the requestor because all the defined ranges are null.

Although you can specify the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions anywhere in a partitioning expression that a date or timestamp constant is valid, you must take appropriate caution in doing so.

For example, reconciliation of rows with a newly resolved DATE, CURRENT_DATE, or CURRENT_TIMESTAMP value in a partitioning expression using an ALTER TABLE TO CURRENT request might be expensive both from the time required to scan the table to determine which rows need to be reconciled and because of the time to required to move or delete rows that require reconciliation.

If more than a small percentage of the rows in a table must be reconciled without the benefit of optimizations such as whole partition delete operations, some other form of partitioning might be more appropriate.

The cases provided in the next several topics demonstrate how reconciliation can be expensive and how it can sometimes be optimized.

Guideline for Refreshing Partition Statistics After An ALTER TABLE TO CURRENT Request

You might need to refresh the partition statistics for a table or join index after you resolve its date or timestamp values because, depending on the options you specify for the request, those statistics can become stale after an ALTER TABLE TO CURRENT request completes.

The following table presents guidelines for refreshing statistics on the columns specified in the partitioning expression of a table or join index after an ALTER TABLE TO CURRENT request.

| IF the ALTER TABLE TO CURRENT request ... | THEN partition statistics ... |
|---|---|
| specifies a null partition handler | are stale after the request completes. You should refresh partition statistics for this case. |
| does not specify a null partition handler | are not stale after the request completes. There is no need to refresh partition statistics for this case. |

If an ALTER TABLE TO CURRENT request is submitted for a table or join index, the PARTITION statistics for the affected table or join index always become stale and you must refresh them after the request completes.

Using ALTER TABLE TO CURRENT Requests to Reconcile Join Indexes

The ALTER TABLE TO CURRENT statement enables refreshing join index rows without having to drop, and then recreate, that index. The efficiency of using this method to refresh join index rows compared to the drop-and-create alternative depends on two factors.

- How frequently you must submit ALTER TABLE TO CURRENT requests to refresh join index rows.
- The type of DATE, CURRENT_DATE, or CURRENT_TIMESTAMP condition that is specified in the join index definition.

If the join index rarely requires refreshing and its DATE, CURRENT_DATE, or CURRENT_TIMESTAMP condition is such that Vantage must remove a large volume of old rows and then insert a large number of new rows, it is usually more efficient to drop and then recreate the join index.

For obvious cases, such as when the primary index column of a join index is updated because of a new DATE, CURRENT_DATE, or CURRENT_TIMESTAMP value, ALTER TABLE TO CURRENT internally deletes all of the rows from the join index and then rebuilds it using the new DATE, CURRENT_DATE, or CURRENT_TIMESTAMP value.

For the less obvious cases for which the DELETE ALL option followed by an index rebuild operation is more efficient, but for some reason is not used by ALTER TABLE TO CURRENT, you should consider using the drop-and-recreate method. If you decide to use this method, Vantage drops the existing privileges and statistics on the join index when it drops the join index, so be sure to factor the time required to grant the dropped privileges and to recollect statistics into your determination of whether it would be the better approach or not.

When you must issue ALTER TABLE TO CURRENT requests for the base table and the join indexes defined on that table, you should consider submitting the ALTER TABLE TO CURRENT requests only on those join indexes that specify a lower bound DATE, CURRENT_DATE, or CURRENT_TIMESTAMP condition first. A lower bound DATE, CURRENT_DATE, or CURRENT_TIMESTAMP condition is one that causes rows from the join index to be deleted when the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP is refreshed to the latest date or timestamp value, for example using a condition such as `j > CURRENT_DATE`.

The outcome of this method is that any join index maintenance resulting from the ALTER TABLE TO CURRENT request on the base table does not also need be submitted unnecessarily on those join index rows that would have been deleted by the ALTER TABLE TO CURRENT request on the join index.

Related Information

See the following statements for further information about altering tables or creating join indexes.

- [ALTER TABLE \(Basic Table Parameters\)](#)
- [CREATE JOIN INDEX](#)
- In *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144:
 - ALTER TABLE
 - ALTER TABLE TO CURRENT
 - CREATE JOIN INDEX

For further information about the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

For further information about column-partitioned tables and join indexes, see [CREATE TABLE](#), *Teradata Vantage™ - Database Design*, B035-1094, and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

ALTER TYPE

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Adding Attributes To a Structured UDT

The system automatically generates an associated observer and mutator method for each attribute you add to a UDT.

The following rules apply to adding attributes.

- The UDT that you specify must be a structured UDT.
- The specified UDT must not be referenced by any other database object.
- You cannot add an attribute to a UDT being referenced by some other database object, such as being used as the column data type of any table.

This includes:

- A table in any database for which the data type of one of the columns is the specified UDT.
- A structured UDT for which the data type of one of the attributes is the specified UDT.
- The specified UDT is referenced in a user-defined cast (see [CREATE CAST and REPLACE CAST](#)).
- A method or UDF in any database that references the specified UDT.
- An ordering or transform group defined for the UDT (see [CREATE ORDERING and REPLACE ORDERING](#) and [CREATE TRANSFORM and REPLACE TRANSFORM](#)).
- The name of an added attribute must be different than all existing attribute names for the specified UDT.
- The database creates a default observer and mutator method for each new attribute that you add.

The method signatures of the observer and mutator must be different than all existing method signatures for the specified UDT.

- You cannot add a character data type with a server data set of KANJI1 to a structured data type. Otherwise, the database returns an error to the requestor.

This means that you, as a UDT developer, must perform all the necessary cleanup on all of the following database objects *before* you add any attributes to a structured UDT.

- Casts associated with the UDT.
See [CREATE CAST and REPLACE CAST](#).
- Orderings for the UDT.
See [CREATE ORDERING and REPLACE ORDERING](#).
- Transforms for the UDT.

See [CREATE TRANSFORM and REPLACE TRANSFORM](#).

- Tables whose columns are typed with the UDT.

See [CREATE TABLE](#).

- Structured UDTs that use the UDT.

See [CREATE TYPE \(Structured Form\)](#).

- UDFs and methods that use the UDT as a parameter type.

See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#) and [CREATE METHOD](#).

You must also remove any dependent objects before you add attributes to a structured type. To assist you in this task, you can use a system-installed macro found in the *SYSUDTLIB* database that lists all dependencies and the order in which they must be dropped.

To do this, type the following request.

```
EXEC SYSUDTLIB.HelpDependencies('UDT_name');
```

where *UDT_name* is the name of the structured UDT to which you are adding an attribute set.

The ability to add multiple attributes using the ADD ATTRIBUTE option is a Teradata extension to the ANSI SQL:2011 standard. You have the option of either submitting ALTER TYPE requests that comply with the ANSI SQL standard by adding only one attribute at a time, or of adding multiple attributes simultaneously using the Teradata extension to the ANSI SQL:2011 syntax.

You cannot add more than 512 attributes at a time to a structured UDT.

Workaround for Adding More Than 512 Attributes to a Structured UDT

The number of attributes that can be added to a structured UDT per ALTER TYPE ... ADD ATTRIBUTE request varies between 300 and 512, depending on the platform. The limit is imposed because of restrictions on the availability of the Parser memory that is required to register the autogenerated observer and mutator methods for a structured type, not because of the limit on the total number of attributes that can be defined per structured type, which is roughly 4,000.

The workaround for this is just to submit as many individual ALTER TYPE ... ADD ATTRIBUTE requests, each having an upper limit of 512 attributes, as required to define the desired total number of attributes for the structured UDT.

Dropping Attributes From a Structured UDT

Vantage automatically removes the associated observer and mutator methods for each attribute that you drop.

The following rules apply to dropping attributes.

- The specified UDT must be a structured type.
- The specified UDT must not be referenced by any other database object.

An attempt to drop an attribute from a UDT being referenced by some other database object, such as being used as the column data type of any table, the system returns an error to the requestor.

To be explicit, any of the following conditions cause the request to abort.

- There is a table in any database for which the data type of one of the columns is the specified UDT.
- There is a structured UDT for which the data type of one of the attributes is the specified UDT.
- The specified UDT is referenced in a user-defined cast.
- There is a method or UDF in any database that references the specified UDT.
- There is an ordering or transform group defined for the UDT.
- The attribute to be dropped must not be the only attribute of the UDT.

As a UDT developer, you are responsible for performing all the necessary cleanup on all of the following database objects *before* you drop any attributes from a structured UDT.

- Casts associated with the UDT.
- Orderings for the UDT.
- Transforms for the UDT.
- Tables whose columns are typed with the UDT.
- Structured UDTs that use the UDT.
- UDFs and methods that use the UDT as a parameter type.

You can drop attributes from a structured UDT definition even if any of the following database objects references the observer or mutator methods associated with that attribute.

- Cast
- Macro
- Procedure
- Trigger
- View

However, when you attempt to execute any of the affected database objects, the system returns an error to the requestor.

For more information, see:

- [CREATE TABLE](#)
- [CREATE CAST and REPLACE CAST](#)
- [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#)
- [CREATE METHOD](#)
- [CREATE ORDERING and REPLACE ORDERING](#)
- [CREATE TRANSFORM and REPLACE TRANSFORM](#)
- [CREATE TYPE \(Structured Form\)](#)

Adding Methods To a UDT

This clause adds a method signature to the specified UDT. You cannot invoke the new method until its body is specified completely with an appropriate CREATE METHOD request (see [CREATE METHOD](#)).

A new method cannot be a duplicate of any existing methods associated with the specified UDT.

| IF the UDT is this type ... | THEN the new method associated with it must be specified as a ... |
|-----------------------------|---|
| distinct | distinct type method. |
| structured | structured type method. |

The ability to add multiple methods using the ADD METHOD option is a Teradata extension to the ANSI SQL:2011 standard. You have the option of either submitting ALTER TYPE requests that comply with the ANSI SQL standard by adding only one method at a time, or of adding multiple methods simultaneously using the Teradata extension to the ANSI SQL:2011 syntax.

Dropping Methods From a UDT

This clause is used to drop a method signature associated with the specified UDT from the definition of that UDT. This is the only way you can drop a method. There is no DROP METHOD statement.

If you do not specify a method type, the default type is INSTANCE.

You cannot drop observer or mutator methods from a UDT definition.

When you drop a method signature from a UDT definition, the system also destroys its associated external routines.

The method to be dropped must not be referenced by any of the following database definitions.

- A cast.
See [CREATE CAST and REPLACE CAST](#).
- An ordering.
See [CREATE ORDERING and REPLACE ORDERING](#).
- A transform set.
See [CREATE TRANSFORM and REPLACE TRANSFORM](#).

You can drop methods from a UDT definition even if any of the following database objects references it.

- Cast
- Macro
- Procedure
- Trigger
- View

However, when you attempt to execute any of the affected database objects, the system returns an error to the requestor.

The ANSI SQL standard does not permit you to drop methods from a distinct UDT definition; however, you can drop methods from distinct UDTs as a Teradata extension to the ANSI SQL:2011 standard.

To replace the external routine for a method, use the REPLACE METHOD statement (see [REPLACE METHOD](#) and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

Altering Attributes Of A Structured UDT Definition

If you attempt to add or drop an attribute from the specified structured UDT, the system aborts the request if any other database object references that UDT.

This action protects the integrity of the database from corruption caused by objects depending on the previous definition of a structured UDT that is no longer correct.

You, as a UDT developer, are responsible for performing all the necessary clean up on all of the following database objects before you drop any attributes from a structured UDT.

- Casts associated with the UDT.
See [CREATE CAST and REPLACE CAST](#).
- Orderings for the UDT.
See [CREATE ORDERING and REPLACE ORDERING](#).
- Transforms for the UDT.
See [CREATE TRANSFORM and REPLACE TRANSFORM](#).
- Tables whose columns are typed with the UDT.
See [CREATE TABLE](#).
- Structured UDTs that use the UDT.
See [CREATE TYPE \(Structured Form\)](#).
- UDFs and methods that use the UDT as a parameter type.
See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#) and [CREATE METHOD](#).

Altering The Method Signatures Of A Structured UDT Definition

If you attempt to add or drop a method signature from the specified UDT, the system aborts the request if any cast, ordering, or transform set definition in the system references the method (see [CREATE CAST and REPLACE CAST](#), [CREATE ORDERING and REPLACE ORDERING](#), and [CREATE TRANSFORM and REPLACE TRANSFORM](#)).

This action protects the integrity of the database from corruption caused by objects depending on the previous definition of a structured UDT that is no longer correct.

Recompiling a UDT

It might become necessary to recompile one or all of your UDTs. An example might be an upgrade or migration to a newer release.

Recompiling a UDT is a simple operation. All you must do is submit an ALTER TYPE request that does the following.

- Specifies the name of the UDT to be recompiled.
- Specifies the COMPILE option.

A successful ALTER TYPE ... COMPILE request does the following things.

- Recompiles the code source for the specified type.
- Generates the new object code.
- Recreates the appropriate .so file.
- Distributes the recreated .so file to all nodes of the system.

Related Information

The following topics are all related to UDTs.

- [CREATE TYPE \(Distinct Form\)](#)
- [CREATE TYPE \(Structured Form\)](#)
- [HELP TYPE](#)
- DROP TYPE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

BEGIN QUERY CAPTURE- COMMENT

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

BEGIN QUERY CAPTURE

Rules and Restrictions for BEGIN QUERY CAPTURE

The following rules and restrictions apply to the BEGIN QUERY CAPTURE statement.

BEGIN QUERY CAPTURE and BEGIN QUERY LOGGING

BEGIN QUERY CAPTURE has the same functionality as the following form of BEGIN QUERY LOGGING, except that the XML query plans are captured in a query capture database (QCD) instead of DBQL logs.

```
BEGIN QUERY LOGGING WITH XMLPLAN,STATSUSAGE,VERBOSE,STATSDetails;
```

The database does not capture the XML plans in DBQL logs, ensuring that NO EXEC queries are not mixed with executed queries in the DBQL logs. Instead, the system captures the XML plans in the QCD table XMLQCD.

The scope of BEGIN QUERY CAPTURE is at the session level, not at the user level like BEGIN QUERY LOGGING. Unlike DBC.DbqlRuleTbl, which is maintained for query logging, the database does not maintain query capture rules in a query capture database. For more information about the relationship between BEGIN QUERY CAPTURE and BEGIN QUERY LOGGING, see *Teradata Vantage™ - Database Administration*, B035-1093.

BEGIN QUERY CAPTURE supports the same statement types as BEGIN QUERY LOGGING.

Note:

BEGIN QUERY CAPTURE does not capture query plans for the EXPLAIN request modifier or for the INSERT EXPLAIN and DUMP EXPLAIN statements.

You cannot submit a BEGIN QUERY LOGGING request for a user that has an active BEGIN QUERY CAPTURE session.

Query Banding and BEGIN QUERY CAPTURE

You can use query banding with BEGIN QUERY CAPTURE statements, but you cannot enable or disable query banding within the scope of a BEGIN QUERY CAPTURE and END QUERY CAPTURE statement pair.

The database captures the name of the query band used and stores it in the QCD XMLQCD table.

VERBOSE Option for BEGIN QUERY CAPTURE

When you specify the VERBOSE option for BEGIN QUERY CAPTURE, the logged XML document contains details on Object, SQL, Step Details, and Verbose EXPLAIN text regardless of whether you specified the XMLPLAN option.

The database logs the Verbose EXPLAIN text in the existing <PlanStep>@StepText attribute.

Note:

Verbose EXPLAIN text always includes the normal EXPLAIN text for a request.

STATSDetails and XMLPLAN Options

When you specify the STATSDetails option but not the XMLPLAN option, the logged XML document contains details on Object, SQL, Step Details, and Statistics Details from Statistics header and Statistics Recommendation DDLs.

When you specify both STATSDetails and XMLPLAN, the database logs a single integrated document containing data from both options.

The database logs Object, SQL, and Step Details only once. Statistics Details from the Statistics Header are logged into a new XML element.

BEGIN QUERY CAPTURE Statement Scope

You cannot submit a BEGIN QUERY CAPTURE statement within the scope of a previously submitted BEGIN QUERY CAPTURE statement.

BEGIN QUERY LOGGING

Uses for Database Query Logging

Database query logging provides DBAs with a set of measures that can be analyzed over a period of time to verify system usage or to more closely examine a query that uses an unexpectedly large amount of system resources.

Several tools support these analyses, including several System FE views and macros, diagnostics, and Database Query Analysis tools.

The Teradata Viewpoint Stats Manager portlet enables you to manage statistics collection, which includes the ability to collect and analyze statistics, create and control jobs, and manage recommendations.

You can use the logged information with other system tables to maximize system throughput.

You can use many of the SQL statements described here to collect and analyze this data.

For detailed information about analytical facilities, see the following documents:

- *Teradata Vantage™ - Application Programming Reference*, B035-1090
- *Teradata Vantage™ - SystemFE Macros*, B035-1103
- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100
- *Teradata® Viewpoint User Guide*, B035-2206

Logging Options for BEGIN QUERY LOGGING

Logging options that you set with BEGIN QUERY LOGGING are recorded in the DBQLOptions field of DBQLRuleTbl.

| Logging Option | Mask Value | Indicator Bit Position |
|----------------|------------|------------------------|
| EXPLAIN | 2 | 2 |
| SQL | 4 | 3 |
| OBJECTS | 8 | 4 |
| STEPINFO | 16 | 5 |
| XMLPLAN | 32 | 6 |
| SUMMARY | 64 | 7 |
| THRESHOLD | 128 | 8 |
| STATSUSAGE | 256 | 9 |
| LOCK | 512 | 10 |
| VERBOSE | 1024 | 11 |

| Logging Option | Mask Value | Indicator Bit Position |
|---------------------|------------|------------------------|
| DETAILED STATSUSAGE | 2048 | 12 |
| USECOUNT | 4096 | 13 |
| UTILITYINFO | 8192 | 14 |
| PARAMINFO | 16384 | 15 |

| Option | Definition |
|----------------|---|
| ALL | <p>Log query object information, step information, EXPLAIN text, and all SQL requests for all logged on users. This is equivalent to specifying the EXPLAIN, OBJECTS, SQL, and STEPINFO options individually.</p> <p>The ALL option does not include LOCK, PARAMINFO, STATSUSAGE, XMLPLAN, or UTILITYINFO.</p> <p>This option does not invoke the SUMMARY or THRESHOLD limit options.</p> <p>You cannot specify the ALL logging option if you also specify the THRESHOLD limit option.</p> <p>If you specify ALL, then you should also specify a value of 0 for SQLTEXT to ensure that the text for the SQL request is not logged in both the DBC.DBQLogTbl and DBC.DBQLSqlTbl tables.</p> <p>If you specify ALL, then you cannot specify other logging options.</p> |
| EXPLAIN | <p>Log the unformatted EXPLAIN text for the request.</p> <p>You cannot specify the EXPLAIN logging option if you also specify the THRESHOLD limit option.</p> <p>This option generates and logs the unformatted EXPLAIN text for each request. It does not generate EXPLAIN text for requests preceded by the EXPLAIN request modifier. For example, the system does not log EXPLAIN text for the following request.</p> <pre>EXPLAIN SELECT * FROM table_a ; /* EXPLAIN text logging does not occur */</pre> <p>If you perform the same query without the preceding EXPLAIN modifier, however, the system does log unformatted EXPLAIN text.</p> <pre>SELECT * FROM table_a; /* EXPLAIN text logging does occur */</pre> <p>Use care when specifying this option because of the performance cost of generating the EXPLAIN text for a query.</p> |
| LOCK= <i>n</i> | <p>Log any lock contention that exceeds <i>n</i> hundredths of a second in XML format in DBQLXMLLockTbl. For more information on DBQLXMLLockTbl, including how to shred the lock plan data, see <i>Teradata Vantage™ - Database Administration</i>, B035-1093.</p> |
| OBJECTS | <p>Log information about databases, tables, columns, and indexes accessed by SQL requests for the specified user set.</p> |

| Option | Definition |
|-------------|--|
| | <p>You can specify the OBJECTS logging option if you also specify the THRESHOLD limit option.</p> <p>This option does not log activity for macros or views, or log activity for dictionary tables and columns. For example, CREATE TABLE does not log any objects in the DBQL system because the statement only generates dictionary objects. For example, a query like the following does not log any objects because the query only accesses dictionary tables.</p> <pre>SELECT * FROM DBC.TVFields;</pre> |
| PARAMINFO | Parameter values and metadata are logged in DBQLParamTbl. |
| SQL | <p>Log the full text of all SQL requests performed by the specified user set in system table DBC.DBQLSqlTbl.</p> <p>You can specify the SQL logging option if you also specify the THRESHOLD limit option. This option does not log SQL requests within SQL procedures, macros, views, or triggers.</p> |
| STATSUSAGE | <p>Log DML requests that require cardinality estimation in their query optimization as an XML document in system table DBC.DBQLXMLTbl.</p> <p>If you specify STATSUSAGE with XMLPLAN, the database logs the collected data in a single integrated document.</p> <p>Data logged with this option is independent of the THRESHOLD or SUMMARY option limit options.</p> <p>STATSUSAGE does not log activity for explained requests or for the DUMP EXPLAIN and INSERT EXPLAIN statements. For information about these statements and the EXPLAIN request modifier, see <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146.</p> <p>The ALL option does not include STATSUSAGE.</p> |
| STEPINFO | <p>Log AMP step-level information for all SQL requests performed by the specified user set in system table DBC.DBQLStepTbl.</p> <p>You can specify the STEPINFO logging option if you also specify the THRESHOLD limit option.</p> |
| USECOUNT | <p>Log use count collection for a database or user.</p> <p>If you specify USECOUNT for a user, you can specify any of the other logging options. If you specify USECOUNT for a database, you cannot specify any other logging options. Otherwise, the system returns an error to the requestor.</p> <p>When you submit a BEGIN QUERY LOGGING request, the use counts and timestamps for the database or user are reset.</p> |
| UTILITYINFO | Utility information is logged in DBC.DBQLUtilityTbl. |
| XMLPLAN | <p>Logs the query plan generated by the Optimizer for SQL DML requests as an XML document in system table DBC.DBQLXMLTbl.</p> <p>When logging problematic queries, the recommended best practice is to specify XMLPLAN logging. The XMLPLAN option captures the query plan with additional detail to assist diagnosing performance issues.</p> <p>If you specify XMLPLAN with STATSUSAGE, the database logs the collected data in a single integrated document.</p> |

| Option | Definition |
|--------|---|
| | <p>Logs basic information, such as StatementType and the corresponding StepNames, for DDL statements. In addition, detailed information is logged for:</p> <ul style="list-style-type: none"> • COLLECT STATISTICS • CREATE INDEX • CREATE TABLE • DROP INDEX • DROP TABLE <p>XMLPLAN also logs detailed information for FastLoad and MultiLoad jobs.</p> <p>Because the XML document includes the query and step text, you usually do not need to specify the EXPLAIN and SQL options if you specify XMLPLAN. You should also specify a value of 0 for SQLTEXT to avoid redundant logging when you specify XMLPLAN.</p> <p>The XMLPLAN option does not log query plans for EXPLAIN request modifiers, or for INSERT EXPLAIN and DUMP EXPLAIN requests.</p> <p>The ALL option does not include XMLPLAN.</p> <p>You cannot specify the XMLPLAN logging option if you also specify the THRESHOLD limit option.</p> |

Limit Options for BEGIN QUERY LOGGING

The following table documents the SQLTEXT option for BEGIN QUERY LOGGING.

| Option | Definition |
|----------------------|---|
| SQLTEXT[= <i>n</i>] | <p>Sets the maximum number of SQL text characters to log in the default row. The default value is 200.</p> <p>The value specified cannot be negative. If you specify 0, then no characters are logged.</p> <p>If you specify SQLTEXT without specifying a numeric value to limit the number of characters logged, then the entire SQL request is logged up to a maximum of 10,000 characters in <i>DBC.DBQLogTbl</i>. If the request exceeds 10,000 characters, then the excessive characters are not logged.</p> <p>If you specify either ALL or SQL, SQLTEXT is logged redundantly in both <i>DBC.DBQLogTbl</i> and <i>DBC.DBQLSqlTbl</i>.</p> <p>You should not specify both ALL and SQLTEXT. If you specify ALL, then you should set the value for SQLTEXT to 0. Otherwise, the SQL text is redundantly logged in both the <i>DBC.DBQLogTbl</i> and <i>DBC.DBQLSqlTbl</i> tables.</p> |

The following tables document the SUMMARY option for BEGIN QUERY LOGGING.

| Option | Definition |
|---|---|
| SUMMARY= <i>n1</i> , <i>n2</i> , <i>n3</i> | <p>Designed for use with short, OLTP-like, queries.</p> <p>You can specify SUMMARY to reduce collection overhead for short running, high-volume queries. You can further specify the units of ELAPSEDTIME, CPUUSAGE, or IOCOUNT.</p> <p>Counts the number of requests for a session that fall into each of four time intervals or count ranges. Interval values can be specified in CPU time, normalized CPU time, elapsed seconds, elapsed hundredths of seconds, or I/O counts.</p> |

| Option | Definition |
|--------|---|
| | <p>If you do not specify a summary unit for the intervals, the default is expressed in elapsed seconds.</p> <p>If you specify SUMMARY, then you cannot specify any other options.</p> <p>You must specify the first three intervals explicitly. The fourth interval is created by default.</p> <ul style="list-style-type: none"> • The range for the first interval is from 0 to n_1 seconds. • The range for the second interval is from n_1 to n_2 seconds. • The range for the third interval is from n_2 to n_3 seconds. • The range for the fourth interval is $> n_3$ seconds. <p>Counts for each interval are stored in the <i>QueryCount</i> column of four separate rows in <i>DBC.DBQLSummaryTbl</i>.</p> <p>SUMMARY is the only option whose cache is flushed at regular, system-defined, intervals.</p> |

The following table documents the valid SUMMARY option modifiers.

| SUMMARY Option Modifier | Description |
|----------------------------|---|
| CPUTIME | <p>Use this option to set ranges and to summarize counts of the number of requests that fall into a CPU time interval.</p> <p>The SUMMARY value is expressed in units of 0.01 second.</p> <p>For example, if you specify 500 for one of the intervals, then the value used to make the determination is 5 CPU seconds.</p> |
| CPUTIMENORM | <p>Use this option to set ranges and to summarize counts of the number of requests that fall into a normalized CPU time interval.</p> <p>This option is designed for use with coexistence systems to aid in managing mixed nodes more efficiently, but it can be used with any system.</p> <p>The SUMMARY value is expressed in units of 0.01 second.</p> |
| <u>ELAPSEDSEC</u> | <p>Use this option to set ranges and to summarize counts of the number of requests that fall into an elapsed time interval.</p> <p>The SUMMARY value is expressed in units of 1.00 seconds.</p> <p>This is the default.</p> |
| ELAPSEDTIME | <p>Use this option to set ranges and to summarize counts of the number of requests that fall into an elapsed time interval.</p> <p>The SUMMARY value is expressed in units of 0.01 second, so it provides finer granularity for elapsed time than ELAPSEDSEC.</p> |
| IOCOUNT | <p>Use this option to set ranges and to summarize counts of the number of requests that fall into an I/O interval.</p> |

The following tables document the THRESHOLD option for BEGIN QUERY LOGGING.

| Option | Definition |
|------------------------|--|
| THRESHOLD[= <i>n</i>] | <p>Sets a threshold (the optional value <i>n</i> specifies the number of units of either seconds or I/O counts for the threshold) that determines whether a query is to be logged fully or just counted. Queries that exceed the threshold are logged, while those that do not meet the threshold are not logged.</p> <p>You can specify THRESHOLD to reduce collection overhead for short running, high-volume queries. You can further specify the units of ELAPSEDTIME, CPUUSAGE, or IOCOUNT.</p> <p>You can specify the THRESHOLD limit option with the SQL, STEPINFO, and OBJECTS logging options.</p> <p>You <i>cannot</i> specify the THRESHOLD limit option with the ALL, EXPLAIN, STATSUSAGE, or XMLPLAN logging options.</p> <p>If you do not specify a threshold option modifier to specify the logging units, the default is expressed in elapsed seconds.</p> <p>If you do not specify a THRESHOLD limit or SUMMARY, then all queries are logged fully.</p> <p>The following restrictions apply to time thresholds only.</p> <ul style="list-style-type: none"> • If a query completes earlier than or equal to the defined threshold value, then it is only logged as a count in <i>DBC.DBQLSummaryTbl</i>. The Threshold row in <i>DBC.DBQLSummaryTbl</i> is identified by a <i>HighHist</i> column value of 0. • If a query completes later than the defined threshold value, then a full entry is logged for it in <i>DBC.DBQLLogTbl</i> with values for all columns of the row, as would information for the SQL, STEPINFO, and OBJECTS options if you specify them. • If you specify THRESHOLD without also specifying a value for <i>n</i>, then the value 5 seconds is assigned by default. • The maximum value for <i>n</i> is 4,294,967,295 seconds (approximately 1,193,046 hours). <p>The following restriction applies to I/O count thresholds only.</p> <ul style="list-style-type: none"> • The maximum value for <i>n</i> is 32,767 I/O counts. <p>See the list of valid THRESHOLD option modifiers and their descriptions on the next page.</p> |

The following table lists the valid THRESHOLD option modifiers.

| THRESHOLD Option Modifier | Description |
|------------------------------|---|
| CPUTIME | <p>If you do not specify a threshold value for <i>n</i>, then the system uses the default CPUTIME value of 0.05 CPU seconds.</p> <p>The THRESHOLD value is expressed in units of 0.01 second.</p> <p>For example, if you specify 500, then the value used to make the determination is 5 CPU seconds.</p> |
| CPUTIMENORM | <p>This option is designed for use with coexistence systems to aid in managing mixed nodes more efficiently.</p> <p>The THRESHOLD value is expressed in units of 0.01 second.</p> |
| <u>ELAPSEDTIME</u> | <p>The THRESHOLD value is expressed in units of 1.00 seconds.</p> <p>This is the default.</p> |

| THRESHOLD Option Modifier | Description |
|------------------------------|--|
| ELAPSEDTIME | The THRESHOLD value is expressed in units of 0.01 second, so it provides finer granularity for elapsed time than ELAPSEDSEC. |
| IOCOUNT | If you do not specify a THRESHOLD value for <i>n</i> , then the system uses the default IOCOUNT value of 5. |

Flushing the Database Query Log

You can specify the rate at which the system flushes entries in DBQL using the DBQLFlushRate parameter of the DBS Control record. Valid rates range between 1 and 3600 seconds. The default, which is also the recommended flush rate, is to flush the query log every 600 seconds. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Flushing DBQL Rules From the Cache

Vantage flushes the DBQL rules cache at the same rate as the Plastic Steps cache. You cannot change this flush interval.

Dealing With Session Account String Changes

If you change the account string for a session using a SET SESSION ACCOUNT request, DBQL automatically ensures that any rules associated with an affected user and account string are applied when the next query is issued from that session.

Hierarchy of Applying Database Query Logging Rules

Database Query Logging works from a hierarchical foundation that allows BEGIN QUERY LOGGING requests to be submitted for individual users even if a rule exists for ALL users. However, if a rule exists for a specific account:user pair, you must submit an appropriate END QUERY LOGGING request to delete the rule before you can issue a new rule for that account:user pair.

The database applies the rules in the following order.

| Order in Hierarchy | Type of Rule |
|--------------------|---|
| 1 | A rule based on an application name. |
| 2 | A rule for this specific user and specific account. |
| 3 | A rule for this specific user and any account. |
| 4 | A rule for all users and this specific account. |
| 5 | A rule for all users and any account. |

As you can see from this table, DBQL first searches for a rule based on an application name. If no such rule exists, DBQL then looks for a rule specific to the user and account, and so on down the hierarchy. You can submit a `SHOW QUERY LOGGING` request to determine which rules the database applies. See [SHOW QUERY LOGGING](#).

The rules cache contains rules either for an application or for a specific account:user combination. As each user logs on, DBQL first searches the rules cache in hierarchical order for a rule. If there are no specific rules in the rules cache for level 1 or 2, DBQL searches `DBC.DBQLRuleTbl` in hierarchical order for the best fit. DBQL makes an entry in the rules cache for the account:user pair: either a rule that DBQL is not enabled for the account:user or the DBQL rule that applies with its options. If a match is made on the rules table at level 1, DBQL makes an application name entry in the rules cache.

The hierarchical scheme permits you to invoke a DBQL rule for a specific account:user pair, and a different rule for that specific user for all other accounts. Similarly, you might establish a rule for all users with a specific account, and a different rule for all other users, such as an ALL users ALL accounts rule.

For example, you can submit a `BEGIN QUERY LOGGING` request for default logging on ALL users, and DBQL can also be enabled for *user1* with objects and steps. If *user1* logs on, DBQL collects objects and steps. When users other than *user1* log on, DBQL only logs default information for them.

Similarly, if there is an ALL users rule to log information for objects, you can implement a rule for a specific user so that DBQL logs does default logging for that user.

You can also selectively exclude users, applications, and so on from logging through the use of the `WITH NONE` option. For example, if there is a rule that specifies default logging for ALL users, and you want to exclude logging for *busy_user1*, you can issue the following request.

```
BEGIN QUERY LOGGING WITH NONE ON busy_user1;
```

Once a set of query logging rules is created, it applies to the current session and all subsequent sessions to which an active DBQL rule applies.

Note that DBQL does not allow multiple rules for any named instance of one of the five query logging rule types. For example, if a rule currently exists for ALL users under a specific account name, the system does not accept a duplicate rule.

Application Names

The application names you specify with the `APPLNAME` option are the names the system passes in the reserved query band `UtilityName`.

The following table lists the `UtilityName` strings.

| Client/DBS Protocol | Utility Type | Utility Name Value |
|---------------------|---|--------------------|
| FastExport | Standalone FastExport | FASTEXP |
| | Teradata Parallel Transporter EXPORT operator | TPTEXP |
| | JDBC FastExport | JDBCE |

| Client/DBS Protocol | Utility Type | Utility Name Value |
|---------------------|---|--------------------|
| | .NET FastExport | DOTNETE |
| FastLoad | Standalone FastLoad | FASTLOAD |
| | Teradata Parallel Transporter LOAD operator | TPTLOAD |
| | JDBC FastLoad | JDBCL |
| | .NET FastLoad | DOTNETL |
| | Crashdumps Save Program (CSP) Save Dump | CSPLOAD |
| MultiLoad | Standalone MultiLoad | MULTLOAD |
| | Teradata Parallel Transporter UPDATE operator | TPTUPD |
| | JDBC MultiLoad | JDBCM |
| | .NET MultiLoad | DOTNETM |

Not All SQL Request Text Is Logged By The Query Logging Feature

The text of SQL requests that are performed from within the following features are not logged by query logging.

- Macros
- Triggers
- Views

In addition to this SQL request text, DCL requests are also not logged. Furthermore, query logging does not log rows for cached requests for the EXPLAIN option.

Limits of Dynamically Enabled Query Logging

Query logging can be begun or ended dynamically for a maximum of 100 users per BEGIN QUERY LOGGING request. When more than 100 users are to be logged, you must submit a second BEGIN QUERY LOGGING request. For active users, query logging begins as soon as they perform their next SQL request.

Query Logging and SQL Transactions

You can only initiate query logging in Teradata session mode if you make the request outside the boundaries of an explicit (BT/ET) transaction.

You cannot perform BEGIN QUERY LOGGING in Teradata session mode within an explicit transaction.

You cannot perform BEGIN QUERY LOGGING in ANSI session mode.

Minimum Level of Query Logging

When query logging is enabled at the default level, at least one row of query-level information is logged for each query performed by each user specified by the initiating BEGIN QUERY LOGGING statement.

Database Query Log Tables and Views

The tables and views that log SQL query data are in the data dictionary. For definitions of the query log tables, see *Teradata Vantage™ - Data Dictionary*, B035-1092. The following table provides a summary of log tables for tables and views.

| Object Type | Dictionary Object Name | Description |
|-------------|------------------------|---|
| Table | DBC.DBQLExplainTbl | Stores the unformatted EXPLAIN text for a query. |
| | DBC.DBQLObjTbl | Stores query object information. |
| | DBC.DBQLogTbl | Stores default rows for the Database Query Log, including Query Band information. |
| | DBC.DBQLRuleCountTbl | Stores the cardinality of DBQLRuleTbl. |
| | DBC.DBQLRuleTbl | Stores the rules for the Database Query Log. |
| | DBC.DBQLSqlTbl | Stores SQL query text. |
| | DBC.DBQLStepTbl | Stores query step information. |
| | DBC.DBQLSummaryTbl | Stores summary and threshold information for the logged query. |
| | DBC.DBQLXMLLockTbl | Logs lock contentions that exceed threshold. |
| | DBC.DBQLXMLTbl | Stores the Optimizer query plan for the logged query as an XML document. |
| View | DBC.DBQLRulesV | A view of DBC.DBQLRuleTbl. |
| | DBC.QryLockLogXMLV | A view of DBC.DBQLXMLLockTbl. |
| | DBC.QryLogExplainV | A view of DBQLExplainTbl. |
| | DBC.QryLogObjectsV | A view of DBQLObjTbl. |
| | DBC.QryLogSQLV | A view of DBQLSQLTbl. |
| | DBC.QryLogStepsV | A view of DBQLStepTbl. |
| | DBC.QryLogSummaryV | A view of DBQLSummaryTbl. |
| | DBC.QryLogTDWMV | A view of DBQLogTbl. |
| | DBC.QryLogV | A view of DBC.DBQLogTbl. |
| | DBC.QryLogXMLV | A view of DBQLXMLTbl. |

Query Logging User Rules

Because you can log different information for different users, the system table DBC.DBQLRuleTbl maintains the various logging rules for each user. These rules are based on the logon account string for each logged user and are maintained in cache to promote their quick access.

This topic does not describe the rules hierarchy, but can be helpful for understanding default accounts and DBQL rules. See [Hierarchy of Applying Database Query Logging Rules](#).

You must access the DBC.DBQLRulesV view. You cannot access either DBC.DBQLRuleTbl or DBC.DBQLRuleCountTbl directly.

Sessions are always associated with an account. At logon, the session is associated with the default account of the user who is logging on unless the logon string for that user specifies a different account assignment. See the logon pointer information in *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418.

DBQL searches for rules in the following order:

- 1. Local cache.
- 2. Rules database.

Assume that *user1* has a default account of *abc*, but is also assigned to account *def* and that you have submitted the following BEGIN QUERY LOGGING requests:

```
BEGIN QUERY LOGGING ON ALL ACCOUNT = 'ABC';

BEGIN QUERY LOGGING WITH OBJECTS ON user1 ACCOUNT = 'def';
```

In this scenario, if a user does not specify an account string when logging on, then the system applies the ALL rule instead of the *user1* rule.

| IF user1 logs on ... | THEN the following rule applies ... |
|-------------------------------|---|
| specifying account def | <div>user1. The system uses the rule specified by the following request to log the user1 queries and to log objects for those queries.</div> <div>BEGIN QUERY LOGGING WITH OBJECTS ON user1 ACCOUNT = 'def ';</div> |
| without specifying an account | <div>ALL. The system uses the rule specified by the following request to log the <i>user1</i> queries, but does <i>not</i> log object information about them.</div> |

| IF user1 logs on ... | THEN the following rule applies ... |
|----------------------|---|
| | <pre>BEGIN QUERY LOGGING ON ALL ACCOUNT = 'abc ';</pre> |

Using BEGIN QUERY LOGGING to Log Query Plan Information

You can capture the Optimizer query plan for logged SQL DML requests in the DBQL table DBC.DBQLXMLTbl. To do this, specify XMLPLAN for *logging_option* to enable query plan capture for executing DML queries as XML text.

Query plan information in XML format is particularly useful for diagnosing performance and query plan issues.

You cannot capture query plans for the following statements and request modifier with the XMLPLAN option:

- DUMP EXPLAIN
- INSERT EXPLAIN
- EXPLAIN

Also, the database does not log text definitions of objects referenced by SQL requests. Instead, the system logs a short identifier that can be used as a key to fetch the text definition from the data dictionary if it is needed.

XMLPLAN logging is not an alternative to the information captured by INSERT EXPLAIN requests because:

- Apart from the respective content of the documents they produce, there is another important difference between XMLPLAN logging and INSERT EXPLAIN requests, as indicated by the following table.

| BEGIN QUERY LOGGING ... XMLPLAN | INSERT EXPLAIN |
|--|---|
| Logs query plans for executed queries in XML format. | Captures query plans without executing the query. |

XMLPLAN logging is ideal when you want to record query plans for your executing workloads and have found that capturing query plans for the requests in those workloads using INSERT EXPLAIN requests is too slow for your needs.

On the other hand, if you are only *tuning* a query and do not want to execute it, XMLPLAN logging is not as useful as capturing the query plan for a request using INSERT EXPLAIN requests.

In this case, executing an INSERT EXPLAIN INTO *QCD_name* IN XML request or an EXPLAIN IN XML *SQL_request* is a more analogous alternative. You cannot capture query plans in XML format using DUMP EXPLAIN requests.

- Runtime information from the traditional DBQL tables is also captured for a logged plan.
- XMLPLAN logging is more an extension to query logging than an extension to the Query Capture Facility.

The EXPLAIN option does not log rows for cached requests.

Using SUMMARY Query Logging for Tactical Queries

The two query logging tables that are most relevant to tactical queries are the default logging table, *DBC.DBQLLogTbl*, and the summary logging table, *DBC.DBQLSummaryTbl*.

You can enable query logging for one, a group, all users, or for an application, and you can only view the logged data after it has been written to disk. The summary table cache is written to disk every 10 minutes. Rows to the default and other DBQL tables are written when either of the following conditions occurs.

- The DBQL table caches become full.
- The threshold for DBQLFlushRate is reached.
- You issue an END QUERY LOGGING request.

Avoid enabling query logging for tactical queries at the default logging level beyond their testing period because the overhead can have an impact on the performance of single- or few-AMP queries.

You might want to enable default query logging during the test phase of your tactical queries to validate that your plans are being cached, because the *DBC.DBQLLogTbl* table carries a flag indicating whether the query used a cached plan or not. However, there is rarely any value in enabling default logging for the well-tuned, predictable tactical query application, and if your queries are single- or few-AMP operations, query logging incurs some slight overhead by writing one row to the log for each query run.

A more general purpose use of query logging for tactical queries is to evaluate and record query response time consistency. You can accomplish this by using summary logging, which writes rows only to *DBC.DBQLSummaryTbl*. When you perform summary logging of a time variable, you must define three time thresholds, expressed in units of seconds. You can request summary logging using either elapsed, CPU, or normalized CPU time. You can also request summary logging of I/O counts.

For example.

```
BEGIN QUERY LOGGING LIMIT SUMMARY = 1, 5, 10
ON cab;
```

For each session having a user included in this summary logging, counts are kept for each of four defined buckets. Buckets having a zero count for an interval are not logged. For example, if all queries for an interval fall into the same bucket, then only one bucket is logged for that session. These buckets, which contain counts of queries within that time range, are kept independently for each session, and summary rows carry a session ID to identify the session to which they belong.

These counter buckets are flushed to disk at 10-minute intervals. At the end of the first 10-minute interval, you can examine the buckets for that, and earlier, time intervals by querying the *DBC.DBQLSummaryTbl* table with a request similar to the following example.

```

SELECT procid, collecttimestamp, userid, acctstring, sessionid,
       qrycnt, qrysecs, lowhist, highhist
FROM dbqlsummarytbl
ORDER BY collecttimestamp, sessionid, lowhist;

```

This query produces the following report that illustrates some of the summary data collected for a single session.

| ProcID | CollectTimeStamp | UserID | AcctString | SessionID | QryCnt | QrySecs | LowHist | HighHist |
|--------|---------------------|----------|------------|-----------|--------|---------|---------|---------------|
| 16,383 | 2003-05-30 16:32:11 | 00000100 | ? | 1000 | 8 | 1 | 0 | 1 |
| 16,383 | 2003-05-30 16:32:11 | 00000100 | ? | 1000 | 2 | 98 | 10 | 4,294,967,295 |
| 16,383 | 2003-06-13 15:22:44 | 00000100 | ? | 1001 | 5 | 3 | 0 | 1 |
| 16,383 | 2003-06-13 15:22:44 | 00000100 | ? | 1001 | 2 | 26 | 10 | 4,294,967,295 |
| 16,383 | 2003-06-13 15:32:44 | 00000100 | ? | 1001 | 8 | 4 | 0 | 1 |
| 16,383 | 2003-06-13 15:32:44 | 00000100 | ? | 1001 | 2 | 6 | 1 | 5 |
| 16,383 | 2003-06-13 15:32:44 | 00000100 | ? | 1001 | 2 | 16 | 5 | 10 |
| 16,383 | 2003-06-13 15:32:44 | 00000100 | ? | 1001 | 1 | 17 | 10 | 4,294,967,295 |

For ease of analysis, these summary rows are ordered by each 10-minute interval and formatted with a blank line between collection intervals. Based on the value of *CollectTimeStamp*, you can see that the first grouping was collected several days prior to the others. Information remains in the query log tables until you delete it explicitly.

The primary index of the *DBC.DBQLSummaryTbl* table is the first two columns, chosen to expedite the write-to-disk time for the rows. Each 10-minute interval grouping is written efficiently in a batch using the same NUPI value.

The following list provides some descriptive information about *DBC.DBQLSummaryTbl* table information.

- The *LowHist* and *HighHist* columns define the range of that particular bucket. Only buckets with at least one query count for that interval are produced. The third grouping illustrated in the preceding report is the only grouping with query counts in all 4 interval buckets defined by the SUMMARY list of the BEGIN QUERY LOGGING request used to produce the report.

| Interval Number | Interval Time Range (seconds) |
|-----------------|-------------------------------|
| 1 | 0 - 1 |
| 2 | 1 - 5 |
| 3 | 5 - 10 |
| 4 | 10 - 4,294,967,295 |

The highest possible value for *HighHist* in *DBC.DBQLSummaryTbl* is 4,294,967,295 (2^{32}) seconds, which is approximately 1,193,046 hours.

- The *QueryCount* column (abbreviated in the example report as *QryCnt*) lists how many queries ran with a response time within the range of that interval. For example, in the first row of output, 8 queries ran in less than 1 second, and 2 queries ran in greater than 10 seconds.
- The *QuerySeconds* column (abbreviated in the example report as *QrySecs*) is not strictly the total execution time of all queries in that bucket. A query response time must be greater than 0.5 seconds to be accumulated into that total, and a query that ran in 0.5 or more seconds counts as 1 second. Fractions of a second are rounded up or down in the accumulation.
- The values for *UserID* are expressed with a BYTE data type because that is how the information appears to the system. One way to view the summary table with an easily recognizable user ID is to join to the DBC.databases2V view, as the query below illustrates. The same details are contained in the summary table as when the previous example query was run.

```
SELECT st.collecttimestamp, db.databasename, st.sessionID,
       st.querycount, st.queryseconds, st.lowhist, st.highhist
FROM DBC.databases2V AS db, DBC.dbqlsummarytbl AS st
WHERE db.databaseid = st.userid
ORDER BY collecttimestamp, lowhist;
```

| CollectTimeStamp | DBName | SessionID | QryCnt | QrySecs | LowHist | HighHist |
|---------------------|--------|-----------|--------|---------|---------|---------------|
| 2003-05-30 16:32:11 | CAB | 1000 | 8 | 1 | 0 | 1 |
| 2003-05-30 16:32:11 | CAB | 1000 | 2 | 98 | 10 | 4,294,967,295 |
| 2003-06-13 15:22:44 | CAB | 1001 | 5 | 3 | 0 | 1 |
| 2003-06-13 15:22:44 | CAB | 1001 | 2 | 26 | 10 | 4,294,967,295 |
| 2003-06-13 15:32:44 | CAB | 1001 | 8 | 4 | 0 | 1 |
| 2003-06-13 15:32:44 | CAB | 1001 | 2 | 6 | 1 | 5 |
| 2003-06-13 15:32:44 | CAB | 1001 | 2 | 16 | 5 | 10 |
| 2003-06-13 15:32:44 | CAB | 1001 | 1 | 17 | 10 | 4,294,967,295 |

Some additional considerations when using DBQL for tactical queries include the following issues.

- Query log tables are designed for efficient inserts into the log tables rather than for ease of querying. For that reason, accessing log tables with high row counts can be a slow process. Keeping the tables well-maintained aids viewing, space management, and ease of analyzing the log information.
- For users whose queries are primarily single- or few-AMP operations, enable logging on the default table, *DBC.DBQLLogTbl*, during testing, and then log those queries in production only when necessary.
- DBQL summary information is useful for evaluating the extent of response time anomalies for tactical applications.
- DBQL summary information is not suitable for establishing accurate query rates for short-running queries because of its subsecond rounding. Because of the 10-minute lag between logging and being able to view the data, summary logging is not likely to match the needs of real time performance alerting.
- An additional alternative to summary logging is threshold logging, in which only queries running longer than a specified threshold times (either elapsed time, CPU time, or normalized CPU time, all expressed in units of seconds) or for more than a specified number of I/O counts, are logged

in *DBC.DBQLogTbl*. All queries with an execution time below the threshold value are logged in *DBC.DBQLSummaryTbl*.

Related Information

The following topics and documents are all related to query logging.

- [END QUERY LOGGING](#)
- [REPLACE QUERY LOGGING](#)
- In *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144:
 - BEGIN QUERY LOGGING
 - END QUERY LOGGING
 - REPLACE QUERY LOGGING
- *Teradata Vantage™ - Database Administration*, B035-1093

COLLECT STATISTICS (Optimizer Form)

Why You Need To Collect Statistics

The purpose of collecting statistics is to compute statistical data that the Optimizer uses to optimize table access and join plans. The computed statistical data is retained in a synopsis data structure that is stored in the data dictionary for use during the optimizing phase of SQL request parsing.

Accurate demographics enable the Optimizer to determine the least costly access and join plans for a query. Accurate table demographic information is especially important when data to be accessed using a column or index might be skewed.

Naming Collected Statistics

When you collect statistics on an index or column set, you can optionally specify a name for them. A name is required if statistics are for other than column references.

You can later use the name for recollections, copies, transfers, help information, show information, and for dropping the collected statistics. The rules for naming statistics are the same as the rules for naming database objects, which are documented in *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

When you recollect statistics, if the column ordering is different, the Optimizer treats the statistics as new. By naming the statistics you collect and then using the names when you recollect statistics guarantees that the existing statistics are recollected instead of creating a new set of statistics.

Duplicate names are not allowed for table or constant expression statistics. You can only use COLUMN specification to recollect statistics using statistics names.

Keep Statistics Current

If a table or partition has been extensively modified since its statistics were last computed, residual statistics are likely to cause the Optimizer to generate poor access and join plans. A table or partition is considered extensively modified if more than 10 percent of the rows are added to or deleted from the table or, in the case of a row-partitioned table, more than 10 percent of the rows are added to or deleted from a partition. Therefore, statistics should be periodically recollected or dropped. For all partitioned tables, any refreshment operation should include the system-derived PARTITION column set. See [Collecting Statistics on the PARTITION Column and the Row Partitioning Column Set](#). Keeping statistics fresh is an important factor in producing accurate query plans even when your system enables derived statistics and extrapolation. For details, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

If you recollect PARTITION statistics any time a partition or table becomes populated after having been empty, or becomes empty after having been populated, the operation should be fairly fast.

Following is an example of how stale statistics can produce an inefficient query plan.

| Table | Cardinality | Statistics? | Remarks |
|-------|-------------|-------------|--|
| A | 1,000,000 | Y | Statistics are not current. They were collected during an earlier phase, when the cardinality of the table was only 1,000 rows. |
| B | 75,000 | N | The Optimizer collects a dynamic AMP statistical sample for this table. |

You submit a request that performs a product join between tables *A* and *B*, and one of the tables must be duplicated on all AMPs. The Optimizer chooses to redistribute the rows from table *A* because when it checks the dictionary for statistics, it sees that the cardinality of table *A* is only 1,000 rows, which is far fewer than the 75,000 rows of table *B* (estimated from a dynamic AMP sample). Because table *A* currently has 1,000,000 rows, a difference of three orders of magnitude from what the current, stale, statistics indicate, the Optimizer uses an incorrect assumption to redistribute the 1,000,000 rows of table *A* instead of the 75,000 rows of table *B*. As a result, the query runs much longer than it would have if the Optimizer had current statistics from which to create its query plan.

Even though the Optimizer can use derived statistics and extrapolation when it detects stale statistics, those methods base their calculations on the current statistics in the interval histogram for a column set, so their estimates can be more accurate because they are based on statistics that are more current than the statistics currently stored. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

The derived statistics framework uses statistics taken from a dynamic AMP sample if statistics have not been collected on a column set. For details, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142. The Optimizer also compares stored dynamic AMP cardinality estimates with current dynamic AMP cardinality estimates to extrapolate cardinalities to determine table growth.

You should consider statistics to be stale in the following cases.

- The cardinality of the table or, in the case of a partitioned table, of any of its partitions, has changed significantly, for example, by more than 10 percent. However, this percentage is just a recommendation and can vary for a particular table or system.
- The range of values for an index, column, or partition on which statistics have been collected has changed significantly, for example, by more than 10 percent. However, this percentage is just a recommendation and can vary for a particular table or system.

To make an informal assessment of whether the table or partition cardinality has changed significantly, you can use the following statements:

| Statement | Information Returned |
|---|---|
| <code>HELP STATISTICS table_name;</code> | <ul style="list-style-type: none"> • Number of unique values for each statistic on a table • Date and time statistics were last collected |
| <code>SELECT COUNT(*) FROM table_name;</code> | Cardinality of statistics on unique indexes for comparison with the report produced by <code>HELP CURRENT STATISTICS</code> |

| Statement | Information Returned |
|---|---|
| <pre>SELECT COUNT (DISTINCT column_name) FROM table_name;</pre> | Cardinality of statistics on nonunique columns for comparison with the report produced by HELP CURRENT STATISTICS |

HELP CURRENT STATISTICS shows the extrapolated unique value counts. SHOW CURRENT STATISTICS VALUES displays the extrapolated summary information for a column or index.

You can update previously collected statistics by performing a COLLECT STATISTICS request without specifying an explicit list of columns or indexes. See [Collecting Statistics When No COLUMN or INDEX Clause Is Specified](#). The system then automatically recollects statistics for all existing statistics. This operation requires a full-table scan, so it can take a significant amount of time. See [Collecting and Recollecting Statistics Is Often Time Consuming](#).

The efficacy of collected statistics varies with the types of access used on a table. If performance does not seem to be improved by the statistics you have collected, then the Optimizer is probably not using the column to access or join the table. When you observe this behavior and EXPLAIN reports do not indicate that the Optimizer is using the column set as you thought it might, use the DROP STATISTICS statement to remove the statistics for that column from the data dictionary.

Perform a HELP STATISTICS request to determine which columns and indexes currently have statistics and to see a portion of the collected information or use SHOW STATISTICS VALUES to report complete information.

The Teradata Viewpoint Stats Manager portlet allows you to manage Vantage statistics collection, which includes the ability to collect and analyze statistics, create and control jobs, and manage recommendations. See *Teradata® Viewpoint User Guide*, B035-2206.

You can also run a query that reports the last time statistics were collected for each column. The following query lists database names, column names, and the date statistics were last collected.

```
SELECT DatabaseName
       ,TableName
       ,ColumnName
       ,CAST>LastCollectTimeStamp AS Date) As CollectionDate
       ,CAST>LastAlterTimeStamp AS Date) As LastAlter
       ,current_date - collectiondate AS FromCurrent
       ,lastalter - collectiondate AS FromAlter
FROM DBC.StatsV
WHERE ColumnName IS NOT NULL;
```

To limit the report, you can edit the query to specify a subset of columns.

PARTITION Statistics and ALTER TABLE TO CURRENT Requests

You should refresh PARTITION statistics as soon as an ALTER TABLE TO CURRENT request for a table or join index completes because the existing PARTITION statistics will no longer be valid after the reconciliation process.

Statistics on the partitioning column itself are not stale and are still valid if you do not specify a null partition handler, but you should always refresh the statistics on a partitioning column if you *do* specify a null partition handler.

Location of Stored Statistics

Table statistics collected by the Optimizer form of COLLECT STATISTICS are stored in the form of interval histograms in *DBC.StatsTbl*. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Collecting and Recollecting Statistics Is Often Time Consuming

Because Vantage always does a full-table scan (or an index scan if collecting statistics on an index) when it collects full statistics on a table (except when only PARTITION statistics are collected), the process can take a long time to complete. See [Reducing the Cost of Collecting Statistics by Sampling](#). The exception to this is collecting statistics on the system-derived column PARTITION columns, which is always a fast operation. The duration of the scan is dependent on the size of the table, the number of partitions, the system configuration, and the workload on the system.

Additional resources are consumed by the creation of interval histograms and the computation of the various statistical measures used to summarize the characteristics of a column set or index. For a definition of interval histograms, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Columns that are not indexed or that are unique indexes take more time to process than NUSI columns.

Because of the time it can take to collect and recollect statistics, you should consider specifying one or more of the USING options, particularly those that establish recollection thresholds, to establish rules for when Vantage should recollect statistics. For more information about these options, see [Comparison of Full and Sampled Statistics](#) and [Using the THRESHOLD Options to Collect and Recollect Statistics](#).

You should always collect statistics on newly created, empty tables. This defines the columns, indexes, and partitions for a partitioned table, as well as the synoptic data structures for subsequent collection of statistics and demographics. For a definition of synoptic data structure, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

You should also *recollect* statistics on newly created tables as soon as they are populated, and thereafter as needed.

When you perform a HELP STATISTICS request on an empty table, the Unique Value column returns a value of 0.

Reducing the Cost of Collecting Statistics by Sampling

You can specify a system-determined sample percentage to use a downgrade approach for determining when to switch to sampling. The Optimizer samples intelligently and adjusts the percentage of sampling to use, up to full statistics, while maintaining the quality of the collected statistics.

The downgrade approach works as follows.

1. When you initially submit a request to collect sampled statistics, the Optimizer collects full statistics. That is, the statistics are not sampled.

The Optimizer uses the full statistics to determine when to reduce sampling to a smaller percentage.

2. On subsequent requests to recollect statistics, the Optimizer collects full statistics until adequate statistics have been captured to provide a reliable historical record.
3. With the statistics history, the Optimizer can recognize, for example, whether the column data is skewed, the column is rolling or static, and so forth.

The Optimizer then considers the column usage data (detailed buckets or only summary data) that it maintains in *DBC.StatsTbl.UsageType* and the user-specified number of intervals to determine an appropriate time to reduce collection from full statistics to sampled statistics.

The database is more aggressive in reducing the percentage of sampled statistics for histograms whose summary data is frequently used, but whose detailed interval data is not.

4. The Optimizer determines an optimal sampling percentage to use for statistics recollections and determines the appropriate formula to use for scaling based on the history and nature of the column. For more information, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.
5. Finally, the Optimizer compares recollected statistics to its history data for quality.

The Optimizer lowers the sampling percentage only if a smaller sampling percentage provides quality results.

The automated sampling percentage reduction feature alleviates you from having to determine the columns for which sampled statistics can be collected and the columns for which full statistics should be collected. By collecting full statistics initially, this method provides the Optimizer with the necessary information to make an intelligent decision about how much to sample in subsequent recollections of statistics.

For small tables, skewed columns, columns included in a partitioning expression, and columns that require detailed histogram buckets, the Optimizer always collects full statistics.

The Optimizer can detect very non-singular columns using full statistics, and in subsequent recollections intelligently switch to sampled statistics at a significantly lower percentage than 100%. For example, the Optimizer might sample only 2% of table rows and then apply the appropriate scaling formula for nonunique columns to obtain the extrapolated full statistics, and obtain quality statistics much faster than by collecting full statistics.

For rolling, unique, and nearly-unique columns as detected in the full statistics collection, the Optimizer can subsequently collect sampled statistics at a much lower percentage and apply a linear scaling formula to generate extrapolated full statistics.

After a few collections of sampled statistics, depending on demographics patterns and change counts, the Optimizer can recollect full statistics to verify, adjusting the sample percentage and the extrapolation method used as necessary.

Comparison of Full and Sampled Statistics

The following table compares the two methods of collecting statistics using COLLECT STATISTICS (Optimizer Form) and lists the most productive uses.

The ability to specify thresholds for collecting and recollecting statistics minimizes these issues.

| Method | Characteristics | Best Use |
|-------------------------|--|---|
| Full statistics | <ul style="list-style-type: none"> Collects all statistics for all of the data. Time consuming. Most accurate of the three methods of collecting statistics. Stored in interval histograms in the data dictionary. | <ul style="list-style-type: none"> Best choice for columns or indexes with highly skewed data values. Recommended for small tables, where a small table is one with fewer than 1,000 rows per AMP. Recommended for selection columns having a moderate to low number of distinct values. Recommended for most NUSIs and other selection columns. Collection time on NUSIs is very fast. Recommended for all column sets or index column sets for the following cases. Where full statistics add value. Where sampling does not provide satisfactory statistical estimates. |
| User-sampled statistics | <ul style="list-style-type: none"> Collects all statistics for a sample of the data, not just cardinality estimates. Significantly faster collection time than full statistics. Stored in interval histograms in the data dictionary. | <ul style="list-style-type: none"> Acceptable for columns or indexes that are highly singular, that is, the number of distinct values approaches the cardinality of the table. Recommended for unique columns, unique indexes, and for columns or indexes that are highly singular. Sampled statistics are useful for very large tables, such as tables with tens of billions of rows. <i>Not</i> recommended for small tables, that is, tables whose cardinality is less than 20 times the number of AMPs in the system. |

Using the THRESHOLD Options to Collect and Recollect Statistics

Statistics collection thresholds enable you to minimize the unnecessary collection of statistics. For a system-determined threshold, the Optimizer automatically determines whether a recollection is needed or whether extrapolation is adequate. You can collect and recollect threshold statistics on tables.

The THRESHOLD options enable you to automatically skip recollecting statistics if the amount of data change since the last statistics collection or the age of the current statistics is below the thresholds in effect for the statistics.

The Optimizer can automatically determine the appropriate thresholds to apply based on previously collected statistics, column histories, change counts, and other factors, or you can explicitly specify the thresholds as part of a COLLECT STATISTICS request by specifying a change percentage, a number of days, or both.

You can submit collect statistics requests at regular intervals, and the Optimizer can use stored threshold and historical data to determine whether the specified recollection of statistics is necessary or not. The Optimizer can use the same data to determine when to recollect statistics for those columns that exceed the specified threshold. For example, if you determine that the threshold is to be a 10% data change and you submit a request to recollect statistics, the Optimizer does not recollect those statistics if the change in the data from the last collection is less than 10%.

If you specify a THRESHOLD option for first time statistics collections, the Optimizer collects the statistics and stores the THRESHOLD options you specified for future use without consulting the threshold values you specify. The Optimizer uses the saved collection thresholds to determine whether the current statistics collection can be skipped or not when you submit recollection requests at a later time.

Specifying a new THRESHOLD option for statistics recollection overrides the current setting, if any. The Optimizer applies the new THRESHOLD value to the current request, and remembers the updated THRESHOLD option for future statistics recollection.

You can specify the following kinds of thresholds to control recollecting statistics.

- Change-based
- Time-based

For a change-based threshold, the optimizer consults the system maintained UDI (Update, Delete, Insert) counts, random AMP samples and available table history to determine the amount of data change from the last statistics collection.

You can also specify a combination of change-based and time-based thresholds in a single COLLECT STATISTICS request.

For SYSTEM THRESHOLD or SYSTEM THRESHOLD PERCENT, the Optimizer considers column history and internal extrapolations techniques to determine the appropriate change threshold. When the Optimizer recollects full statistics for a small table NUSI, for example, it ignores threshold information and collects full statistics because, in that case, collecting full statistics provides more accurate statistics for a very small cost.

A time-based threshold uses the age of the current statistics to determine when statistics are refreshed.

Note:

For SYSTEM THRESHOLD DAYS, the system uses a change-based threshold for recollecting statistics.

USING Options

The COLLECT STATISTICS (Optimizer Form) USING options apply to various sampling and threshold options that you can use to more finely tune the row sampling you specify or to avoid recollecting statistics when the data has not changed enough according to recollection thresholds you specify. You cannot specify USING options if you also specify the SUMMARY option for a COLLECT STATISTICS request. For more information about sampled and threshold statistics, see [Reducing the Cost of Collecting Statistics by Sampling](#).

The following table describes the USING options.

| Option | Description |
|-------------------------|--|
| MAXINTERVALS <i>n</i> | <p>Maximum number of histogram intervals to be used for the collected statistics. Vantage might adjust the specified maximum number of intervals depending on the maximum histogram size.</p> <p>This option is valid for tables and constant expressions.</p> <p>The value for <i>n</i> must be an integer number.</p> <p>The valid range for <i>n</i> is 0 through 500.</p> <p>If you do not specify this option, Vantage determines the maximum number of intervals to use for the histogram.</p> <p>You can only specify this option if you also specify an explicit column or index.</p> <p>You cannot specify MAXINTERVALS <i>n</i> for a standard recollection of statistics on an implicitly specified index or column set.</p> <p>If you specify 0 intervals, the request only captures summary statistics such as the number of unique values, the number of nulls, and so on.</p> <p>The larger the number of intervals, the more optimal the granularity of the statistical data in the histogram. A finer granularity enables better single-table and join selectivity estimates for non-uniform data; however, you should apply this optimal granularity selectively because the larger the number of intervals, the larger the size of the histogram, which can increase query optimization time.</p> |
| MAXVALUELENGTH <i>n</i> | <p>Maximum size for histogram values such as MinValue, ModeValue, or MaxValue. The value for <i>n</i> must be an integer number.</p> <ul style="list-style-type: none"> For single-column statistics, the valid range of <i>n</i> is 1 through the maximum size of the column. For multicolumn statistics, the valid range of <i>n</i> is 1 through the combined maximum size of all the columns. <p>This option is valid for both tables and constant expressions.</p> <p>If you specify a larger size than the maximum size, Vantage automatically adjusts the value to the maximum size.</p> <p>Vantage might also adjust the maximum value length you specify based on the size of the histogram that contains the statistics.</p> |

| Option | Description |
|-------------------|---|
| | <p>If you do not specify a MAXVALUELENGTH, Vantage determines the maximum size of the intervals to be used for the histogram.</p> <p>For single-character statistics on CHARACTER and VARCHAR columns, <i>n</i> specifies the number of characters. For all other options, <i>n</i> specifies number of bytes.</p> <p>For multicolumn statistics, Vantage concatenates the values and truncates them if necessary to fit into the specified maximum size.</p> <p>The system does not truncate numeric values for single-column statistics. The system increases the interval size automatically if the specification is not sufficient to accommodate the full value for single-column statistics on numeric columns.</p> <p>For multicolumn statistics, if the maximum interval size truncates numeric statistical data, Vantage automatically increases the maximum interval size to accommodate the numeric column on the maximum size boundary.</p> <p>A larger maximum value size causes Vantage to retain the value until the specified maximum is reached, which can enable better single-table and join selectivity estimates for skewed columns. However, you should be selective when increasing the size for the required columns because increasing the maximum value size also increases the size of the histogram, which can increase query optimization time.</p> <p>You can only specify this option if you also specify an explicit column or index.</p> <p>You cannot specify MAXVALUELENGTH <i>n</i> for a standard recollection of statistics on an implicitly specified index or column set.</p> |
| NO SAMPLE | <p>Use a full-table scan to collect the specified statistics.</p> <p>You can only specify this option if you also specify an explicit index or column set.</p> <p>You cannot specify NO SAMPLE for a standard recollection of statistics on an implicitly specified index or column set.</p> <ul style="list-style-type: none"> • For the first collection of statistics, NO SAMPLE overrides the default. • For recollections of statistics, NO SAMPLE overrides the previous SAMPLE options and collects full statistics. |
| NO THRESHOLD | <ul style="list-style-type: none"> • Do not apply any thresholds to the collection of statistics. • Remove the existing threshold before collecting the statistics. <p>You can only specify this option if you also specify an explicit column or index.</p> <p>You cannot specify NO THRESHOLD for a standard recollection of statistics on an implicitly specified index or column set.</p> <ul style="list-style-type: none"> • For the first collection of statistics, NO THRESHOLD overrides the default settings. • For recollections of statistics, NO THRESHOLD overrides any previously specified THRESHOLD options and recollects the statistics without any thresholds. |
| NO THRESHOLD DAYS | <ul style="list-style-type: none"> • Do not apply a DAYS threshold to the collection of statistics. • Remove the existing DAYS threshold before collecting the statistics. <p>You can only specify this option if you also specify an explicit index or column set.</p> <p>You cannot specify NO THRESHOLD DAYS for a standard recollection of statistics on an implicitly specified index or column set.</p> <ul style="list-style-type: none"> • For the first collection of statistics, NO THRESHOLD DAYS overrides the default setting. |

| Option | Description |
|-------------------------|--|
| | <ul style="list-style-type: none"> For recollection of statistics, NO THRESHOLD DAYS overrides any previous DAYS threshold specification. |
| NO THRESHOLD PERCENT | <ul style="list-style-type: none"> Do not apply a PERCENT change threshold to the collection of statistics. Remove the existing PERCENT change threshold before collecting the statistics. <p>You can only specify this option if you also specify an explicit column or index. You cannot specify NO THRESHOLD PERCENT for a standard recollection of statistics on an implicitly specified index or column set.</p> <ul style="list-style-type: none"> For the first collection of statistics, NO THRESHOLD PERCENT overrides the default. For recollections of statistics, NO THRESHOLD PERCENT overrides any previous change threshold percent specification. |
| SAMPLE | <p>Scan a system-determined percentage of table rows to collect the specified statistics.</p> <p>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.</p> <p>You should use the SYSTEM SAMPLE option instead of SAMPLE.</p> |
| SAMPLE <i>n</i> PERCENT | <p>Scans the percentage of table rows that you specify rather than scanning all of the rows in the table to collect statistics.</p> <p>The value for <i>n</i> can be a decimal number or integer from 2 through 100. Specifying SAMPLE 100 PERCENT is equivalent to collecting full statistics.</p> <p>You can only specify this option if you also specify an explicit index or column set. You cannot specify SAMPLE <i>n</i> PERCENT for a standard recollection of statistics on an implicitly specified index or column set.</p> <ul style="list-style-type: none"> For the first collection of statistics, the specified sample percentage overrides the default. For recollection of statistics, SAMPLE <i>n</i> PERCENT overrides any previous SAMPLE option specifications and instead scans <i>n</i> percent of the rows in the table. |
| SYSTEM MAXINTERVALS | <p>Use the system-determined maximum number of intervals for this histogram. This option is valid for tables.</p> <p>You can only specify this option if you also specify an explicit index or column set. You cannot specify SYSTEM MAXINTERVALS for a standard recollection of statistics on an implicitly specified index or column set.</p> <p>Use the system-determined maximum number of intervals for the histogram.</p> |
| SYSTEM MAXVALUELENGTH | <p>Use the system-determined maximum column width for histogram values such as MinValue, ModeValue, or MaxValue.</p> <p>This option is valid for tables.</p> <p>You can only specify this option if you also specify an explicit index or column set. You cannot specify SYSTEM MAXVALUELENGTH for a standard recollection of statistics on an implicitly specified index or column set.</p> |

| Option | Description |
|--------------------------|--|
| SYSTEM SAMPLE | <p>Scan a system-determined percentage of table rows to collect statistics. Vantage may collect a sample of 100 percent several times before downgrading the sampling percentage to a lower value.</p> <p>SYSTEM SAMPLE is the default option if you do not specify the SAMPLE option. You can only specify this option if you also specify an explicit index or column set. You cannot specify SYSTEM SAMPLE for a standard recollection of statistics on an implicitly specified index or column set.</p> <p>The DBS Control field SysSampleOption in the STATISTICS field group contains the default for this option. For more information, see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</p> |
| SYSTEM THRESHOLD | <p>Collect statistics only if the percentage of changed data or the age of the current statistics exceeds the currently specified threshold for the statistic.</p> <p>Vantage automatically determines the appropriate change threshold to skip recollections if the changes are below this threshold. When doing this, the system takes into consideration the changes of update, delete and insert counts from the last collection of statistics, the history of column demographics, column usage, and Optimizer extrapolation techniques to determine the appropriate change threshold.</p> <p>If statistics are being collected first time, they are not skipped because there is no existing histogram data that can be used to determine the delta data change or age of the statistics.</p> <p>The change threshold can be different for different columns.</p> <p>You can only specify this option if you also specify an explicit index or column set. You cannot specify SYSTEM THRESHOLD for a standard recollection of statistics on an implicitly specified index or column set.</p> |
| SYSTEM THRESHOLD DAYS | <p>Uses a change-based percentage as a threshold for recollecting statistics.</p> <p>You can only specify this option if you also specify an explicit index or column set. You cannot specify SYSTEM THRESHOLD DAYS for a standard recollection of statistics on an implicitly specified index or column set.</p> |
| SYSTEM THRESHOLD PERCENT | <p>Only applies to change percentage. This is the default if you do not specify a PERCENT threshold option.</p> <p>Vantage automatically determines the appropriate change threshold to skip recollections if the percentage is below this threshold. When doing this, the system takes into consideration the percentage of change since the last update, delete, and insert counts from the last collection of statistics, the history of column demographics, column usage, and Optimizer extrapolation techniques to determine the appropriate change percentage threshold.</p> <p>You can only specify this option if you also specify an explicit index or column set. You cannot specify SYSTEM THRESHOLD PERCENT for a standard recollection of statistics on an implicitly specified index or column set.</p> <p>If you are collecting these statistics for the first time, they are not skipped because there is no existing histogram data that can be used to determine the percentage of change for the data.</p> <p>The DBS Control field SysChangeThresholdOption in the STATISTICS field group contains the default for this option. For more information, see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</p> |

| Option | Description |
|----------------------------|---|
| THRESHOLD <i>n</i> DAYS | <p>Do not to recollect statistics if the age of the statistic is less than the number of days specified.</p> <p>The value for <i>n</i> must be an integer number that represents the number of days. The valid range for <i>n</i> is 1 - 9999.</p> <p>You can only specify this option if you also specify an explicit index or column set. You cannot specify THRESHOLD <i>n</i> DAYS for a standard recollection of statistics on an implicitly specified index or column set.</p> <ul style="list-style-type: none"> For the first collection of statistics, collecting statistics is not skipped because no current histogram exists to use to determine the age of the current statistics. Instead, the specified number of days overrides the default setting. For recollection of statistics, THRESHOLD <i>n</i> DAYS overrides the previous THRESHOLD <i>n</i> DAYS specification and instead applies the specified number of days threshold to the collection. <p>The DBS Control field DefaultTimeThreshold in the STATISTICS field group contains the default for this option. For more information, see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</p> |
| THRESHOLD <i>n</i> PERCENT | <p>Do not to recollect statistics if the percentage of data change since the last collection is less than the specified percentage.</p> <p>The value for <i>n</i> can be either a decimal number or an integer number. The valid range of <i>n</i> is 1 - 9999.99.</p> <p>You can only specify this option if you also specify an explicit column or index. You cannot specify THRESHOLD <i>n</i> PERCENT for a standard recollection of statistics on an implicitly specified index or column set.</p> <ul style="list-style-type: none"> For the first collection of statistics, THRESHOLD <i>n</i> PERCENT overrides the default setting. For recollection of statistics, THRESHOLD <i>n</i> PERCENT overrides the previous threshold change percentage specification and instead applies the specified threshold percentage. <p>The DBS Control field DefaultUserChangeThreshold in the STATISTICS field group contains the default for this option. For more information, see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</p> |

FOR CURRENT Option

If you specify the FOR CURRENT option in a USING clause, Vantage applies that USING clause only for the current collection of statistics.

If you do not specify FOR CURRENT, then Vantage applies the USING clause to the current statistics collection and all subsequent collections of statistics on the specified column set.

Rules and Guidelines for COLLECT STATISTICS (Optimizer Form)

The following rules apply to using COLLECT STATISTICS.

- You can submit a COLLECT STATISTICS request in the following ways:

- As a single-statement request.
 - As the only statement in a macro.
 - As the only or last statement in an explicit Teradata session mode transaction bracketed by BEGIN and END TRANSACTION statements.
- Because base global temporary tables do not contain data, the statistics you collect on them have dummy histograms without intervals and a cardinality of 0 rows.

When you specify TEMPORARY to collect statistics on a materialized global temporary table for the first time, without specifying COLUMN or INDEX options, the materialized table inherits its statistics definitions from the base temporary table.

Specifying the TEMPORARY keyword to collect statistics on a global temporary table materializes the table in the current session, if it is not already materialized.

When you log off from a session or if the system forces a logoff, the database automatically drops the statistics from all materialized temporary tables.

- The system treats multicolumn statistics the same as index statistics if an index has been defined on the same column set. For an example, see [Collecting Statistics on Multiple Columns](#).
- The database preserves the ordering of columns when you collect multicolumn statistics. However, if the base table statistics are collected on an index using the INDEX keyword, the system uses index column ordering in the ascending order of the field ids.
- The maximum number of column and index sets you can recollect on a base table, global temporary table, hash index, or join index is 512.

This limit can be lower than 512 column or index sets subject to limits on the other resources such as the number of available spools and plastic or concrete step segment sizes.

- You cannot collect statistics on columns with a data type of Period, XML, BLOB, CLOB, or any UDT other than Geospatial.

You *can* collect statistics on the BEGIN and END expressions of a Period column.

You cannot collect statistics on an entire column with the JSON data type. However, you can collect statistics on extracted portions of the JSON data type. See *Teradata Vantage™ - JSON Data Type*, B035-1150.

- You can collect statistics on a base table column defined using a complex expression. You can also collect statistics on a single-table join index or hash index column defined using a complex expression in the respective select or column list. Either method enables the Optimizer to make accurate single-table cardinality estimates for queries that specify complex expressions in their predicates that it can match to a simple index column that is defined on that expression or a superset of it. For details, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142. However, COLLECT STATISTICS on base table is preferred as base table's statistics can be directly inherited by all join indexes.
- The following rules apply to collecting statistics on partitioned tables.
 - You can collect statistics on both the primary index, if the table is primary-indexed, and row partitioning column sets.

- If you specify to collect statistics using the name of the primary index, then statistics are collected only on the primary index column set.
- You cannot submit a COLLECT STATISTICS request on a table where:
 - Statistics have been dropped
 - COLLECT STATISTICS ... COLUMN or COLLECT STATISTICS ... INDEX clause has not been specified

You cannot recollect statistics on a table on which statistics have been dropped.

- You cannot specify a USING SAMPLE clause for standard recollection of statistics on implicitly specified column and index sets.
- When you specify a USING option for first time statistics collection or to reset an existing option, the database automatically applies the same options for subsequent recollections of statistics.
- When you specify the FOR CURRENT option, the database uses the USING options that you specify only for the current COLLECT STATISTICS request.

The database does not remember the USING options you specify FOR CURRENT for future recollections.

- To collect multiple statistics, you should group the statistics into a single COLLECT STATISTICS request.

For first time collections, group the statistics that specify the same USING options together. This enables the Optimizer to apply global optimizations such as early aggregation and aggregation rollups to hasten the collection process.

You can group recollections of statistics without regard to the various USING options specified for individual statistics.

You should use a table- level COLLECT STATISTICS request without specifying column or index references to refresh all the statistics for a given table or constant expression.

- You cannot specify the SUMMARY option with column references or USING options.
- When you recollect statistics on a column or index, the database automatically updates the table-level demographics.

Rules and Guidelines for the COLLECT STATISTICS (Optimizer Form) FROM *source_table* Clause

Specifying a FROM clause with a COLLECT STATISTICS request provides the same results as specifying a CREATE TABLE ... AS ... WITH DATA AND STATISTICS with the following exceptions.

- You must have the SELECT privilege on the source table to copy its statistics to a target table.
- COLLECT STATISTICS FROM does *not* copy data. It copies only the statistics for the specified column sets.
- You cannot specify a USING clause for a COLLECT STATISTICS request used to copy statistics.

Vantage instead copies the USING options for the most recent COLLECT STATISTICS request from the source to the target.

- Use the SUMMARY option to copy table-level demographics from a source table to a target table. When you copy statistics without specifying the column list for the source and target tables, Vantage implicitly copies the SUMMARY statistics from the source to the target. If the system copies the SUMMARY statistics implicitly, you should recollect those statistics natively when you later refresh them.

Unless you have a specific reason for copying SUMMARY statistics from a source table to a target table, you should not copy them.

- You can use a COLLECT STATISTICS request after you have created an identical target table using either a basic CREATE TABLE request or using a CREATE TABLE ... AS ... WITH DATA request.

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.

A target table created using a CREATE TABLE ... AS ... WITH DATA AND STATISTICS request may not 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 two.

- All data in the columns whose statistics are copied using COLLECT STATISTICS must be identical in the source and target tables.

If they are not identical, you are strongly advised to recollect the statistics for the target table.

- You do not obtain the same results from a COLLECT STATISTICS ... FROM *source_table* request and a CREATE TABLE ... AS ... WITH DATA AND STATISTICS request if the internal partition numbers are different. While Vantage copies the same statistics using both methods, in the COLLECT STATISTICS ... FROM *source_table* case, there are cases where the statistics are not valid for the data even though 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 following rules apply to copying statistics from a source table to an identical target table using a COLLECT STATISTICS request.

- When you specify column lists for both the source and target tables, Vantage copies the source table statistics to the target table only if the attributes for the columns in the specified target table are identical to the attributes for the columns in the specified source table. If they are not, the system does not copy the statistics and returns an error to the requestor.

The source table and target table column lists do *not* need to be identical.

- When you do not specify a column list for either the source table or for the target table, Vantage copies statistics from the source table to the target table only if the attributes for each column in the target table are the same as the attributes for the corresponding columns in the source table. Otherwise, Vantage does not copy the statistics and returns an error to the requestor.

For example, the attributes of all four columns of the target table must be identical to those of the four columns of the source table.

This rule applies to all column attributes: the data type, NULL/NOT NULL definition, uniqueness, case specificity, uppercase definition, and so forth, must match.

- When you specify a column list for the source table, but not for the target table, then both tables must have matching columns.

In this case, the system copies the statistics from the specified source table into the corresponding column set in the target table.

- When you specify a column list for the target table, but not for the source table, then the column attributes for both tables must be identical.

In this case, Vantage copies the statistics from the corresponding source columns into the columns specified in the target table.

- After checking the general eligibility rules, Vantage retrieves all source table single-column statistics.

The following additional rules apply to copying single-column statistics.

- If the appropriate single-column statistics have been collected for a column in the source table, then they are copied to the corresponding column of the target table.
- If the appropriate single-column statistics have not been collected for a column in the source table, statistics are not copied.
- If the appropriate single-column statistics have already been collected for both the source table and the target table, Vantage writes over the target table statistics with the statistics previously collected for the source table.

- After checking the general eligibility rules, Vantage uses an algorithm to determine which multicolumn statistics are eligible to be copied to the target table. The system copies statistics collected on all multicolumn sets or indexes in the source table to the corresponding multicolumn sets or multicolumn index sets of the target table.

The following additional rules apply to copying multicolumn statistics.

- If the appropriate multicolumn or multicolumn index statistics have been collected for the source table, but not for the target table, then Vantage copies those statistics from the source table to the target table.
- If the appropriate multicolumn or multicolumn index statistics have been collected for both the target table and the source table, Vantage writes over the existing target table statistics with the statistics previously collected for the source table.
- If the appropriate multicolumn or multicolumn index statistics have been collected for the target table, but not for the source table, Vantage does not change the existing target table statistics.

- The following additional rule applies to copying only a subset of single-column statistics.

If no statistics have been collected on the specified source table column, the request aborts and returns an error to the requestor.

- The following additional rule applies to copying only a subset of multicolumn statistics.

If statistics have not been collected on the specified source table multicolumn set, the request aborts and returns an error to the requestor.

- The following additional rules apply to copying statistics for global temporary tables.
 - If both the source and target tables are global temporary tables, then the system copies the temporary statistics from the source table as temporary statistics on the target table.
 - If the source table is a global temporary table and the target table is a permanent table, then the system copies the temporary statistics from the source table as permanent statistics on the target table.
 - If the target table is a global temporary table and the source table is a permanent table, then the system copies the statistics from the source table as temporary statistics on the target table.
 - If the target table is a base global temporary table, then the system copies the source table statistics as zeroed statistics on the target table.

The term *zeroed statistics* refers to the condition in which the synopsis data structures, or histograms (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142), for the statistics on a column set or index have been constructed, but no statistics have been collected on the column set or index. A common example of this state is when you create a new table and then collect statistics on it when it does not yet contain any data rows.

- If the target table is an unmaterialized global temporary table, then the system materializes it before it copies temporary statistics from the source table as temporary statistics on the target table.

Locks and Concurrency

When you perform a COLLECT STATISTICS request, the system places an ACCESS lock on the table from which the demographic data is being collected. The database places a rowhash-level WRITE lock on DBC.StatsTbl after the statistics have been collected and holds the lock only long enough to update the dictionary rows. You can easily see this behavior documented in the boldface text in stage 4 of the following EXPLAIN of a COLLECT STATISTICS request on COLUMN x1.

```
EXPLAIN COLLECT STATISTICS ON t1 COLUMN x1;
```

```
*** Help information returned. 18 rows.
```

```
*** Total elapsed time was 1 second.
```

```
Explanation
```

```
-----
```

```
1) First, we lock DF2.t1 for access.
```

```
2) Next, we do a COLLECT STATISTICS step from DF2.t1 by way
```

of an all-rows scan into Spool 3 (all_amps), which is built locally on the AMPs.

- 3) Then we save the UPDATED STATISTICS from Spool 3 (Last Use) into Spool 4, which is built locally on the AMP derived from **DBC.StatsTbl** by way of the primary index.
- 4) We lock **DBC.StatsTbl** for write on a RowHash.
- 5) We do a single-AMP MERGE DELETE to **DBC.StatsTbl** from Spool 4 (Last Use) by way of a RowHash match scan. New updated rows are built and the result goes into Spool 5 (one-amp), which is built locally on the AMPs.
- 6) We do a single-AMP MERGE into **DBC.StatsTbl** from Spool 5 (Last Use).
- 7) We spoil the parser's dictionary cache for the table.
- 8) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

-> No rows are returned to the user as the result of statement 1.

These locks prevent parsing of new requests against the data table for which the statistics are being collected until the operation completes.

In general, COLLECT STATISTICS requests can run concurrently with DML requests, other COLLECT STATISTICS requests, DROP STATISTICS requests, and HELP STATISTICS requests against the same table.

You should *not* collect statistics for the first time concurrently on the same table. However, you can collect or recollect statistics from different tables and run other statistics-related requests such as DROP STATISTICS, HELP STATISTICS, and SHOW STATISTICS concurrently.

COLLECT STATISTICS requests can also run concurrently with a CREATE INDEX request against the same table as long as they do not specify the same index.

The following bullets list the exceptions to this rule. The exceptions assume that you do not specify a LOCKING modifier on the table with the COLLECT STATISTICS request (or if you do specify a LOCKING modifier, it requests only ACCESS-level locking).

- If a transaction containing a DML request places an EXCLUSIVE lock on a table, the DML request blocks COLLECT STATISTICS requests, or a COLLECT STATISTICS request blocks the other DML request, depending on which request places its table lock first. Recall that COLLECT STATISTICS uses an ACCESS lock on the table by default. You should not normally specify an EXCLUSIVE lock on a table in conjunction with a DML request.
- A COLLECT STATISTICS request is blocked from obtaining its row-hash WRITE locks on **DBC.StatsTbl**. These locks are requested near the completion of the COLLECT STATISTICS operation and are held only long enough to update the dictionary rows with the newly collected statistics.

This can cause other requests to be blocked if the dictionary information for the table is not already cached. Note that once the WRITE lock is obtained, the COLLECT STATISTICS request should finish quickly, and parsing of the blocked requests should resume.

Collecting Statistics When No COLUMN or INDEX Clause Is Specified

If you do not specify a COLUMN or INDEX clause, a COLLECT STATISTICS request updates the statistics for all columns and indexes for which statistics had previously been collected. This is called *recollecting* statistics.

The exceptions are as follows:

- Statistics that have been deleted by a DROP STATISTICS request are not updated.
To replace deleted statistics, specify the appropriate set of column or index names in a COLLECT STATISTICS request.
- Statistics can be recollected on a combined maximum of 512 implicitly specified columns and indexes. The system returns an error if statistics have been collected for more than 512 columns and indexes in the table.
This limit can be lower than 512 columns subject to limits on the other resources such as the number of available spools and plastic or concrete step segment sizes.
- The system recollects statistics in the same mode (full file scan, sampled, or cylinder index scan) that you specified when the information was originally collected.
A cylinder index scan performs up to $n-1$ data block reads, where n is the number of partitions, when collecting statistics on the system-derived PARTITION column of a table.
- You cannot specify an explicit USING SAMPLE clause to recollect statistics.
If the original statistics were collected using sampling, then the recollected statistics are sampled as well. Otherwise, the system recollects full statistics for the specified table.
- You cannot collect sampled statistics on a COLUMN set that is also a component of the row partitioning expression of a partitioned table or join index; therefore, the system recollects full statistics on any such column.
You can, however, collect sampled statistics on an INDEX set that contains partitioning columns.

Collecting Statistics on Single-Column Indexes

If an index has a single column *column_name*, then collecting statistics for COLUMN (*column_name*) has the same result as collecting statistics for INDEX (*column_name*).

Collecting Statistics on Multiple Columns

If you frequently perform queries with search conditions specified on multiple columns, you should collect statistics on those columns jointly. Statistics collected in this manner permit the Optimizer to more accurately estimate the number of qualifying rows for queries that specify both of those columns. You cannot collect multicolumn statistics on global temporary tables.

Note that the statistics derived for *column_1* and *column_2* when collected jointly are different from those derived by collecting statistics separately on those same columns. For example, suppose you have two columns named *FirstName* and *LastName*. Specifically, the statistics collected on *FirstName* by itself provide an estimate of the number of individuals having the first name *John*, for example, and the statistics collected on *LastName* provide an estimate of the number of individuals having the last name *Smith*.

These statistics might not provide good estimates of the number of individuals named *John Smith*. For that, you need to collect statistics on the two columns jointly. This example is provided only to make a clear distinction between collecting statistics on individual columns and collecting statistics on those same columns jointly.

The following table provides a general guideline for determining how to collect statistics on multiple columns.

| IF a set of unindexed columns is ... | THEN collect statistics on the ... |
|---|------------------------------------|
| frequently specified together as an equality condition or WHERE clause join condition | column set jointly. |
| infrequently specified together as an equality condition or WHERE clause join condition | individual columns separately. |

Vantage treats multicolumn statistics the same as index statistics if an index has been defined on the same column set.

For example, if you have defined an index on (*x1*, *y1*) in table *t1*, then the following two requests collect the identical statistics and store those statistics in DBC.StatsTbl.

```
COLLECT STATISTICS ON t1 COLUMN (x1,y1);

COLLECT STATISTICS ON t1 INDEX (x1,y1);
```

As a general rule, if a column has many nulls and it is not specified frequently in predicates, you probably should not include it when you are collecting multicolumn statistics.

For example, suppose you are considering collecting multicolumn statistics on columns 1, 2, and 3, and their values are like the following, where a QUESTION MARK character represents a null.

| <u>column 1</u> | <u>column 2</u> | <u>column 3</u> |
|-----------------|-----------------|-----------------|
| 1 | 2 | 3 |
| ? | 2 | 3 |
| 1 | ? | 3 |
| 1 | 2 | ? |
| ? | ? | ? |
| 1 | 2 | ? |

| | | |
|---|---|---|
| 1 | 2 | ? |
| 1 | 2 | ? |
| 1 | 2 | ? |
| 1 | 2 | ? |

You would probably not want to collect multicolumn statistics that include column 3 unless it is specified in a large number of predicates.

Collecting Statistics on the PARTITION Column and the Row Partitioning Column Set

On partitioned tables (row or column), always collect PARTITION statistics and statistics on the row partitioning column set. Refresh statistics whenever row partition demographics change significantly. The Ten Percent Rule, the usual guideline of refreshing statistics after a 10 percent change in *table* demographics, does not apply for row partitioning columns. Instead, you should recollect statistics whenever the demographics for a row *partition* change by 10 percent of more. This guideline applies to both the row partitioning column set and to the system-derived PARTITION column for a partitioned table. You cannot collect statistics on the system-derived PARTITION#L *n* columns of a partitioned table.

This is a guideline, not a rule, and you might need to adjust the point at which you refresh partition statistics upward or downward to maintain high quality query plans.

For row partitioned tables, the Optimizer uses PARTITION statistics to estimate the number of populated partitions, selectivity based on partition elimination, I/O reduction ratio, and so forth. For column partitioned tables, the Optimizer uses PARTITION statistics to estimate the column compression ratio. This cannot be estimated from evaluating standard data statistics and demographics.

You should collect single-column PARTITION statistics to enable the Optimizer to take advantage of the more accurate partitioning costing made possible when PARTITION statistics exist. For more information about the system-derived PARTITION column, see *Teradata Vantage™ - Database Design*, B035-1094. Collecting these statistics is fairly quick because collecting such statistics, even at a 100 percent level, only requires scanning of the cylinder indexes for the table plus at most $n+1$ data blocks, where *n* is the number of partitions for the table. Therefore, the system does not scan all of the data blocks for the table as it does for other columns when collecting statistics on 100 percent of the data.

The system does *not* use this fast collection method if a PARTITION column is part of a multicolumn statistics collection. Statistics on the system-derived PARTITION column provide the Optimizer with the best estimate of the number of populated partitions and the cardinality of each partition when statistics were last collected.

PARTITION statistics provide the Optimizer with information on whether partitions are empty or populated. The Optimizer can then use this information to make better cost estimates of candidate query plans when there is a significant number of empty partitions. Note that the Optimizer must generate a plan that accesses *all* defined partitions because they might not be empty at the time the plan executes.

If PARTITION statistics have not been collected, the Optimizer estimates the cardinality of a partition as follows.

$$\text{Partition cardinality} = \frac{\text{Cardinality}_{\text{table}}}{\text{Adjusted number of partitions}}$$

Therefore, empty partitions can cause the Optimizer to underestimate the partition cardinality. The Optimizer reduces the number of partitions (especially if the maximum number could be 65,535) in some cases to avoid underestimating. However, the best policy to ensure accurate partition cardinality estimates is to define only partitions that are actually in use and to collect statistics on the system-derived PARTITION column.

Collecting multicolumn statistics for a base table that include the system-derived column PARTITION can provide some benefit for estimating plan costs for a query when all the columns, including the system-derived PARTITION column, in the multicolumn statistics have single-table equality predicates for the query.

If a partition changes from being empty to being populated or from being populated to empty, you must recollect statistics on the system-derived PARTITION column.

You must also refresh statistics when significant changes occur to the table. The guideline of 10 percent change (from insert, update, delete, and merge operations) in rows applies at the partition level for partitioned tables instead of the table level. You might need to adjust this guideline as needed for your application.

The only exception to this guideline is a partitioned table row-partitioned by a single-column expression. In this case it is not necessary to collect separate PARTITION statistics in addition to the statistics you collect on the row partitioning column because those column statistics are automatically inherited as single-column PARTITION statistics.

For example, assume the following table definition.

```
CREATE TABLE table_1 (
  col_1 INTEGER,
  col_2 INTEGER)
PRIMARY INDEX(col_1) PARTITION BY col_2;
```

If you collect individual column statistics on *col_2*, which is the partitioning column, then those statistics are also inherited as PARTITION statistics.

Because PARTITION is not a reserved keyword, you can use it as the name for any column in a table definition, but you should not. This practice is *strongly* discouraged in all cases, and particularly for partitioned tables, because if you explicitly name a column *partition*, the database resolves it as a regular column. As a result, you cannot collect statistics on the system-derived PARTITION column of any table that also has a user-defined column named *partition*.

The system uses multicolumn PARTITION statistics to do single-table estimates when all the columns, including the PARTITION column, have single-table equality conditions.

You should also collect SUMMARY statistics on nonpartitioned tables, because these statistics can be collected quickly and they are very useful to the Optimizer for making cardinality estimates.

You can specify a USING SAMPLE clause to collect single-column PARTITION statistics, but the specification is ignored. Instead, the system automatically resets the internal sampling percentage to 100. It is important to collect accurate statistics for this case, and collecting PARTITION statistics is a very fast operation. As a result, sampling is not required to improve the performance of collecting those statistics. The sampling flag in a detailed statistics report is always reported as 0 to document this behavior. See [SHOW STATISTICS](#). Note that the USING SAMPLE clause *is* honored for multicolumn PARTITION statistics. The database begins the collection process using a 2% sampling percentage and then increases the percentage dynamically if it determines that the values are skewed more than a system-defined threshold value.

Collecting Statistics for Star Joins

You should always collect statistics on the primary index or primary AMP index of small tables that you plan to join regularly to large tables to ensure maximal star join optimization.

Collecting Statistics on NUSIs Defined Using ALL and ORDER BY

When you use the ALL and ORDER BY keywords in a CREATE INDEX request, you should always collect statistics on the column on which the index is defined. This permits the Optimizer to more accurately compare the cost of using a NUSI-based access path with range or equality conditions on the sort key column.

Collecting Statistics on Large Objects, UDTs, and Period Columns

You cannot collect statistics on BLOB, CLOB, XML, non-deterministic UDT, ARRAY, VARRAY, or Period columns.

You *can* collect statistics on the BEGIN and END expressions for a Period type column.

Collecting Statistics on Derived Period Columns

You cannot collect statistics on derived Period columns. Instead, you should collect statistics on the Begin and End expressions of the derived Period column.

Collecting Statistics on Hash and Join Indexes

Single- and multitable join indexes are fundamentally different database objects in many ways, and they are used for different purposes.

Statistics for base tables and multitable join indexes are *not* interchangeable, though a base table and an underlying non-sparse single-table join index can inherit statistics from one another. See the description of *bidirectional inheritance* in *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142. If

a column is defined for a base table and a multitable join index, you must collect statistics for each object on which it is defined. The statistics for column values stored in a base table can be very different from those for the same column stored in a derived join result in a multitable join index.

If you do not explicitly collect statistics for a multitable join index, the Optimizer does *not* attempt to use the statistics collected for its underlying base tables in an effort to derive statistics for the join index.

A multitable join index table can be thought of as a special base table that represents a derived or pre-join table. For example, execution plans that involve a join index must access it using a full table scan or an indexed scan. Statistics might need to be collected on search condition keys to assist the Optimizer in evaluating these alternative access paths.

Because execution plans might involve joining a join index with yet another table that is not part of the join index, it is often useful to collect statistics on these join columns to assist the Optimizer in estimating cardinality.

Minimally, you should collect statistics on the primary index or primary AMP index of the join index to provide the Optimizer with baseline statistics including the cardinality of the join index.

You should also collect statistics on secondary indexes defined on any join indexes. This helps the Optimizer to evaluate alternate access paths when scanning a join index.

You might also want to consider collecting statistics on any additional join index columns that frequently appear in WHERE clause search conditions, especially if the column is the sort key for a value-ordered join index.

There is little value in collecting statistics for joining columns that are specified in the join index definition itself. Instead, you should collect statistics on their underlying base tables.

This action also improves the performance of creating and maintaining join indexes, particularly during updates. Guidelines for doing this are identical to those for collecting statistics on the tables accessed for any regular join query.

Collecting Statistics on Unique Join Indexes

You should always collect statistics on the primary index of a unique join index, whether it is user-defined or system-defined, so the Optimizer can use those statistics to most accurately estimate the cost of using a particular unique join index.

Collecting Statistics on a PERIOD Data Type Column

You can collect statistics on the BEGIN and END bound functions of PERIOD type columns, for example,

```
COLLECT STATISTICS COLUMN BEGIN(Policy_Duration)
                        AS Stats_BegDuration,
                        COLUMN END(Policy_Duration) AS Stats_EndDuration
ON policy_types;
```

Collecting Statistics on a UDF

You can collect statistics on deterministic UDFs.

For example, suppose you create the following UDF.

```
CREATE FUNCTION months_between(Date1 TIMESTAMP, Date2 TIMESTAMP)
RETURNS FLOAT
LANGUAGE C
NO SQL
DETERMINISTIC
SPECIFIC months_between_tt
EXTERNAL NAME 'CS!months_between_tt!$PGMPATH$/months_between_tt.c'
PARAMETER STYLE SQL;
```

The following request collects statistics on the UDF *months_between()*.

```
COLLECT STATISTICS
COLUMN months_between(BEGIN(policy_duration),
END(policy_duration)) AS Stats_MthsBetweenBegAndEnd
ON Policy_Types;
```

Using Named Indexes To Collect Statistics

You can specify an index name or index definition in a COLLECT STATISTICS request. To collect statistics from an unnamed index, you must specify the complete index definition.

Related Information

The following topics and documents are related to optimizer statistics:

- [DROP STATISTICS \(Optimizer Form\)](#)
- [HELP STATISTICS \(Optimizer Form\)](#)
- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Design*, B035-1094
- *Statistics Enhancements Orange Book*, 541-0010042

COMMENT (Comment Placing Form)

Typical Use for COMMENT

The COMMENT statement is typically used in conjunction with a CREATE statement such as CREATE TABLE, CREATE METHOD, or CREATE TYPE to describe the:

- Newly created database object.
- Definition for a method (UDM) or user-defined data type (UDT).

Note that you cannot place comments on indexes, so you cannot use this statement to place comment strings on hash, join, or secondary indexes.

Rules for Using COMMENT

The following rules apply to using the COMMENT statement.

Transaction Processing

The comment-placing form of COMMENT is processed as a DDL statement for transaction processing, so it cannot be used in 2PC session mode. For a description of the two-phase commit protocol, see *Introduction to Teradata*.

If you specify a COMMENT request within an explicit transaction in Teradata session mode, it must be the last request in that transaction.

Comments on UDFs

COMMENT ON FUNCTION includes for the following UDF types:

- Simple external UDFs
- Table UDFs
- SQL UDFs

COMMENT ON TYPE

COMMENT ON TYPE works for both UDT types:

- Distinct
- Structured

COMMENT ON METHOD

COMMENT ON METHOD works for the following method types:

- Constructor
- Instance
- Mutator

You must specify the specific method name for any method on which you want to place a comment.

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
```

Related Information

For information on the comment returning form of the COMMENT statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

CREATE AUTHORIZATION - CREATE ERROR TABLE

These topics provide supplemental usage information about selected SQL DDL statements alphabetically from CREATE AUTHORIZATION through CREATE ERROR TABLE.

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE AUTHORIZATION and REPLACE AUTHORIZATION

Providing Security for User-Written External Routines

Authorization definitions permit users to issue operating system I/O calls from within an external routine. The ANSI SQL:2011 specification collectively refers to user-written non-SQL modules as external routines.

Vantage requires any external routine that performs operating system I/O to run in protected mode as a separate process than runs under an explicitly specified user ID. See [Protected and Unprotected Execution Modes](#). Authorization objects provide a flexible, yet robust, scheme for providing the authorizations required by these external routines without exposing the system to these potential problems.

The principal difference between an external routine running in protected mode (or in secure mode is that when an external routine runs in protected mode, it always runs as the OS user *tdatuser*, while an external routine that runs in secure mode can run as any OS user you want to associate with an external authorization. While *tdatuser* has no special privileges, an OS user associated with an external authorization can have any privileges on OS files you want to assign to it. All that is required is that the OS user with special privileges be specified in the EXTERNAL SECURITY clause of the SQL definition for the external routine associated with it.

See the following topics:

- [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#)
- [ALTER METHOD](#)
- [ALTER FUNCTION \(External Form\)](#) or [ALTER PROCEDURE \(External Form\)](#)
- [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#)

DEFINER and INVOKER Authorizations

An external routine with an EXTERNAL SECURITY clause DEFINER authorization always uses the OS user authorization that is associated with the creator of the authorization object.

An external routine with an EXTERNAL SECURITY clause INVOKER authorization uses the OS user authorization that is associated with the database user who invokes it.

Both routines require external authorization. The difference is in which OS user authorization the routine uses when it runs:

- All use the same OS user authorization, that of the user who created the authorization, when the routine is defined with a DEFINER authorization.
- All use a different OS user authorization, that of the user who invokes the routine, when the routine is defined with an INVOKER authorization.

The following table summarizes these differences:

| WHEN an external routine is to be run with the ... | THEN you should specify this type of authorization ... |
|---|--|
| same OS user authorization independent of the user who runs the routine | DEFINER. |
| OS user authorization as specified by the user who runs the routine | INVOKER. |

Note that when you define an INVOKER authorization, the database creates a second authorization named `INVOKER_AUTH` under the same database (see the report on the next page).

There can only be one INVOKER authorization per database, so when one is defined, the database creates one entry using the specified name and another, duplicate, entry using the name `INVOKER_DEFAULT`, both of which are stored in `DBC.TVM`. Apart from their names, the two INVOKER default entries are identical.

When you specify an `EXTERNAL SECURITY INVOKER` clause for a UDF, no authorization name exists. the database looks up the `INVOKER_DEFAULT` name in `DBC.TVM` for the current database, where it finds a correct entry because there can only be one per database.

If you `DROP AUTHORIZATION` specifying the name you created for it, the database drops both entries. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

```
SELECT *
FROM DBC.authorizationsV
ORDER BY databasename;
```

| DatabaseName | AuthorizationName | AuthorizationId | TableKind | Version | AuthorizationType | AuthorizationSubType | OSDomainName | OSUserName |
|--------------|-------------------|-----------------|-----------|---------|-------------------|----------------------|--------------|------------|
| fisjco04 | INVOKER_DEFAULT | 0000E3060000 | X | 1 | I | D | | fisjco04 |
| fisjco04 | JohnAuth | 0000E2060000 | X | 1 | I | D | | fisjco04 |
| sinan04 | Andreiauth | 0000E4060000 | X | 1 | D | N | | sinan04 |
| venba01 | INVOKER_DEFAULT | 0000E8060000 | X | 1 | I | D | | venba01 |
| venba01 | VenkatAuth | 0000E7060000 | X | 1 | I | D | | venba01 |
| vuolo01 | INVOKER_DEFAULT | 0000E6060000 | X | 1 | I | D | | vuolo01 |
| vuolo01 | LouAuth | 0000E5060000 | X | 1 | I | D | | vuolo01 |
| vuolo01 | TestAuth | 0000DC060000 | X | 1 | D | N | | vuolo01 |

Authorization Rules

- When you submit a `CREATE AUTHORIZATION` request, the database validates the authorization. The system verifies the following items, and if any of them is false, the authorization attempt returns an error:
 - The OS password is validated on each node.
 - The OS platform user must belong to the OS group named `tdatudf`.

`tdatudf` need not be the primary group for the OS platform user, but that user must be a member of the group. Consult your Linux operating system documentation, as appropriate, for the specific definitions of group, local group, and so on.

The tdatudf group is used by protected mode external routines and must always exist. You can change the tdatudf group assignment for external routines using their EXTERNAL SECURITY clause.

The best practice is to make this group serve only the external routines that specify an EXTERNAL SECURITY clause and not use it for any other purpose.

- The authentication process uses whichever authentication methods are set up by your site. For example, if your site uses Kerberos for authentication, then users are validated using that method. Authentication is automatic as long as the platform nodes are part of the global authentication process. If they are not part of that process, then you must create a local OS user on each node. A local OS user is any user that must be created manually. Local OS users are not associated with a domain.
- You cannot specify CREATE AUTHORIZATION in a macro definition because the authorization password is not saved in either encrypted or unencrypted form. This would compromise the security of the OS logon user ID. See [CREATE MACRO and REPLACE MACRO](#).
- Any external routine that accesses external data must run in either secure or protected mode as a separate process under the user ID of an OS platform user.

When you create an external routine that includes an EXTERNAL SECURITY clause, the name of the OS user specified in that clause must be a member of the group tdatudf. The system creates this group by default during the installation of the database.

The tdatudf group must always exist and must not be altered.

The reason for these restrictions is that the database always uses the tdatudf group to run protected or secure mode routines.

When an external routine runs in either protected or secure mode, it runs as the OS user associated with its authorization.

In both protected and secure modes, the system sets the platform process up on a PE or AMP vproc as needed to execute the external routine.

You can run a maximum of 20 protected mode servers and 20 secure mode servers per AMP or PE. To clarify, you can run up to 40 “protected” mode servers concurrently per AMP or PE, but not more than 20 each of the protected and secure mode types.

- The number of server processes that can be set up on any one vproc is limited. The limit for both C/C++ and Java servers is controlled by different fields of the cufconfig utility. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

When modifying the cufconfig fields, note that each protected or secure mode process uses its own private shared memory, which requires private disk swap space. Each process for a routine written in C or C++ requires 256 KB of disk space.

For example, if two protected mode servers are allocated per vproc and there are 8 vprocs on the node, 4 megabytes of system disk file space is required. Refer to the following equation.

$$\text{Total swap space required} = \frac{\text{Number of vprocs}}{\text{node}} \times \frac{\text{Number of protected mode servers}}{\text{vproc}} \times \frac{256 \text{ KB}}{\text{server}}$$

By substitution, a system defined with 2 protected mode servers per vproc requires 4 MB of disk swap space.

$$\text{Total swap space required} = \frac{8 \text{ vprocs}}{\text{node}} \times \frac{2 \text{ servers}}{\text{vproc}} \times \frac{256 \text{ KB}}{\text{server}} = 4 \text{ MB per node}$$

A system defined with 20 protected mode servers per vproc for C and C++ external routines requires 40 MB of disk swap space.

$$\text{Total swap space required} = \frac{8 \text{ vprocs}}{\text{node}} \times \frac{20 \text{ servers}}{\text{vproc}} \times \frac{256 \text{ KB}}{\text{server}} = 40 \text{ MB per node}$$

The system does not allocate space until an external routine is executing that requires the space. If you run 10 sessions, and each one executes a query that uses a UDF at the same instance, then the system creates 10 protected mode servers per vproc.

However, if the 10 sessions execute the UDF queries sequentially, only one protected mode process per vproc must be created. Once a protected mode server is set up, it remains in existence until the database restarts. If any servers are not used, the system swaps them out to disk, so they do not consume any physical memory resources.

Each Java server for external routines requires roughly 30 MB of memory for swap space, and there can be two such Java servers per node. A Java UDF multithreaded server for non-secure mode Java UDFs uses a minimum of an additional 30 MB (the amount required can be larger, depending on the size of the JARs for a user), so each node requires approximately 100 MB of swap space if all server flavors are used.

Function of CREATE AUTHORIZATION Requests

The purpose of an authorization object is to specify the user context to use when running an external routine that performs operating system I/O operations. See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#), [CREATE FUNCTION \(Table Form\)](#), [CREATE METHOD](#), [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#), and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Authorization objects associate a user with an OS platform user ID. With an OS platform user ID, a user can log onto a database node as a native operating system user and be able to run external routines that perform OS-level I/O operations.

You must create an authorization object for any external routine that has an EXTERNAL SECURITY clause as part of its definition. You must define authorization objects for the following users and situations:

- A user who needs to run external routines that contain an INVOKER security clause.
- A user who needs to be the definer of any external routine modules that contain the DEFINER external clause.

Without the appropriate authorization objects having been created, none of the external routines containing an EXTERNAL SECURITY clause can run.

When you submit a CREATE AUTHORIZATION statement, the system validates the values for the specified user variables. If the specified user object has not yet been created on all database nodes or if any of the other information you specified is not correct, the statement returns an error message to the requestor.

The system permits only three failed attempts to create an authorization object. After three failed attempts, the system returns an appropriate error message to the requestor.

You must first log off the system and then log back on. The DBA also has the option of activating access logging on CREATE AUTHORIZATION to enable the tracking of suspicious attempts to perform it. See BEGIN LOGGING in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE CAST and REPLACE CAST

Difference Between CREATE CAST and REPLACE CAST

To perform a CREATE CAST statement, a user-defined cast that performs the same conversion cannot exist or the statement returns an error to the requestor.

| IF you perform a REPLACE CAST statement and there is ... | THEN the system ... |
|--|--|
| no existing user-defined cast that performs the specified conversion | creates a new user-defined cast. The result is the same as if you had performed a CREATE CAST statement with all the same specifications. |
| an existing user-defined cast that performs the specified conversion | replaces the existing user-defined cast definition with the newly defined cast routine. |

If the cast routine is a method, then that method must be already be defined or the system returns an error to the requestor and does not register the method for casting.

Function of Casting UDTs

A cast operation converts the specified *source_data_type* to the specified *target_data_type* using the specified *cast_function* or *cast_method* routine analogous to the CAST operator used to convert predefined data types. For more information about the CAST operator, see *Teradata Vantage™ - Data Types and Literals*, B035-1143. At least one of *source_data_type* or *target_data_type* must be a UDT (see [CREATE TYPE \(Distinct Form\)](#) and [CREATE TYPE \(Structured Form\)](#)).

Casts are the mechanism the system uses to make conversions in either direction between a UDT and another data type, which might or might not be another UDT. UDTs can have multiple casts, where each cast supports the conversion for a particular source-target combination.

For example, the UDT named *circle_type* might have the following four separate casts:

- *circle_type* to VARCHAR
- VARCHAR to *circle_type*
- *circle_type* to VARBYTE
- VARBYTE to *circle_type*

After you create a user-defined cast, it is immediately available for use with the SQL CAST function. See *Teradata Vantage™ - Data Types and Literals*, B035-1143. For example, if you define a cast to convert a *circle* UDT to a VARCHAR(40), you can specify the following cast operation:

```
CAST(circle_udt_expression AS VARCHAR(40))
```

The system automatically and implicitly supports casts from or to a predefined data type when you cast a compatible predefined type to the appropriate predefined type serving as the cast source of target type.

The scope of operations under which the system applies UDT-related implicit casts is a subset of the scope of operations under which traditional predefined data type-related implicit casts are applied. See *Teradata Vantage™ - Data Types and Literals*, B035-1143.

For example, consider the following equality comparison in a WHERE clause:

```
WHERE myUdt = NEW CircleUdt(1,1,9)
```

This is not within the scope of the implicit casts the system supports because it is not an assignment operation, supported system operator, or supported predefined function, so no implicit casts are applied. You must specify explicit cast operations to ensure that the UDTs being compared are identical.

Using the AS ASSIGNMENT Clause To Make a Cast Operation Implicitly Invokable

You can make a user-defined cast implicitly invokable by including the AS ASSIGNMENT clause in its CREATE CAST statement. When you define a cast operation in this way, the system implicitly invokes the cast for all assignment operations. An assignment operation is defined to be an INSERT, MERGE, UPDATE, or parameter passing operation.

You can disable this functionality by setting the DisableUDTImplCastForSysFuncOp DBS Control flag to TRUE. The default value for this flag is FALSE, meaning that implicit casting is enabled. See *Teradata Vantage™ - Database Utilities*, B035-1102. This action disables implicit UDT-to-predefined data type casts for built-in system functions and operators.

If DisableUDTImplCastForSysFuncOp is set to 0x01 and you attempt to pass a UDT into a built-in system function or operator, an error is returned to the requestor.

This flag *only* affects built-in system functions and operators. The system continues to invoke implicit casts for insert and update operations and for parameter passing with respect to UDFs, methods, and stored procedure operations when the DBS Control DisableUDTImplCastForSysFuncOp parameter is set to FALSE.

For information about setting the value for the DBS Control DisableUDTImplCastForSysFuncOp parameter, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Creating Parallel Cast and Transform Functionality

For best practice, pair transform group functionality with the equivalent cast functionality. See [CREATE TRANSFORM and REPLACE TRANSFORM](#). Create parallel functionalities for the two conversion types, for example:

| Cast Functionality | Equivalent Transform Functionality |
|---|---|
| External predefined data type to UDT cast | External predefined data type to UDT <i>tosql</i> transform |
| UDT to external predefined data type cast | UDT to external predefined data type <i>fromsql</i> transform |

This is highly recommended if you want to use a UDT in the same way you would use any other data type. Especially in regard to load utilities and the USING row descriptor, it is recommended that the type developer back up their tosql and fromsql functionality with equivalent external type (predefined) to UDT and UDT to external type (predefined) implicit casts, respectfully.

For equivalence, you can reference the same external routines in the CREATE TRANSFORM and CREATE CAST statements for the same UDT.

The following table describes the differences for distinct and structured UDTs:

| FOR this kind of UDT ... | You must do the following work to create parallel casting and transform functionality... |
|--------------------------|--|
| distinct | none if you plan to use the system-generated casts and transforms. However, if you decide to write your own external cast and transform routines for the UDT, you must create the parallel cast and transform functionalities explicitly using the CREATE CAST and CREATE TRANSFORM statements, respectively. |
| structured | create the necessary casts and transforms explicitly, using the same external routines for both. |

For more information, see [CREATE CAST and REPLACE CAST](#) and [CREATE TRANSFORM and REPLACE TRANSFORM](#).

Possible Predefined Type Source and Target User-Defined Cast Combinations

For user-defined casts, the system supports a degree of implicit predefined data type-to-predefined type conversions. The following table lists the inputs for a predefined data type-to-UDT cast and targets for a UDT-to-predefined data type cast.

| Source Input | Source Data Type of CAST Definition | Target Data Type of CAST Definition | Target Output |
|---|--------------------------------------|-------------------------------------|---------------|
| BYTEINT SMALLINT INTEGER BIGINT DECIMAL NUMERIC REAL FLOAT DOUBLE PRECISION DATE CHARACTER VARCHAR | BYTEINT | UDT | UDT |
| | SMALLINT | | |
| | INTEGER | | |
| | BIGINT | | |
| | DECIMAL NUMERIC | | |
| | REAL FLOAT DOUBLE PRECISION | | |
| | | | |

| Source Input | Source Data Type of CAST Definition | Target Data Type of CAST Definition | Target Output |
|---|---|---|---------------|
| BYTE VARBYTE BLOB | BYTE VARBYTE BLOB | UDT | UDT |
| CHARACTER VARCHAR CLOB BYTEINT SMALLINT INTEGER BIGINT DECIMAL NUMERIC REAL FLOAT DOUBLE PRECISION DATE | CHARACTER VARCHAR CLOB | | |
| DATE BYTEINT SMALLINT INTEGER BIGINT DECIMAL NUMERIC REAL FLOAT DOUBLE PRECISION DATE | DATE | | |
| TIME TIME WITH TIME ZONE CHARACTER VARCHAR | TIME TIME WITH TIME ZONE | | |
| TIMESTAMP TIMESTAMP WITH TIME ZONE CHARACTER VARCHAR | TIMESTAMP TIMESTAMP WITH TIME ZONE | | |
| INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH | INTERVAL YEAR INTERVAL YEAR TO MONTH | | |

| Source Input | Source Data Type of CAST Definition | Target Data Type of CAST Definition | Target Output |
|---|---|---|---|
| | INTERVAL MONTH | | |
| INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL SECOND | INTERVAL DAY | | |
| | INTERVAL DAY TO HOUR | | |
| | INTERVAL DAY TO MINUTE | | |
| | INTERVAL DAY TO SECOND | | |
| | INTERVAL HOUR | | |
| | INTERVAL HOUR TO MINUTE | | |
| | INTERVAL HOUR TO SECOND | | |
| | INTERVAL MINUTE | | |
| | INTERVAL MINUTE TO SECOND | | |
| | INTERVAL SECOND | | |
| UDT1 | UDT1 | UDT2 | UDT2 |
| UDT | UDT | BYTEINT | BYTEINT SMALLINT INTEGER BIGINT DECIMAL NUMERIC REAL FLOAT DOUBLE PRECISION CHARACTER VARCHAR DATE |
| | | SMALLINT | |
| | | INTEGER | |
| | | BIGINT | |
| | | DECIMAL NUMERIC | |
| | | REAL FLOAT DOUBLE PRECISION | |
| UDT | UDT | BYTE | BYTE VARBYTE BLOB |
| | | VARBYTE | |

6: CREATE AUTHORIZATION - CREATE ERROR TABLE

| Source Input | Source Data Type of CAST Definition | Target Data Type of CAST Definition | Target Output |
|--------------|---|---|-----------------------------|
| | | BLOB | |
| | | CHARACTER | CHARACTER |
| | | VARCHAR | VARCHAR |
| | | CLOB | CLOB |
| | | | BYTEINT |
| | | | SMALLINT |
| | | | INTEGER |
| | | | BIGINT |
| | | | DECIMAL |
| | | | NUMERIC |
| | | | REAL |
| | | | FLOAT |
| | | | DOUBLE PRECISION |
| | | | DATE |
| | | | TIME |
| | | | TIME WITH TIME ZONE |
| | | | TIMESTAMP |
| | | | TIMESTAMP WITH TIME ZONE |
| | | DATE | DATE |
| | | | BYTEINT |
| | | | SMALLINT |
| | | | INTEGER |
| | | | BIGINT |
| | | | DECIMAL |
| | | | NUMERIC |
| | | | REAL |
| | | | FLOAT |
| | | | DOUBLE PRECISION |
| | | | CHARACTER |
| | | | VARCHAR |
| UDT | UDT | TIME | TIME |
| | | TIME WITH TIME ZONE | TIME WITH TIME ZONE |
| | | TIMESTAMP | TIMESTAMP |
| | | TIMESTAMP WITH TIME ZONE | TIMESTAMP WITH TIME ZONE |
| | | INTERVAL YEAR | INTERVAL YEAR |
| | | INTERVAL YEAR TO MONTH | INTERVAL YEAR TO MONTH |
| | | | INTERVAL MONTH |

| Source Input | Source Data Type of CAST Definition | Target Data Type of CAST Definition | Target Output |
|--|---|---|--|
| | | INTERVAL MONTH | INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO SECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL SECOND |
| | | INTERVAL DAY | |
| | | INTERVAL DAY TO HOUR | |
| | | INTERVAL DAY TO SECOND | |
| | | INTERVAL HOUR | |
| | | INTERVAL HOUR TO MINUTE | |
| | | INTERVAL HOUR TO SECOND | |
| | | INTERVAL MINUTE | |
| | | INTERVAL MINUTE TO SECOND | |
| | | INTERVAL SECOND | |
| ARRAY/VARRAY | ARRAY/VARRAY | XML | XML |
| BLOB | BLOB | | |
| CLOB | CLOB | | |
| DATE | DATE | | |
| DECIMAL NUMERIC | DECIMAL NUMERIC | | |
| NUMBER | NUMBER | | |
| FLOAT REAL DOUBLE PRECISION | FLOAT REAL DOUBLE PRECISION | | |
| BYTEINT SMALLINT INTEGER BIGINT | BYTEINT SMALLINT INTEGER BIGINT | | |
| NUMBER | NUMBER | | |

| Source Input | Source Data Type of CAST Definition | Target Data Type of CAST Definition | Target Output |
|---------------------------------|---|---|---------------|
| NUMERIC | NUMERIC | | |
| INTERVAL DAY | INTERVAL DAY | | |
| INTERVAL DAY TO HOUR | INTERVAL DAY TO HOUR | | |
| INTERVAL DAY TO MINUTE | INTERVAL DAY TO MINUTE | | |
| INTERVAL DAY TO SECOND | INTERVAL DAY TO SECOND | | |
| INTERVAL HOUR | INTERVAL HOUR | | |
| INTERVAL HOUR TO MINUTE | INTERVAL HOUR TO MINUTE | | |
| INTERVAL HOUR TO SECOND | INTERVAL HOUR TO SECOND | | |
| INTERVAL MINUTE | INTERVAL MINUTE | | |
| INTERVAL MINUTE TO SECOND | INTERVAL MINUTE TO SECOND | | |
| INTERVAL MONTH | INTERVAL MONTH | | |
| INTERVAL SECOND | INTERVAL SECOND | | |
| INTERVAL YEAR | INTERVAL YEAR | | |
| INTERVAL YEAR TO MONTH | INTERVAL YEAR TO MONTH | | |
| PERIOD | PERIOD | | |
| TIME | TIME | | |
| TIMESTAMP | TIMESTAMP | | |
| TIMESTAMP WITH TIME ZONE | TIMESTAMP WITH TIME ZONE | | |
| BYTE VARBYTE LONG VARBYTE | BYTE VARBYTE LONG VARBYTE | | |
| CHARACTER VARCHAR | CHARACTER VARCHAR | | |

| Source Input | Source Data Type of CAST Definition | Target Data Type of CAST Definition | Target Output |
|--------------|---|---|--|
| XML | XML | BLOB | BLOB |
| | | CLOB | CLOB |
| | | DATE | DATE |
| | | DECIMAL NUMERIC | DECIMAL NUMERIC |
| | | NUMBER | NUMBER |
| | | FLOAT REAL DOUBLE PRECISION | FLOAT REAL DOUBLE PRECISION |
| | | BYTEINT SMALLINT INTEGER BIGINT | BYTEINT SMALLINT INTEGER BIGINT |
| | | INTERVAL DAY | INTERVAL DAY |
| | | INTERVAL DAY TO HOUR | INTERVAL DAY TO HOUR |
| | | INTERVAL DAY TO MINUTE | INTERVAL DAY TO MINUTE |
| | | INTERVAL DAY TO SECOND | INTERVAL DAY TO SECOND |
| | | INTERVAL HOUR | INTERVAL HOUR |
| | | INTERVAL HOUR TO MINUTE | INTERVAL HOUR TO MINUTE |
| | | INTERVAL HOUR TO SECOND | INTERVAL HOUR TO SECOND |
| | | INTERVAL MINUTE | INTERVAL MINUTE |
| | | INTERVAL MINUTE TO SECOND | INTERVAL MINUTE TO SECOND |
| | | INTERVAL MONTH | INTERVAL MONTH |
| | | INTERVAL SECOND | INTERVAL SECOND |

| Source Input | Source Data Type of CAST Definition | Target Data Type of CAST Definition | Target Output |
|--------------|---|---|---------------------------------|
| | | INTERVAL YEAR | INTERVAL YEAR |
| | | INTERVAL YEAR TO MONTH | INTERVAL YEAR TO MONTH |
| | | PERIOD | PERIOD |
| | | TIME | TIME |
| | | TIMESTAMP | TIMESTAMP |
| | | TIMESTAMP WITH TIME ZONE | TIMESTAMP WITH TIME ZONE |
| | | BYTE VARBYTE LONG VARBYTE | BYTE VARBYTE LONG VARBYTE |
| | | CHARACTER VARCHAR | CHARACTER VARCHAR |

Cast Requirements for Teradata Utilities

Some Teradata utilities require you to define casts to permit them to process UDT data. The following list explains what those requirements are:

- Database utilities

None of the platform-based utilities has transform requirements.

The USING modifier requires you set up implicit casts for casting a predefined external data type to a UDT. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

- Teradata Tools and Utilities

The following table explains the casting requirements for various Teradata Tools and Utilities:

| Utility | Cast Requirements |
|--|--|
| FastExport | None |
| FastLoad | None |
| MultiLoad | None |
| Teradata Parallel Data Pump (TPump) | Implicit cast from a predefined external data type to a UDT. |
| Teradata Parallel Transporter: ◦ UPDATE ◦ LOAD | None |

| Utility | Cast Requirements |
|---------------------------------------|--|
| ◦ EXPORT | |
| Teradata Parallel Transporter STREAMS | Implicit cast from a predefined external data type to a UDT. |

CREATE DATABASE

Rules for Using CREATE DATABASE

The following rules apply to CREATE DATABASE:

- The maximum number of databases and users for any one system is 4.2 billion.
- When a database contains a default journal table, that journal table shares the PERMANENT storage capacity of the database with its data tables.
- If necessary, the defined PERM, SPOOL, or TEMPORARY space is changed to the highest multiple of the number of AMPs on the system less than or equal to the requested space.
- When a CREATE DATABASE statement is processed, the system places an EXCLUSIVE lock on the database being created.
- You can only specify options once.

For more information about space management, see *Teradata Vantage™ - Database Administration*, B035-1093.

PERMANENT, SPOOL, and TEMPORARY Space

You can define the maximum disk space available for various purposes for any database.

If necessary, the system changes the defined PERMANENT, SPOOL, or TEMPORARY disk space limits to the highest multiple of the number of AMPs on the system less than or equal to the requested space.

Global temporary tables require a minimum of 512 bytes from the PERM space of their containing database or user. This space is used for the table header of the table.

You should limit the amount of SPOOL space assigned to databases to a level that is appropriate for the workloads undertaken by its users. While some users require more spool space than others, you should minimize the allotment of spool for any database to a size that is large enough to permit result sets to be handled and space management routines such as MiniCylPack to run when disk space becomes low, but small enough to force spool overflow errors when runaway queries resulting from Cartesian products and other common SQL coding errors occur.

See *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - Database Administration*, B035-1093 for more information about user space.

Local Journaling

LOCAL single AFTER image journals are supported analogously to single BEFORE image journals as follows:

- The DROP DATABASE privilege is required to create or drop LOCAL single AFTER image journals. This is the same privilege required to perform the analogous operation on single BEFORE image journals.
- LOCAL single AFTER image journaling is restricted to non-fallback data tables.

Teradata MultiLoad, Teradata Parallel Transporter, and Teradata FastLoad are impacted by the use of LOCAL journaling.

Activating Permanent Journaling

If you specify only `DEFAULT JOURNAL TABLE = table_name`, then the system creates a journal table, but does not activate it.

To activate the permanent journal, you must also specify either the `AFTER JOURNAL` journaling option or the `BEFORE JOURNAL` option or both.

This action causes permanent journaling to be activated for all tables created in this database afterward.

To determine which tables in your system are journal tables, use the following query:

```
SELECT DBC.dbase.databasename (FORMAT 'X(15)'),
       DBC.tvm.tvpname (FORMAT 'X(25)')
FROM DBC.tvm, DBC.dbase
WHERE DBC.dbase.databaseid=DBC.tvm.databaseid
AND   DBC.tvm.tablekind='j'
ORDER BY 1,2;
```

To determine which databases and users in your system currently have a default journal table defined for them, use the following query:

```
SELECT d.databasename (TITLE 'Database'), TRIM(dx.databasename)
||'. '||TRIM(t.tvpname)(TITLE 'Journal')
FROM DBC.dbase AS d, DBC.TVM AS t, DBC.dbase AS dx
WHERE d.journalid IS NOT NULL
AND   d.journalid <> '00'xb
AND   d.journalid = t.tvmid
AND   t.databaseid = dx.databaseid
ORDER BY 1;
```

To determine which tables in your system are currently being journaled, use the following query:

```
SELECT TRIM(Tables_DB)||'. '||TableName (TITLE 'Table',
CHARACTER(26)), 'Assigned To' (TITLE ' '), TRIM(journals_db)
||'. '||JournalName (TITLE 'Journals', CHARACTER(26))
FROM DBC.journals
ORDER BY 1,2;
```

You can also determine which tables in your system are currently being journaled using the following query that has a somewhat different syntax:

```

SELECT TRIM(d.databasename)||'.'||TRIM(t.tvname) (FORMAT
'x(45)',TITLE 'Table'),TRIM(dj.databasename)
||'.'||TRIM(tj.tvname) (TITLE 'Journal')
FROM DBC.TVM AS t, DBC.TVM AS tj, DBC.dbase AS d, DBC.dbase AS dj
WHERE t.journalid IS NOT NULL
AND t.journalid <> '00'xb
AND t.journalid = tj.tvmid
AND d.databaseid = t.databaseid
AND dj.databaseid = tj.databaseid
ORDER BY 1;

```

Related Information

See [Activating Permanent Journaling](#) for information about activating permanent journaling for existing tables.

Also see the following topics:

- [ALTER TABLE \(Basic Table Parameters\)](#)
- [CREATE DATABASE](#)
- [MODIFY DATABASE](#)
- [MODIFY USER](#)

CREATE ERROR TABLE

Rules and Restrictions for Error Tables

- You can define only one error table per data table.

A single error table-to-data table relationship simplifies error management and allows the error table to be transparent to users because they do not need to know the name of the error table for a particular data table. You can create or drop an error table and display its structure indirectly by referencing the data table name in place of the error table name.

- You can define an error table for a column-partitioned table.

The system creates the error table as it would for a non-column-partitioned table except that the error table is a PI table with partitioning.

- You can define an error table for a foreign table.

Vantage logs format, syntax, or data conversion errors detected while parsing or accessing the data in the foreign table.

Vantage grants the user the INSERT privilege on the error table when the user is granted the SELECT privilege on the foreign table.

- You cannot use an ALTER TABLE request to change any property of an error table.
- The database maintains the compatibility between data tables and their associated error tables by disallowing any of the following ALTER TABLE operations when an error table is defined for a data table:

- Adding or dropping columns from the data table.
- Changing the primary index column set of the data table.

You can alter the partitioning of a data table that has an error table.

- Changing the fallback protection of the data table.
- Any ALTER TABLE request on the error table.

- You must create your error tables on base tables, not on views. creating an error table on a view is not permitted.

A security issue arises if you create an error table over all the base table columns in a view. Base table columns that were previously hidden by the view then become visible to the creator of the error table.

Although you could create the error table in a database the LOGGING ERRORS user does not have access to, that prevents the LOGGING ERRORS user from being able to perform error recovery.

If an error table is created with view columns only, error recovery using error table rows could become difficult, especially when errors come from a non-view column.

- You must create an error table before you can log errors against its associated data table.
- When you create an error table, error logging is enabled by default.

To disable or re-enable error logging, use the SET QUERY_BAND statement with the ForeignTableErrorsData flag. For information about SET QUERY_BAND, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. For information about ForeignTableErrorsData, see *Teradata Vantage™ - Database Utilities*, B035-1102.

- The base data table for which you are creating an error table cannot have more than 2,048 columns. The newly created error table for this data table adds another 13 error columns to this total, bringing the maximum number of columns in an error table to 2,061. See [System-Defined Attributes for Error Table-Specific Columns](#).

- If the size of an error table row (data table row plus the 13 ETC_ columns) exceeds the system row size limit, the CREATE ERROR TABLE request returns an error message to the requestor.
- An error table can be contained in either the same database or user as its associated data table or in a different database or user.
- Error tables are MULTiset tables by default to avoid the expense of duplicate row checks. You cannot change an error table to be a SET table.
- Error tables have the same fallback properties as their associated data table.
- The database copies the data types of data table columns to the error table.

The system does not copy any other column attributes from the data table to the error table.

- Error tables have the same primary index as their associated data table if the table has a primary index and is not column partitioned.

Because all error tables are MULTiset tables by definition, if the primary index for its associated data table is defined as a UPI, the system converts its definition to a NUPI for the error table.

- If the data table associated with an error table has a partitioned primary index, the system does *not* copy its partitioning to the error table.

Error tables are always nonpartitioned tables by default and cannot be changed to partitioned tables.

For information about partitioned primary indexes, see [Partitioned and Nonpartitioned Primary Indexes](#) and *Teradata Vantage™ - Database Design*, B035-1094.

- You can define an error table for a nonpartitioned NoPI table.
- You cannot define secondary indexes for an error table using the CREATE ERROR TABLE statement, nor does the system copy secondary indexes defined on the data table to its error table.

However, you can use the CREATE INDEX statement to add secondary indexes to an existing error table to facilitate scanning its captured data for analysis (which you cannot do for a Teradata MultiLoad error table). See [CREATE INDEX](#).

You must drop any of the secondary indexes you have defined on the error table before you perform an INSERT ... SELECT or MERGE operation that specifies error logging on the data table associated with the error table.

If you do not drop all secondary indexes on an error table, the system returns an error when you attempt to perform a bulk loading operation into the data table.

- You cannot define triggers, join indexes, or hash indexes on an error table.

- Error tables do not handle batch referential integrity violations.

Because batch referential integrity checks are all-or-nothing operations, a batch RI violation results in the following type of session mode-specific abort and returns an error message to the requestor:

| IF this session mode is in effect ... | THEN the erring ... |
|---------------------------------------|---------------------|
| ANSI | request aborts. |
| Teradata | transaction aborts. |

- The system rejects attempts to insert data rows that cause local target table errors. Such rows are instead inserted into the error table. Once such errors have been corrected, you can reload the rows either directly from the error table or from the original source table.
- As a general rule, the system does not handle error conditions that do not allow useful recovery information to be logged in the error table.

These errors typically occur during intermediate processing of input data before they are built into a row format that corresponds to that of the target table.

The database detects such errors before the start of data row inserts and updates. The errors include the following:

- UDT, UDF, and table function errors
- Version change sanity check errors
- Non-existent table sanity check errors
- Down AMP request against non-fallback table errors
- Data conversion errors.

However, conversion errors that occur during data row inserts and merge updates are handled as local data errors.

These kinds of errors result in the following type of session mode-specific abort and returns an error message to the requestor:

| IF this session mode is in effect ... | THEN the erring ... |
|---------------------------------------|---------------------|
| ANSI | request aborts. |
| Teradata | transaction aborts. |

However, the system *does* preserve all error table rows logged by the aborted request or transaction and does not roll them back.

- In addition to the previously listed errors, the system does not handle the following types of errors, which result in the following type of session mode-specific abort and returns an error message to the requestor:
 - Out of PERM or spool space errors

- Duplicate row errors in Teradata session mode (because the system rejects duplicate rows silently in Teradata mode)
- Trigger errors
- Join Index maintenance errors
- Identity column errors
- Implicit-USI violations.

When a table is created with a primary key that is not also the primary index, the system implicitly defines a USI on that primary key.

A violated implicit USI on a primary key cannot be invalidated because the system does not allow such a USI to be dropped and then recreated later

These kinds of errors result in the following type of session mode-specific abort and returns an error message to the requestor:

| IF this session mode is in effect ... | THEN the erring ... |
|---------------------------------------|---------------------|
| ANSI | request aborts. |
| Teradata | transaction aborts. |

- When an error type that is excluded from logging occurs, such as a USI or referential integrity error, the system causes the following type of session mode-specific abort and returns an error message to the requestor:

| IF this session mode is in effect ... | THEN the erring ... |
|---------------------------------------|---------------------|
| ANSI | request aborts. |
| Teradata | transaction aborts. |

In this case, the database rolls back *only* the target table rows, and only after the batch load operation has completed to enable the collect of all rows that cause errors. The database preserves existing error table rows and does not roll them back.

- The system sets other general table properties, such as journaling, checksums, datablock size, and so on to the defaults applied to CREATE TABLE statements that do not explicitly specify these settings.
- You *can* define an error table for a queue table.
- You *cannot* specify the QUEUE attribute when you create that error table.

In other words, while an error table itself cannot be a queue table, a queue table can have an error table associated with it.

- You can submit DML requests against an error table, just as you can for a Teradata MultiLoad error table, and you can make error corrections directly on the data row images stored in the error table.

After you correct errors in the error table rows, you can then reload the modified rows into the target table directly from the error table instead of reloading them from the staging table.

- CREATE ERROR TABLE requests add the definitions of row-level security constraint columns to the error table if row-level security constraints are defined for the base table. This is true unless you specify the NO RLS option.

The NO RLS option suppresses the generation of row-level security constraint columns from the base table to the error table.

- When rows are inserted into an error table, the database does not enforce row-level security constraints because such constraints would have been enforced during the MERGE or INSERT ... SELECT operation that generated the error table row.

Error Logging Limits

By default, the maximum number of errors to log is 10. To change the maximum number, use the SET QUERY_BAND statement with the ForeignTableErrorsLimit field. For information about SET QUERY_BAND, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. For information about ForeignTableErrorsLimit, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Function of Error Tables

Data warehousing environments require frequent updates of their underlying databases, and the overall data load strategy must include scheduled incremental loads, or small batches, to be performed throughout the day.

The intent of the LOGGING ERRORS option with INSERT ... SELECT and MERGE batches is to complement the FastLoad and MultiLoad batch load utilities that handle larger volumes of data and more complex loads with multiple target tables or DML operations, and the continuous load utility, Teradata Parallel Data Pump.

By capitalizing on incremental batch loads on a continuous basis using INSERT ... SELECT or MERGE requests or both, you can update your data warehouse as frequently as is necessary, using a mixture of data loading strategies. Furthermore, such batch requests are not restricted by the inability to load data into tables defined with LOB columns (note, however, that error logging does not support LOB data. LOBs in a source table are not copied to an error table; instead, they are represented in the error table as nulls.), unique secondary indexes, referential integrity constraints, join indexes, hash indexes, and triggers that can make bulk data loading into active data warehouse tables a complicated operation (see *Teradata® FastLoad Reference*, B035-2411 and *Teradata® MultiLoad Reference*, B035-2409 for a complete list and description of the database data table restrictions they must observe).

Although Teradata Parallel Data Pump performs batch loads, the INSERT ... SELECT and MERGE batch options and the Teradata Parallel Data Pump option can serve very distinct situations. For example, when logged on sessions are near their full capacity, the INSERT ... SELECT or MERGE batch options are generally preferable to the Teradata Parallel Data Pump option. Conversely, when the specific order of updates is important, Teradata Parallel Data Pump are generally preferable.

The purpose of error tables is to provide a substrate for complex error handling of the following kinds of batch loads:

- Insert errors that occur during an INSERT ... SELECT operation
- Insert and update errors that occur during a MERGE operation

When an insert or update error occurs on an index or data conversion, the transaction or request containing the erring statement does not roll back, but instead runs to completion, logging the rows that caused errors in an error table.

Data errors that cause secondary index or referential integrity errors *do* cause the transaction or request containing the erring statement to roll back. You must then fix the errors that caused the secondary index or referential integrity violations before you can again load those rows into the database.

Error tables are one component of a batch system that includes extensions to MERGE and INSERT ... SELECT batch load operations to support complex error handling (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146). These DML statement extensions permit you to specify error logging for INSERT ... SELECT and MERGE operations. If you specify error logging, the system logs error conditions and captures copies of the data rows that cause them in an error table.

System-Defined Attributes for Error Table-Specific Columns

The following table describes the characteristics of the system-defined and generated error table-specific columns. The ETC prefix indicates an Error Table Column.

| Column Name | Column Definition | Data Type and Attributes |
|---------------|---|---------------------------|
| ETC_DBQL_QID | Query ID for the Database Query Log. The system sets up a time-based query ID to uniquely identify all error rows for a given request regardless of whether DBQL is enabled. The query ID is incremented for each new request. | DECIMAL(18,0) NOT NULL |
| ETC_DMLType | A code for the type of DML request that erred, as follows. <ul style="list-style-type: none"> • D represents a Delete operation. • I represents an Insert operation. Indicates either an INSERT ... SELECT error or a MERGE insert error. • U represents an Update operation. Indicates either a MERGE update error or a join UPDATE error. | CHARACTER(1) |
| ETC_ErrorCode | Either a DBC error code or a value of 0. A row with ETC_ErrorCode = 0 is a special marker row that confirms the request completed successfully, but logged one or more errors in a LOGGING ERRORS operation. A row with a value for ETC_ErrorCode other than 0 indicates that the request aborted because of the recorded DBC error code and logged one or more errors in a LOGGING ERRORS operation. The system does not write a marker row for a request that completes successfully without logging any errors. | INTEGER NOT NULL |

| Column Name | Column Definition | Data Type and Attributes |
|-----------------|--|-------------------------------|
| ETC_ErrSeq | <p>Error sequence number.</p> <p>The value provides a numeric sequence that is easier to refer to for error analysis and recovery purposes than the timestamp values in ETC_TimeStamp.</p> <p>The value stored for ETC_ErrSeq begins the sequence with 1 for the first error of a given request, and increments by 1 for each subsequent error recorded.</p> <p>The ETC_ErrSeq value does not reflect the true processing sequence of index rows that cause USI or referential integrity errors because, for example, the system assigns the sequence numbers for referential integrity errors when it informs the AMPs that own the data rows of the index violations, not when it first detects those violations.</p> | INTEGER NOT NULL |
| ETC_IndexNumber | Contains either the ID for the index that caused a USI or referential integrity violation or is set null. | SMALLINT |
| ETC_IdxErrType | <p>Indicates if the index ID stored in ETC_IndexNumber refers to an USI violation or a referential integrity violation.</p> <ul style="list-style-type: none"> • Null indicates that ETC_IndexNumber is null. • r indicates a parent-delete referential integrity violation. • R indicates a child-insert referential integrity violation. • U indicates a unique secondary index violation. <p>Because ETC_IdxErrType is defined with the NOT CASESPECIFIC attribute, both child and parent referential integrity violations are selected when you specify a predicate condition of WHERE ETC_IdxErrType = 'R'.</p> <p>To work around this problem, make your predicate conditions case specific, as follows.</p> <ul style="list-style-type: none"> • To select only child-insert errors, specify the WHERE condition etc_idxerrtype (CS) = 'R' • To select only patient-delete errors, specify the WHERE condition etc_idxerrtype (CS) = 'r'. <p>This provides a simple and reliable check for child versus parent referential integrity errors, and for parent versus child tables identified by ETC_RTableId.</p> | CHARACTER(1) NOT CASESPECIFIC |
| ETC_RowId | <p>Records the rowIDs or row key values specified in the following table for the listed error conditions.</p> <ul style="list-style-type: none"> • If the insert row is a duplicate of an existing row in the target table, the value is the rowID of the target table row. • If the insert row fails a CHECK constraint, the value is the row key of the insert row with 0s in its uniqueness portion. • If an insert row build fails because an internal error such as a divide by zero error occurred, the value is the row key of the insert row with 0s in its uniqueness portion. • If the update row of a MERGE request is a duplicate of an existing row in the target table, the value is the rowID of the original image of the target table row being updated. | BYTE(10) |

| Column Name | Column Definition | Data Type and Attributes |
|---------------|---|--------------------------|
| | <ul style="list-style-type: none"> If the update row fails a CHECK constraint, the value is the rowID of the original image of the target table row being updated. If an update row build for a MERGE request fails because of an internal error such as a divide by zero error, the value is the rowID of the target table row being updated. If multiple source rows of a MERGE request are attempting to update the same target table row (this is not permitted by the ANSI SQL:2011 standard), the value is the rowID of the target table row. For any other operational error, the system stores a null. <p>This information is intended for Teradata Services personnel.</p> | |
| ETC_TableId | Identifies the table associated with the value stored in ETC_FieldId. | BYTE(6) |
| ETC_FieldId | <p>Stores the id of the column that caused an error condition, which is the field ID stored in DBC.TVFields.</p> <ul style="list-style-type: none"> For a composite key or index violation error, the value is the column ID of the first column in the composite key or index. For a column or constraint violation, the value is the column ID of the column returned by the system for a constraint violation. For a duplicate row error, the system stores a null. <p>If a table constraint contains two columns, the system stores the ID of the column on the right-hand side of the constraint expression. For example, a CHECK (y>x) error captures the field ID of column x.</p> | SMALLINT |
| ETC_RITableId | <p>Identifies the other table involved in an referential integrity violation.</p> <ul style="list-style-type: none"> For a child-insert error, the value is the referencing parent. For a parent-delete error, the value is the referencing child. | BYTE(6) |
| ETC_RIFieldId | <p>Identifies the column in the table associated with an RI violation.</p> <ul style="list-style-type: none"> For a parent table, the value identifies the missing UPI or USI column referenced by an inserted child row. For a child table, the value identifies the foreign key column that referenced the UPI or USI key in a deleted parent row. <p>For composite keys, <i>ETC_RIFieldId</i> identifies only the first column defined for the key, which is often enough information to identify the key uniquely.</p> | SMALLINT |
| ETC_TimeStamp | Indicates the time the system detects an error, not the time the system writes the error row to the error table. | TIMESTAMP(2) NOT NULL |
| ETC_Blob | <p>Not currently used.</p> <p>This column consumes space in the retrieval of rows from an error table.</p> <p>If an error table has a large number of data table columns, the inclusion of this column in the select list might cause the query to reach an internal database limitation and return a 'Row size or Sort Key size overflow' error.</p> <p>If this happens, drop the <i>ETC_Blob</i> column from the select list.</p> | BLOB |

| Column Name | Column Definition | Data Type and Attributes |
|-------------|---|--------------------------|
| | The size limit for <i>ETC_Blob</i> is 2 MB, which is large enough to store a maximum-sized memory segment associated with the step that produces a data conversion error. | |

Error Table Row Size Minimization

To minimize the size of error table rows, the system stores only the IDs of fields and users, not their names. To determine the names associated with the stored field IDs and user IDs, you can run queries against the *DBC.Tables3VX* view.

For example,

```
SELECT TableName, FieldName, ETC_ErrorCode
FROM DBC.Tables3VX, ET_myerrtbl
WHERE TableId = ETC_TableId
AND FieldId = ETC_FieldId
AND ETC_DBQL_QID = 385520516506583068;
```

ET_myerrtbl

The default system-generated name of the error table for base table *myerrtbl*, for which you want to retrieve table and column names.

385520516506583068

The unique DBQL query ID for this request.

See *Teradata® Parallel Data Pump Reference*, B035-3021, *Teradata® FastLoad Reference*, B035-2411, and *Teradata® MultiLoad Reference*, B035-2409 for the definitions of the error tables generated by those bulk loading utilities.

Dictionary Support for Error Tables

The system provides one dictionary table, *DBC.ErrorTbls*, and three views, *DBC.ErrorTblsV*, *DBC.ErrorTblsVX*, and *DBC.Tables3VX*, to support error tables.

DBC.ErrorTbls maintains the relationships between error tables and their data tables rather than the table header.

You can query the *DBC.ErrorTblsV* and *DBC.ErrorTblsVX* views to report error table names and their associated data table and database names.

You can query the *DBC.Tables3VX* view to report the database, table, and column names associated with the table IDs and column IDs stored in error table rows.

See *Teradata Vantage™ - Data Dictionary*, B035-1092 for the definitions and usage details about *DBC.ErrorTbls*, *DBC.ErrorTblsV*, *DBC.ErrorTblsVX*, and *DBC.Tables3VX*. See [Error Table Row Size Minimization](#) for an example of how to use the *DBC.Tables3VX* view.

Usage Considerations for the Error Table Columns

The following table supplies additional usage considerations for Error Table information beyond that listed in [System-Defined Attributes for Error Table-Specific Columns](#).

| Column | Usage Notes |
|--------------|---|
| ETC_DBQL_QID | The system sets up a time-based query ID regardless of whether DBQL is enabled, and increments it for each new request. You can use the column <i>ETC_DBQL_QID</i> to uniquely identify all error rows for a request. The latter is less reliable because in a restart, the session numbers begin incrementing from the same base value. As a result, two different requests might have the same host, session, and request numbers if there are restarts. |
| ETC_ErrSeq | You can use <i>ETC_ErrSeq</i> column to determine the order of row updates and inserts for a MERGE operation. <i>ETC_ErrSeq</i> provides a numeric sequencing that is easier to refer to for error analysis and recovery purposes than the timestamps in <i>ETC_TimeStamp</i> . |

If the data row size plus the fixed error table columns exceeds the system row size limit, an ANSI session mode request aborts and returns an error message to the requestor, while a Teradata session mode transaction containing the erring CREATE ERROR TABLE request aborts and returns an error message to the requestor.

Using Error Table Data to Isolate Problems

Use the following general procedure to isolate problems using error table data:

1. Query the error table to retrieve information on the errors logged for a user in an error table named *ET_t* and the DBQL query ID returned in the error or warning message:

```
SELECT ETC_ErrorCode, FieldName, ETC_IndexNumber, ETC_IdxErrType
FROM ET_t, DBC.Tables3VX
WHERE ETC_DBQL_QID = 123456789012345678
AND ETC_TableID = TableID;
```

2. Look at each error code and column value related to the error and determine if any action needs to be taken.

The following error table columns are more useful for troubleshooting by Teradata Services personnel than for end users:

- *ETC_RowId*
- *ETC_ErrSeq*

Handling Logged Reference Index and Unique Secondary Index Errors

If any Reference Index or USI errors are logged during an INSERT ... SELECT or MERGE operation, delete the data rows that caused the index violations from the source (staging) table and rerun the load request.

CREATE FUNCTION - CREATE GLOBAL TEMPORARY TRACE TABLE

These topics provide supplemental usage information about selected SQL DDL statements alphabetically from CREATE FUNCTION through CREATE GLOBAL TEMPORARY TRACE TABLE.

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE FUNCTION and REPLACE FUNCTION (External Form)

UDF Parameter Styles

Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

The following three parameter styles are all valid for external functions.

| Style | Description |
|------------|---|
| SQL | <p>Uses indicator variables to pass arguments. This is the default parameter style for all UDFs.</p> <p>As a result, you can always pass nulls as inputs and return them in results. If a row-level security policy constraint permits nulls, then its parameter style must be SQL.</p> <p>If the function executes a row-level security policy constraint, each input and output parameter for the function has a second indicator parameter. The following indicator parameter values apply.</p> <ul style="list-style-type: none"> • If the indicator parameter value is -1, its corresponding value parameter is null. • If the indicator parameter value is 0, its corresponding value parameter is not null. <p>No other values are valid for an indicator parameter in a row-level security policy constraint UDF.</p> |
| TD_GENERAL | <p>Uses parameters to pass arguments. Can neither be passed nulls nor return nulls. If a row-level security policy constraint does not permit nulls, then its parameter style must be TD_GENERAL.</p> <p>If the function executes a row-level security policy constraint, there are no indicator parameters.</p> |
| JAVA | <p>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. PARAMETER STYLE JAVA <i>must</i> be specified for all Java functions. A row-level security policy constraint function cannot have a parameter style of JAVA.</p> |

Downloadable UDF Samples

For downloadable samples of Teradata application software, see:

<https://support.teradata.com/community>

<https://www.teradata.com/Developer>

Relationship Among UDFs, Table UDFs, Methods, and External Procedures

UDFs, table UDFs, methods, and external procedures are specific variations of one another and share most properties in common. The generic term used to describe all these is *external routine*.

For more information, see the following topics:

- [CREATE FUNCTION \(Table Form\)](#)
- [CREATE METHOD](#)
- [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#)
- [REPLACE METHOD](#)

Naming Conventions: Avoiding Name Clashes Among UDFs, UDMs, and UDTs

For UDF, UDM, and UDT names:

- The DatabaseID, TVMName column pair must be unique within the DBC.TVM table. Use the DBC.Tables2 view to view rows from DBC.TVM.
- The signature of the *database_name.routine_name(parameter_list)* routine must be unique.

UDFs, UDMs, and UDTs can have the same SQL names as long as their SPECIFIC names and associated routine signatures are different. In the case of UDTs, the SPECIFIC name reference is to the SPECIFIC names of any method signatures within a UDT definition, not to the UDT itself, which does not have a SPECIFIC name.

The value of *database_name* is always SYSUDTLIB for UDFs associated with UDTs, including UDFs used to implement the following functionality on behalf of a UDT:

- Casts
- Orderings
- Transforms

The Database ID column entry is always the same. The name uniqueness is dependent on the TVMName column value only.

TVMName Entry for UDTs and UDMs

The following describes the TVMName entry for UDTs and UDMs.

UDTs created by a CREATE TYPE request have a SPECIFIC name that is system-generated based on the specified UDT name. For information on object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The database generates the SPECIFIC name by truncating any UDT names that exceed the permitted number of characters.

When you submit a CREATE TYPE statement, the system automatically generates a corresponding UDF to obtain a UDT instance.

The SPECIFIC name of a UDT, UDM or UDF must be unique to avoid name clashes.

| Type of External Routine | SPECIFIC Name |
|--|--|
| UDT | always its UDT name. |
| <ul style="list-style-type: none"> UDM UDF | <p>user-defined.</p> <p>There are two exceptions to this rule for structured types:</p> <ul style="list-style-type: none"> System-generated observer methods. System-generated mutator methods. <p>The SPECIFIC names for these are always the same as the names of the structured UDT attributes on which they operate.</p> <p>Note that for UDMs, the SPECIFIC name is defined in its signature within its associated UDT, not within the CREATE METHOD statement.</p> |

The signatures of external routines must be unique. The following rules apply:

- For every UDT you create, the system generates a UDF with the following signature: SYSUDTLIB.UDT_Name().
No other UDF can have this signature.
- When you create a UDM, the system treats it as a UDF for which the data type of its first argument is the same as the UDT on which that UDM is defined.

For example, a UDM named area(), defined on the UDT named circle, would have the signature SYSUDTLIB.area(circle). It follows that there can be no other UDF with this same signature.

For a single database (SYSUDTLIB) Teradata UDT environment, the following rules apply:

- A UDT and a SYSUDTLIB UDF with no parameters cannot have the same name.
- A method for a structured UDT cannot have the same name as any of the attributes for that type if the signature of the method and the signature of either the observer or mutator methods for the attribute match.

You must define a transform group for each UDT you create. Because the system creates a transform group for you automatically when you create a distinct UDT, you cannot create an additional explicit transform group without first dropping the system-generated transform. The names of UDT transform groups need not be unique, so you can use the same name for all transform groups.

The names of transform groups are stored in DBC.UDTTransform.

The system adds SPECIFIC method names to the TVMName column in DBC.TVM for:

- Observer and mutator methods, which are auto-generated for structured type attributes.
- Instance methods and constructor methods created by CREATE TYPE, ALTER TYPE, or CREATE METHOD statements where the coder does not specify a SPECIFIC method name.

A SPECIFIC name of up to 28 characters is generated based on the UDT and attribute names.

The SPECIFIC name is generated as the concatenation of the following elements in the order indicated:

1. The first 8 characters of the UDT name.
2. A LOW LINE (underscore) character.
3. The first 10 characters of the for observer attribute name, mutator attribute name, instance method name, or constructor method name, as appropriate.
4. A LOW LINE (underscore) character.
5. The last 8 HEXADECIMAL digits of the routine identifier assigned to the observer, mutator, instance, or constructor method name, as appropriate.
6. A character sequence is appended, _O for observer, _M for mutator, _R for instance, or _C for constructor, appropriate.

The remaining characters, up to the 30th character, are filled with SPACE characters.

Usage Restrictions for User-Defined Functions

UDF usage is restricted as follows:

- 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 external procedure object even if the new object name contains only single-byte characters. Instead, use a multibyte session character set.
- You cannot perform a CREATE FUNCTION or REPLACE FUNCTION request from an embedded SQL application.
- A UDF cannot reference a recursive view.
- A row-level security policy can only be a scalar function.
- You cannot call a row-level security policy function from a DML request.

Row-level security policy UDFs can only be specified for columns in a CREATE TABLE or ALTER TABLE request.

- Non-Java UDFs can specify both distinct and structured UDTs as input and return parameters.
- If you specify a UDT as either an input parameter or as the function result, the current user of the function must have either the UDTUSAGE privilege on the SYSUDTLIB database or the UDTUSAGE privilege on the specified UDT.
- UDFs that are used as cast routines, ordering routines, or transform routines must be contained within the SYSUDTLIB database.
- Java UDFs cannot be called from within a Java external procedure.
- Any nondeterministic elements referenced in the RETURN expression of a UDF can be replaced by values predefined by Teradata Unity. For more information about Teradata Unity, see the Teradata Unity documentation.

Restrictions On Replacing UDFs That Implement Operations On UDTs

The following restrictions apply to replacing UDFs that implement operations on UDTs:

- A UDF used as a cast, ordering, or transform routine must be created in the *SYSUDTLIB* database.
- A UDF used to implement either ordering or transform functionality for a UDT can only be replaced if all the following conditions are satisfied:

- The REPLACE FUNCTION specification must match exactly with the existing method specification in the dictionary.

This means that the function name, parameter list, method entry point name within the EXTERNAL clause, and so on must all match.

- The execution mode for the UDF being replaced must be EXECUTE PROTECTED.

If the function is currently set to EXECUTE NOT PROTECTED mode, then you must perform an ALTER FUNCTION statement to change the mode to EXECUTE PROTECTED before you can perform the REPLACE FUNCTION statement.

Unless all of these conditions have been met when you submit the REPLACE FUNCTION statement, the system returns an error to the requestor.

- A UDF that implements casting functionality for a UDT can only be replaced by performing the following SQL statements in the order indicated:

1. DROP CAST
2. REPLACE FUNCTION
3. CREATE CAST

You cannot use REPLACE FUNCTION by itself to perform this task. An attempt to replace a casting function directly using REPLACE FUNCTION aborts and returns an error to the requestor. See DROP CAST in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. See [CREATE CAST](#) and [REPLACE CAST](#).

Protected and Unprotected Execution Modes

By default, all UDFs are created to run in protected mode. Protected and secure modes are states in which each instance of a UDF runs in a separate process. The difference between a protected mode server and a secure mode server is that a protected mode server process runs under the predefined OS user *tdatuser*, while a secure server process runs under the OS user specified by the UDF in its EXTERNAL SECURITY clause. The two processes are otherwise identical.

This is done to protect the system from many common programming errors such as non-valid pointers, corrupted stacks, and illegal computations such as divide-by-zero errors that would otherwise crash the database, produce problems with memory leakage, or cause other potentially damaging results.

These problems all cause the database to crash if they occur in unprotected mode. UDFs can also cause the database to crash in protected mode if they corrupt the shared data areas between the database and the protected mode UDF.

Protected mode is designed to be used for the following purposes only:

- Testing all UDFs that are in development.
- Running Java UDFs.

Java UDFs must be run in protected mode at all times. Because of this, they are slower than the equivalent C or C++ functions.

- Running any UDFs that cause the OS to consume system resources. This includes anything that causes the OS to allocate system context, including open files, pipes, semaphores, tokens, threads (processes), and so on.

Do *not* use protected mode for production-ready C or C++ functions that do not make OS system calls. This is because protected mode places additional processing burdens on the system that often result in performance degradation, while UDFs in unprotected mode run in the context of the AMP worker task that is already being used by that query step. If you run C or C++ UDFs in protected mode, they run more slowly.

The best practice is to develop and test your UDFs on a non-production test system. You should run any newly created UDF several hundred times, both to ensure that it does not crash the system and to determine any performance issues that could be avoided by alternate function design and better coding techniques. Poorly designed UDFs can be a significant drain on system performance.

Table cardinality is an important factor in UDF performance, so any tests run against smaller test tables on a non-production system might not scale to the same level of performance when run against significantly larger tables in production databases.

You can use the `cufconfig` utility to expand the number of protected mode servers from the default value of 2 to a maximum of 20 per AMP or PE vprocs. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147. The minimum is 0.

Protected mode servers consume disk resources for C and C++ functions as follows:

$$\text{Total swap space required} = \frac{\text{Number of vprocs}}{\text{node}} \times \frac{\text{Number of protected mode servers}}{\text{vproc}} \times \frac{256 \text{ KB}}{\text{server}}$$

In unprotected mode, a UDF is called directly by the database rather than running as a separate process. You should only alter a new C or C++ function that does not require the OS to allocate system context to run in unprotected mode after you have thoroughly tested and evaluated its robustness and performance impact. Once the newly created CPU-operations-only C or C++ UDF has passed your quality measures and is ready to be put into production use, you should alter it to run in unprotected mode.

Each Java server for UDFs requires roughly 30 MB of memory for swap space, and there can be two such Java servers per node. A Java UDF multithreaded server for non-secure mode Java UDFs uses a minimum of an additional 30 MB (the amount required can be larger, depending on the size of the JARs for a user), so each node requires approximately 100 MB of swap space if all server flavors are used.

UDF Handling of SQL Result Codes

Because UDFs are user-written applications, it is the responsibility of the user-written code to generate all success, warning, and exception messages.

The ANSI SQL:2011 standard defines a return code variable named `SQLSTATE` to accept status code messages. All condition messages are returned to this variable in a standard ASCII character string format.

All `SQLSTATE` messages are 5 characters in length. The first 2 characters define the message class and the last 3 characters define the message subclass.

For example, consider the `SQLSTATE` return code '22021'. The class of this message, 22, indicates a data exception condition. Its subclass, 021, indicates that a character not in the defined repertoire was encountered.

Be aware that SQL warnings do *not* abort a request, while SQL exceptions *do* abort a request.

Run UDFs in protected mode. Otherwise, the database might not trap errors that the function generates and a system error may occur. For more information, see [When to Specify Unprotected Mode](#).

You should ensure that your UDFs always return valid `SQLSTATE` codes. The database does not map `SQLSTATE` values returned by user-defined functions to their equivalent `SQLCODE` values. All `SQLSTATE` codes generated by UDFs are explicitly mapped to database messages as indicated by the following table:

| UDF SQL Return Code | Error Message | Description |
|---------------------|---------------|---|
| Warning | 7505 | *** Warning 7505 in dbname.udfname: SQLSTATE 01Hxx: < user function message text > |
| Exception | 7504 | *** Error 7504 in dbname.udfname: SQLSTATE U0xxx: < user function message text > |

The string represented by <**user function message text**> is a user-defined error message generated by the UDF.

The text represented by x characters in the `SQLSTATE` subclass is also assigned by the user-written UDF code. All characters must be either digits or uppercase Latin characters.

For more information about coding UDFs, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

For more information about SQL exception and warning codes, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

For more information about `SQLSTATE` and `SQLCODE`, see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

Differences Between CREATE FUNCTION and REPLACE FUNCTION

A function can be created or replaced using the same syntax except for the keywords CREATE and REPLACE.

If you specify CREATE, the function must not exist.

If you specify REPLACE, you can either create a new function or replace an existing function with the following restriction: if the function to be replaced was originally created with a specific function name, then that specific function name must be used for the REPLACE FUNCTION statement.

When you replace an existing function, the replaced function does not retain the EXECUTE NOT PROTECTED attribute if one had previously been set using the ALTER FUNCTION statement. See [ALTER FUNCTION \(External Form\)](#).

The advantage to using REPLACE is that you can avoid having to grant the EXECUTE privilege again to all users who already had that privilege on the function.

Refer to the topics below for important restrictions on replacing UDFs that perform UDT-related functionality.

Replacing an Ordering or Transform UDF

A UDF that is being used to implement either the ordering or transform functionality for a UDT can only be replaced if the following conditions are met:

- The REPLACE FUNCTION specification must match exactly with the existing method specification in the dictionary.

The function name, parameter list, method entry point name within the EXTERNAL clause, and so on must all match.

- The execution mode for the UDF being replaced must be set to EXECUTE PROTECTED.

If the UDF is currently set to run in the EXECUTE NOT PROTECTED mode, then you must submit an ALTER FUNCTION statement to switch the mode to EXECUTE PROTECTED before you can submit the REPLACE FUNCTION statement. See [ALTER FUNCTION \(External Form\)](#).

Replacing a Casting UDF for a UDT

A UDF that is being used to implement casting functionality for a UDT can only be replaced using the following procedure:

1. DROP CAST
2. REPLACE FUNCTION
3. CREATE CAST

An attempt to directly replace a casting function using REPLACE FUNCTION results in an error.

CREATE FUNCTION and Multivalue Compression of UDT Columns

You can use multivalue compression only for distinct UDTs based on the following predefined data types:

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

You can specify the compression attributes COMPRESS NULL and NO COMPRESS for the following UDT data types:

- Distinct and structured types that are not based on BLOB or CLOB types
- ARRAY and VARRAY types
- Period types

CREATE FUNCTION and Algorithmic Compression of UDT Columns

You must specify either FOR COMPRESS or FOR DECOMPRESS, respectively, when you create an SQL function definition for an external UDF that is to be used to either compress or decompress BYTE, VARBYTE, CHARACTER, VARCHAR, GRAPHIC, Period, distinct UDT (including ARRAY/VARRAY), BLOB, CLOB, XML, Geospatial, distinct BLOB-based UDT, distinct CLOB-based UDT, or distinct XML-based UDT column values algorithmically. If you do not specify these options in the function definition, it cannot be used to algorithmically compress or decompress column values.

You can use algorithmic compression on distinct and structured UDT types.

The compression and decompression routines you code can be either external scalar UDFs or embedded services scalar UDFs.

Embedded services UDFs that are used for algorithmic compression are for Internal UDT types such as Period, ARRAY, VARRAY, and Geospatial. They *cannot* be used for structured UDTs.

External UDFs are normally used for distinct UDTs.

Accessing or writing an algorithmically-compressed column implicitly encapsulates the column reference with the appropriate routine.

See [Compressing Column Values Using Only Multivalue Compression](#) for additional information about the TD_LZ_COMPRESS and TD_LZ_DECOMPRESS UDFs.

Refer to the topics below for rules that apply to creating UDFs to implement algorithmic compression and decompression of UDT data.

Compression UDF

The compression UDF must have the following signature.

- There must be a single input parameter that can be any of the supported UDT data types.
- The parameter data type of the compression function must match the data type of the UDT column exactly.
- The UDT parameter data type of the compression function must match the return data type of the decompression function exactly.
- The return data type must be VARBYTE(*n*).
- The compression function VARBYTE(*n*) return length *n* must match the length of the decompression function VARBYTE(*n*) parameter exactly.

Decompression UDF

A decompression UDF must have the following signature:

- Single input parameter whose data type must be VARBYTE(*n*).
- The return data type of the decompression function must match the data type of the UDT column exactly.
- The return data type of the decompression function must match the UDT parameter data type of the compression function exactly.
- The decompression function VARBYTE(*n*) return length *n* must match the length of the compression function VARBYTE(*n*) parameter exactly.
- The output of the decompress UDF must be one of the supported UDT data types.

REPLACE FUNCTION and Algorithmic Compression

You can only submit a REPLACE FUNCTION request if the compression-related function is not used by any column across all databases and users for compressing and decompressing its data. This restriction is necessary because column data is already compressed using a certain algorithm and cannot be decompressed if the algorithm is changed.

The only exception to this rule is that a REPLACE FUNCTION request involving algorithmic compression is valid if it does not change the existing function specification in the data dictionary and its execution mode is PROTECTED.

For example, consider the following DDL requests where the objects are created in the order indicated.

```
CREATE FUNCTION scsu_comp ...;
CREATE FUNCTION scsu_decomp ...;
CREATE TABLE t1 (
  col_1 INTEGER,
  col_2 CHARACTER(10) COMPRESS ('abc', 'efg')
                        COMPRESS USING scsu_comp
                        DECOMPRESS USING scsu_decomp);
```

Column `col_2` in table `t1` references the UDFs `scsu_comp` for algorithmic compression and `scsu_decomp` for algorithmic decompression.

After table `t1` has been created, a REPLACE FUNCTION request on either `scsu_comp` `scsu_decomp` returns an error to the requestor. This assumes that the exception conditions noted earlier are not satisfied.

User-Defined Functions and Large Objects

The usage characteristics for a user-defined function with large object parameters or a large object return value are the same as the usage characteristics for any other UDF. In general, you can specify a UDF that accepts a LOB value as an argument in any context in which a UDF is otherwise allowed. You can also use a UDF that returns a LOB value in any context in which a value of that type is appropriate and a UDF is otherwise allowed.

As with other functions and operators, automatic type conversions can be applied to the arguments or the return values of a UDF. You should be careful about the possible performance implications of automatic type conversions with large objects used as UDF parameters. For example, a function whose formal parameter is a BLOB type could be passed a VARBYTE column as the actual parameter. The system converts the VARBYTE value into a temporary BLOB and then passes that to the UDF. Because even a temporary BLOB is stored on disk, the performance cost of the conversion is significant.

To avoid this, you should consider creating an overloaded function that explicitly accepts a VARBYTE TD_ANYTYPE argument. Note that you cannot create overloaded functions to enforce row-level security constraints.

Another possible cause of undesired conversions is truncation. Declared length is part of the data type. Consider a UDF whose formal parameter is declared to be BLOB(100000) AS LOCATOR, and suppose the actual parameter is a column whose type is declared as BLOB without a locator specification. In this case, the maximum length of the source type defaults to 2,097,088,000 bytes. Whenever the source type is longer than the target type, truncation might occur. Because of this, an automatic conversion operation, which results in a data copy as well as the creation of a temporary BLOB, must be generated. Therefore, whenever possible, you should define UDFs to accept the maximum length object by omitting the length specification.

This rule contradicts the policy for all other data types, which is that data type length declarations should only be as long as is necessary. See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Function Identifiers

Function names are distinct from specific function names, external function names, and function entry point names. See [Function Name](#), [Function Class Clause](#), [EXTERNAL NAME Clause](#), and [UDF Default Location Paths](#).

The following table briefly outlines the differences among the different function identifiers:

7: CREATE FUNCTION - CREATE GLOBAL TEMPORARY TRACE TABLE

| Function Identifier | Syntax Variable Name | Definition |
|---------------------------|----------------------------------|---|
| Function name | <i>function_name</i> | <p>The identifier used to call the function from an SQL statement.</p> <p>If no specific name is assigned to the function, then the name stored in the data dictionary is <i>function_name</i>.</p> <p>If a specific name is assigned to the function, then <i>function_name</i> is not the name by which the dictionary knows the function as a database object, and it need not be unique within its database.</p> |
| Specific function name | <i>specific_function_name</i> | <p>The identifier used to define the function as a database object in the dictionary table <i>DBC.TVM</i>.</p> |
| External function name | <i>external_function_name</i> | <p>The external function name for a C or C++ external routine can be expressed in two ways:</p> <ul style="list-style-type: none"> As an identifier that names the entry point for the function object. Case is significant, and the name must match the name of the C or C++ function body. As a string that identifies a client-defined path to the C or C++ function source code. <p>An external function name is mandatory for a Java external routine when you specify JAVA in the Language clause.</p> <p>In this case, <i>external_function_name</i> is referred to as an external Java reference string.</p> <p>The external Java reference string specifies the JAR file, Java class within the JAR, and the Java method within the class that is to be invoked when the Java UDF is executed.</p> <ul style="list-style-type: none"> The first part is the <i>JAR_ID</i>. This is the registered name of the associated JAR file. You must create this name using the external procedure SQLJ.INSTALL_JAR. The second part is the name of the Java class contained within the JAR that contains the Java method to execute. The third part is the name of the method that is executed. <p>For example, if you have a JAR file that has been registered in the database with the ID <i>salesreports</i>, and the class within the JAR is named <i>formal</i>, and the method within the class to be called is named <i>monthend</i>, then the string <i>salesreports:formal.monthend</i> completely defines the exact method to invoke when the Java UDF is executed.</p> |
| Function entry point name | <i>function_entry_point_name</i> | <p>The identifier used to define the entry point name of the function.</p> <p>A function entry point name must be specified if its name is different than the function name or specific function names defined for the function.</p> <p>You cannot specify <i>function_entry_point_name</i> more than one time in the EXTERNAL clause of the CREATE FUNCTION or REPLACE FUNCTION statements.</p> |

Function Name

The function name is the identifier used to call the function from an SQL request. It is not necessarily the database object name that is stored in *DBC.TVM*.

| Specific Name | Name Stored in <i>DBC.TVM</i> |
|------------------------------|---|
| Not assigned to the function | Function name. In this case, the function name must be unique within its database. |
| Assigned to the function | Specific function name. In this case, the function name need not be unique within its database. See Function Class Clause . |

Though you can give a function the same name as a column, you must be very careful to avoid specifying them ambiguously. For example, if the column name is followed with a database-style type clause as in the following example, then the system assumes that *text_find* is a reference to the function named *text_find*, *not* a reference to the identically named column *text_find*:

```
text_find(FLOAT),
```

There are two ways to make your request unambiguous:

- Use the ANSI syntax for CAST to make an explicit declaration of data type rather than a function parameter. For example,

```
CAST (text_find as FLOAT)
```

- Qualify the column name fully. For example,

```
sales.text_find (FLOAT)
```

In this example, *sales* is the table that contains the column named *text_find*.

You can precede the function name with its containing database or user name if the function is created in a different database or user than your default. The scope of the name is the database or user in which it is contained.

A function name need not be unique: several functions can have the same function name. This is referred to as function name overloading. If you overload function names, the parameter type specifications among the various overloaded function names must be sufficiently different to be distinguishable or you must use the TD_ANYTYPE parameter data type. You cannot overload the names of functions written to enforce row-level security.

See [Function Overloading](#) for a list of the rules the system uses to determine the uniqueness of a function by its parameters. See the table in [External Body Reference Clause](#) for more information about function name usage.

Function Calling Argument

The function calling argument is any simple SQL expression, including, but not limited to, constant values, column references, host variables, the NEW VARIANT_TYPE UDT constructor expression (see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210), or an expression containing any of these, including expressions containing UDFs.

You cannot call a Teradata Row-Level Security policy function from a DML request. Teradata Row-Level Security policy functions can only be specified with column definitions in a CREATE TABLE or ALTER TABLE request.

When you call a function, and that function is not stored in either your default database or in database *SYSLIB*, you must fully qualify the function call with a database name. If your default database and *SYSLIB* both contain functions matching the name of the called function, then the system references the UDF in your default database. This is the only exception that requires an explicit qualification for *SYSLIB*.

The argument types passed in the call must be compatible with the parameter declarations in the function definition of an existing function. If there are several functions that have the same name, and no qualifying database is specified, then the particular function that is picked is determined by the following process:

1. The system searches the list of built-in functions.
 - If the called function has the same name as a Vantage built-in function, the search stops and that function is used.
 - If a candidate function is not found, proceed to stage 2.
2. The system searches the list of function names in the default user database.
 - Candidate functions are those having the same name and number of parameters as specified by the function call as well as the best fit based on their parameter type.
 - If a candidate function is not found, proceed to stage 3.
3. The system searches the list of function names in the *SYSLIB* database.
 - Candidate functions are those having the same name and number of parameters as specified by the function call as well as the best fit based on their parameter type.
 - If no candidate function is found, an error is returned.

The rules for selecting a best fit candidate user-defined function once its containing database has been determined are described in [Function Overloading](#).

Function Overloading

Most function names need not be unique within a function class. Vantage uses the parameter types of identically named functions to distinguish among them, so parameter types associated with overloaded function names must be distinct.

Vantage uses the precedence order for compatible type parameters to determine which function is to be invoked when several functions having the same name must be differentiated by their parameter

types. For more information about function overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Vantage does not support name overloading for functions that implement row-level security policies because of the parameter types required by the system for input to the function. The same number of parameters and their corresponding data types 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 include or omit null indicators, depending on whether the constraint allows nulls.

You can simplify function name overloading by using dynamic UDTs to eliminate the need to create multiple UDFs as input parameter data types to cover an overloaded name. A dynamic UDT is a structured UDT with the preassigned name *VARIANT_TYPE*, which dynamically constructs its attributes at runtime based on the attributes the request passes to it.

For example, suppose you create the distinct UDT *integer_udt* and then create a UDF using *VARIANT_TYPE* as the data type for its only parameter:

```
CREATE TYPE integer_udt AS INTEGER FINAL;

CREATE FUNCTION udf_agch002002dyn_udt
  (parameter_1 VARIANT_TYPE)
RETURNS INTEGERUDT
CLASS AGGREGATE (4)
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!udf_agch002002dyn_udt!udf_agch002002dyn_udt.c'
PARAMETER STYLE SQL;
```

You can then construct a dynamic structured UDT at runtime using the *NEW VARIANT_TYPE* constructor expression to pass in the attributes of the type as follows:

```
SELECT udf_agch002002dyn_udt(NEW VARIANT_TYPE(tbl_1.a AS a,
                                             (tbl_1.b+tbl_1.c)
                                             AS b))
FROM tbl_1;
```

This request constructs a dynamic UDT with two attributes named *a* and *b*. See *NEW VARIANT_TYPE* in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

Because *udf_agch002002dyn_udt* is defined using *VARIANT_TYPE* for its only parameter, you could just as easily submit the following *SELECT* request, which the system would resolve by creating a dynamic UDT with *three* attributes named *a*, *b*, and *c*:

```
SELECT udf_agch002002dyn_udt(NEW VARIANT_TYPE(tbl_1.a AS a,
                                           tbl_1.b AS b,
                                           tbl_1.c AS c)
FROM tbl_1;
```

As you can see, by defining the UDF using `VARIANT_TYPE` as the data type for its input parameter, you can save yourself the trouble of having to create multiple UDFs to resolve overloaded function requests.

You cannot import or export a `VARIANT_TYPE` UDT, nor can `VARIANT_TYPE` be cast to a different type.

External routines written in Java do not support UDFs written using the dynamic UDTs `VARIANT_TYPE` and `TD_ANYTYPE`. To write such functions, you must write their external routines in C or C++.

By defining a function using `TD_ANYTYPE` as a parameter or return data type, you can overload the function based on its server character set or numeric precision rather than its name. When you define a function using `TD_ANYTYPE`, Vantage determines the parameter and return data types at execution time based on the parameters that are provided.

In addition to allowing `TD_ANYTYPE` to act as an alias for the type for parameter data types with undetermined attributes, you can also use `TD_ANYTYPE` to resolve all possible parameter data types. This allows you to develop fewer function signatures for the same basic function as well as to provide flexibility in coding the logic for the required function behavior.

However, using `TD_ANYTYPE` as a parameter data type results in the loss of the implicit UDF conversions that automatically convert input values to those that match the function signature. As a result, you have a greater responsibility to make sure that any undefined parameters are properly resolved and processed.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about `TD_ANYTYPE` and see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for more information about how to code UDFs to take advantage of the `TD_ANYTYPE` data type.

Vantage follows a set of parameter rules to determine the uniqueness of a function name. These rules are provided in the following list.

- The following numeric parameter types listed in order of precedence for determining function uniqueness. For example, a `BYTEINT` fits into a `SMALLINT` and a `SMALLINT` fits into an `INTEGER`. Conversely, a `FLOAT` does not fit into an `INTEGER` without a possible loss of information.

The types are distinct and compatible. Types sharing a number are synonyms and are not distinct from one another.

- `BYTEINT`
- `SMALLINT`
- `INTEGER`
- `DECIMAL`, `NUMERIC`

The size specification for `DECIMAL` and `NUMERIC` types does not affect the distinctiveness of a function. For example, `DECIMAL(8,3)` and `DECIMAL(6,2)` are identical with respect to determining function uniqueness.

- `FLOAT1`, `DOUBLEPRECISION`, `REAL`

- The following character parameter types are listed in order of precedence for determining function uniqueness.

The types are distinct and compatible. Types sharing a character are synonyms and are not distinct from one another. The length specification of a character string does not affect the distinctiveness of a function. For example, CHARACTER(10) and CHARACTER(5) are identical with respect to determining function uniqueness. CHARACTER SET clauses also have no effect on the determination of function uniqueness.

The length specification of a character string does not affect the distinctiveness of a function. For example, CHARACTER(10) and CHARACTER(5) are identical with respect to determining function uniqueness. CHARACTER SET clauses also have no effect on the determination of function uniqueness.

- CHARACTER
- VARCHAR, CHARACTERVARYING, LONGVARCHAR
- CHARACTER LARGE OBJECT
- The following graphic parameter types are distinct and compatible. Types sharing a bullet are synonyms and are not distinct from one another.
 - GRAPHIC
 - VARGRAPHIC, LONG VARCHAR CHARACTER SET GRAPHIC
- The following byte parameter types are distinct and compatible:
 - BYTE
 - VARBYTE
 - BINARY LARGE OBJECT
- All date, time, timestamp, and interval parameter types are distinct.
- If the number of parameters in identically named existing functions is different or if the function parameter types are distinct from one another in at least one parameter, then the function being defined is considered to be unique.
- If more than one function has the same *function_name*, then you must supply a *specific_function_name*.
- You can only overload function names within the same class within a given database. For example, you cannot have a scalar function and an aggregate function with the same *function_name* within the same database.

Parameter Names and Data Types

The parameter list contains a list of variables to be passed to the function.

The naming of ordinary function parameters is optional; however, all input parameters specified for a row-level security policy UDF *must* be named.

If you specify one parameter name, then you must specify *all* parameter names explicitly. Parameter names are standard SQL identifiers. If you do not specify parameter names, then Vantage creates arbitrary

names for them in the format P *n*, where *n* is an integer that specifies the order in which the parameters are created, beginning with P1.

Parameter names are used by the COMMENT statement (see [COMMENT \(Comment Placing Form\)](#)) and are reported by the HELP FUNCTION statement (see [HELP FUNCTION](#)). Parameter names, with their associated database and function names, are also returned in the text of error messages when truncation or overflow errors occur with a function call.

Each parameter type is associated with a mandatory data type to define the type of the parameter passed to or returned by the function. The specified data type can be any valid data type, including UDTs (see *Teradata Vantage™ - Data Types and Literals*, B035-1143 for a complete list of data types). Character data types can also specify an associated CHARACTER SET clause.

For character string types like VARCHAR that might have a different length depending on the caller, the length of the parameter in the definition indicates the longest string that can be passed. If there is an attempt to pass a longer string, the result depends on the session mode.

You cannot specify a character data type that has a server character set of KANJI1. Otherwise, the system returns an error to the requestor.

Input parameter names for a row-level security policy UDF must be named because a security constraint policy UDF is never called by a user in an SQL DML request. Instead, Vantage automatically calls the security policy function whenever the corresponding statement type is executed against a table on which the constraint has been defined. Because Vantage automatically generates the input parameter values for the UDF, it must know the source you want, and the input parameter name defines that source for the parameter.

You must specify the following system-defined parameter names as the source of the parameter to Vantage. The meaning of the system-defined parameter name is that the input provided by Vantage to a security policy UDF is the constraint values from the source defined by the specified DELETE, INSERT, SELECT, or UPDATE statement-actions.

| This parameter name ... | Defines the source of the parameter as ... |
|-------------------------|--|
| CURRENT_SESSION | the value that is currently set for the session for the constraint to which the UDF applies. |
| INPUT_ROW | being in the corresponding constraint column of the row that is the object of the request. |

You can declare input parameters in procedures written in C or C++ *only* to have the following parameter data types.

- VARIANT_TYPE
- TD_VALIST

VARIANT_TYPE data type functions as a structured UDT with a fixed type name of VARIANT_TYPE. VARIANT_TYPE is used to represent the dynamic UDT data type and can be used only to type non-Java UDF input parameters. See [Writing a UDF That Uses VARIANT_TYPE UDT Parameters](#).

The system passes a dynamic UDT data type to this parameter position during runtime using the `NEW VARIANT_TYPE` expression. For examples, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

You *cannot* specify the `VARIANT_TYPE` data type for input parameters in a Java procedure or Java UDF, nor can you specify it for returned result sets. The maximum number of input parameters you can declare with the `VARIANT_TYPE` UDT data type is 8.

If you attempt to create a UDF that declares more than 8 input parameters with a `VARIANT_TYPE` type, or if you attempt to create a UDF that declares an output parameter with a `VARIANT_TYPE` type, the system returns an error to the requestor.

You can declare both the input and output parameters in procedures written in C or C++ *only* to have the `TD_ANYTYPE` data type. `TD_ANYTYPE` functions as a structured UDT with a fixed type name of `TD_ANYTYPE`. `TD_ANYTYPE` represents the dynamic UDT data type and can be used only to type non-Java UDF parameters. Vantage passes a dynamic UDT data type to this parameter position during runtime.

You *cannot* specify the `TD_ANYTYPE` data type for parameters in a Java UDF.

For more information about the `TD_ANYTYPE` data type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143 and for more information about how to code your UDFs to make the best use of `TD_ANYTYPE`, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The `TD_VALIST` input parameter data type enables XML domain-specific UDFs *only* to use varying number of variables in XML function definitions. In C and C++, it is useful to be able to declare a function footprint that is not fixed with respect to the number and types of parameters that can be passed into the function at runtime. This is the function of the `TD_VALIST` input parameter data type: to signify that n parameters with any data type can be passed into the function at runtime, where the maximum value for n is 128.

Vantage only supports `TD_VALIST` for XML Embedded Services functions. If you create a UDF that specifies an input parameter of type `TD_VALIST`, it can only be a single input parameter to the UDF. Overloading the function is not allowed when the function input parameter has a `TD_VALIST` data type.

`TD_VALIST` is an input parameter data type that specifies that n parameters with various data types can be passed into the function at runtime. `TD_VALIST` provides the benefit of reducing the number of overloading routines required to support routines with varying numbers of input parameters.

The following table summarizes the standard Vantage session mode semantics with respect to character string truncation.

| IF the session mode is ... | THEN ... |
|----------------------------|--|
| ANSI | <p>any pad characters in the string are truncated silently and no truncation notification is returned to the requestor.</p> <p>A truncation exception is returned whenever non-pad characters are truncated.</p> <p>If there is a truncation exception, then the system does not call the function. The relevant indicator values are not set to the number of characters truncated.</p> |

| IF the session mode is ... | THEN ... |
|----------------------------|---|
| Teradata | the string is truncated silently and no truncation notification message is returned to the requestor. |

For details on parameter passing conventions for the TD_GENERAL, SQL, and JAVA parameter styles, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Row-Level Security UDF Enforcement of the Security Policy for a Row

The database executes the row-level security policy function you assign to the constraint object that corresponds to a table column to enforce the security policy for a data row. You can create a scalar function for each of the four request types (DELETE, INSERT, SELECT, UPDATE) that can be executed against the table on which the security constraint is defined.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to write row-level security policy UDFs.

See *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 for general information about row-level security.

General Rules for Creating a Row-Level Security UDF

You use CREATE FUNCTION requests to generate the functions that Vantage executes to enforce the security policy for a data row. You can create a function for each of the four statement-action types that can be executed on a table defined with the full set of security constraints. See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to code row-level security UDFs.

The following rules apply to creating UDFs to enforce row-level security:

- You can only code the external routines for row-level security functions using C and C++.
Java is not supported for external routines written to enforce row-level security.
- Only scalar type UDFs can be used as row-level security functions.
- Name overloading is not supported for row-level security functions.

Because of the parameter types required by the server for input to the UDF function, the same number of parameters and their data types are required for each UDF that executes the security policy for a specific statement-action.

The only difference there can be between the parameters for UDFs that execute a security policy is that the parameters of different UDFs can include or omit null indicator variables, depending on whether the constraint allows nulls.

No other values are acceptable for an indicator variable parameter.

A parameter style of TD_GENERAL for a row-level security UDF specifies that there are no indicator variable parameters for the UDF parameters.

- Unlike the case for other UDFs, input parameter names for a row-level security UDF are mandatory because they define the source for those parameters.

A security constraint policy UDF cannot be called by a user from an SQL request. Instead, Vantage automatically calls row-level security functions whenever the corresponding statement-action type is executed for a table on which the constraint UDF has been defined. Because Vantage automatically generates the input parameter values for the UDF, it must know the source the user desires.

Two system-defined parameter names exist for row-level security functions.

- The valid system-defined input parameters for each UDF depend on the statement-action for the UDF as defined in the constraint object.

| This parameter name ... | Defines the source of the parameter as ... |
|-------------------------|--|
| CURRENT_SESSION | the value that is currently set for the session for the constraint to which the UDF applies. |
| INPUT_ROW | being in the corresponding constraint column of the row that is the object of the request. |

In these cases, the system-defined parameter names indicate that the input provided by Vantage to a security policy UDF are the constraint values from the source defined in the following table:

| Statement | Required Input Parameters | Statement-Action Result |
|-----------|--|---|
| DELETE | INPUT_ROW | An indication of whether the session passed the security policy test. |
| INSERT | CURRENT_SESSION | A value that is placed in the target row. |
| SELECT | <ul style="list-style-type: none"> CURRENT_SESSION INPUT_ROW | <p>An indication of whether the session passed the security policy test.</p> <p>If the result is that the policy test did not pass, Vantage discontinues processing of the action, moves to the next row, and does not generate an audit row.</p> |
| UPDATE | <ul style="list-style-type: none"> CURRENT_SESSION INPUT_ROW | A value that is placed in the target row. |

- If you do not create a constraint function for a statement-action type, that statement-action can only be executed by a user who has the OVERRIDE privilege required to execute the request.

Either the request must also include the values to be assigned to the constraint columns of the target rows or you must select the rows to be assigned to the target table from the source table.

See [UDF Parameter Styles](#) for more information about function parameter styles.

General Rules for Row-Level Security Function Parameter Styles

The following set of rules applies to all row-level security function parameter styles.

- If the parameter style for a function is SQL, a constraint column can accept a null.
A parameter style of SQL for a row-level security UDF specifies that each input and output parameter for the UDF has a second indicator variable parameter to indicate if the return value is or is not null.
- If the parameter style for a function is TD_GENERAL, a constraint column cannot accept a null.
- If the value for a return indicator variable is -1, the value of the output parameter for the function is null.
- The result of the function call must be a value that Vantage places in the target row.
- A parameter style of SQL for a row-level security UDF specifies that each input and output parameter for the UDF has a second indicator variable parameter.

The SQL parameter style for a row-level security UDF also specifies that each input and output parameter for the UDF has a second indicator variable parameter to indicate if the return value is or is not null.

In the C or C++ definition of the UDF, the set of indicator variable parameters follows the set of corresponding value parameters. Indicator variable parameters must be specified in the same order as the value parameters.

| An indicator variable value of ... | Indicates that the corresponding value parameter is ... |
|------------------------------------|---|
| -1 | null, meaning that it is not defined. |
| 0 | not null. |

- The RETURNS data type that you specify for a row-level security UDF depends on the statement-action the function performs.

| IF the function implements this statement-action ... | THEN the RETURNS clause data type must be ... |
|--|--|
| <ul style="list-style-type: none"> ◦ INSERT ◦ UPDATE | the same as the data type used to specify the applicable constraint. The value for the parameter is the value to be inserted into, or updated in, the row. |
| <ul style="list-style-type: none"> ◦ DELETE ◦ SELECT | a CHARACTER(1) type to return a value of T or F. |

See [Rules for INSERT and UPDATE Row-Level Security Policy Functions](#) and [Rules for SELECT and DELETE Row-Level Security Policy Functions](#) for parameter style information that is specific to those row-level security function types.

Rules for INSERT and UPDATE Row-Level Security Policy Functions

The following set of rules is specific to INSERT and UPDATE row-level security policy functions.

- For an INSERT or UPDATE row-level security constraint UDF, the result must be a value to be placed in the target row.
- The RETURN parameters for a hierarchical (SMALLINT) INSERT or UPDATE constraint with a parameter style of SQL are as follows.

| This return output ... | With this return indicator variable value ... | Indicates that the call ... |
|------------------------|---|---|
| NULL | -1 | passed the policy and the constraint for the new row is null. |
| any non-0 | -1 | passed the policy and the constraint for the new row is null. |
| any non-0 | 0 | passed the policy. The return value is a valid constraint value, is not null, and is the value for the new row. Vantage verifies the returned constraint value meets the constraint definition. |
| 0 | 0 | did not pass the policy and the corresponding parameter is not null. Vantage ignores the target row for any further action. |

- The RETURN parameters for a non-hierarchical (BYTE[(n)]) INSERT or UPDATE constraint UDF with a parameter style of SQL are as follows.

| This return output ... | With this return indicator variable value ... | Indicates that the call ... |
|------------------------|---|---|
| NULL | -1 | passed the policy and the constraint for the new row is null. |
| any non-0 | -1 | passed the policy and the constraint for the new row is null. |
| any non-0 | 0 | passed the policy. The return value is a valid constraint value, is not null, and is the value for the new row. Vantage verifies the returned constraint value meets the constraint definition. |
| 0 | 0 | did not pass the policy and the corresponding parameter is not null. Vantage ignores the target row for any further action. |

- The RETURN parameters for a hierarchical (SMALLINT) INSERT or UPDATE constraint UDF with a parameter style of TD_GENERAL are as follows.

| This return output ... | Indicates that the call ... |
|------------------------|--|
| any non-0 value | passed the policy, the value is a valid constraint value, and is the value for the new row. Vantage verifies the return constraint value meets the constraint definition. |
| 0 | did not pass the policy. Vantage ignores the target row for any further action. |

- The RETURN parameters for a non-hierarchical (BYTE[(n)]) INSERT or UPDATE constraint UDF with a parameter style of TD_GENERAL are as follows.

| This return output ... | With this return indicator variable value ... | Indicates that the call ... |
|------------------------|---|---|
| any byte is non-zero | 0 | passed the policy, the value is a valid constraint value, and is the value for the new row. |
| all bytes are 0 | 0 | did not pass the policy. Vantage ignores the target row for any further action. |

Rules for SELECT and DELETE Row-Level Security Policy Functions

The following set of rules is specific to SELECT and DELETE row-level security policy functions.

- For a SELECT or DELETE UDF, the return value indicates whether the session has passed the security policy test. The following table lists the meanings of the return indicator variables for both SMALLINT and BYTE[(n)] types.

| This return output ... | With this return indicator variable value ... | Indicates that the call ... |
|------------------------|---|--|
| T | 0 | passed the policy. |
| F | 0 | did not pass the policy. Vantage ignores the target row for any further action. |

- If the constraint data type is either SMALLINT or BYTE(n) and the constraint allows nulls, its parameter style must be SQL. A parameter style of SQL for a row-level security UDF specifies that each input and output parameter for the UDF has a second indicator variable parameter to indicate if the return value

is or is not null. In this case the indicator variable parameters are required because they are needed for the input parameters.

The RETURN parameters are as follows.

| This return output ... | With this return indicator variable value ... | Indicates that the call ... |
|----------------------------|---|--|
| any return value | -1 | is not valid. |
| T | 0 | passed the policy. |
| F | 0 | did not pass the policy. Vantage ignores the target row for any further action. |
| anything other than T or F | 0 | is not valid. |

- If the constraint data type is SMALLINT and the constraint does not allow nulls, its parameter style must be TD_GENERAL.

The RETURN parameters for a hierarchical SELECT or DELETE constraint with a parameter style of SQL are as follows.

| This return output ... | Indicates that the call ... |
|------------------------|--|
| T | passed the policy and continues processing the action. |
| F | did not pass the policy and does not continue processing the action. Vantage moves to the next row without reporting an error to the requestor. If you are logging security access to the table, Vantage does not generate an audit row for the failure. |

Writing a UDF That Uses VARIANT_TYPE UDT Parameters

A dynamic UDT is a structured UDT with a type name of VARIANT_TYPE or TD_ANYTYPE. You can write scalar and aggregate UDFs for external routines written in C or C++, but not for external routines written in Java. All FNC library functions that are valid for structured UDTs are also valid for dynamic UDTs. For those FNC library functions that require an attribute name, you must specify the supplied alias name or supplied column name as the corresponding attribute name. See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details. The maximum number of VARIANT_TYPE input parameters you can declare in a UDF is 8.

Writing a UDF with a dynamic result row specification is the same as writing the same routine with a structured UDT because dynamic UDT parameters are structured type parameters and are passed into the function as a UDT_HANDLE the same way as structured types.

An illustrative example CREATE FUNCTION request for the function definition might look like this.

```
CREATE FUNCTION udfwithstructured (...)
RETURNS INTEGER
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL NAME 'CS!UdfWithStructured!td_udf/udfwithstructured.c';
```

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details on how to write C and C++ routines to support UDFs that use dynamic UDTs. See [Function Overloading](#) for an example of how dynamic UDTs can be used to reduce or eliminate the need to create multiple UDFs to handle function name overloading.

Writing a UDF That Uses TD_ANYTYPE UDT Parameters

You can write scalar and aggregate UDFs that use TD_ANYTYPE as a parameter or return data type for external routines written in C or C++, but not for external routines written in Java. The FNC library routines that are valid for structured UDTs are also valid for dynamic UDTs, but the specific FNC library function for the TD_ANYTYPE dynamic UDT is FNC_GetAnyTypeParamInfo. See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for more information about the FNC_GetAnyTypeParamInfo library function.

An illustrative example CREATE FUNCTION request for the function definition might look like this.

```
CREATE FUNCTION udf_3 (
  a TD_ANYTYPE,
  b TD_ANYTYPE)
RETURNS TD_ANYTYPE;
```

This function takes two TD_ANYTYPE input parameters, *a* and *b*, and returns a TD_ANYTYPE result parameter. The parameter data types are determined at run time when the function is called.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about the TD_ANYTYPE data type and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for more information about how to best code external routines that support the TD_ANYTYPE type.

RETURNS Clause

This mandatory clause specifies the data type of the parameter value returned by the function. You can specify any valid data type except TD_ANYTYPE, VARIANT_TYPE and TABLE, including UDTs.

The defined return data type is stored as an entry in DBC.TVFields using the name RETURN0[*n*], where *n* is a sequentially defined integer used to guarantee the uniqueness of the value. This ensures that user-defined parameter names are not duplicated.

Restrictions on Declaring an External C++ Function

If you specify CPP in the Language Clause, then you must declare the main C++ function as extern “C” to ensure that the function name is not converted to an overloaded C++ name, for example:

```
extern "C"
void my_cpp(long int *input_int, long int *result, char sqlstate[6])
{
```

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details.

SQL Data Access Clause

This mandatory clause specifies whether SQL statements are permitted within the user-written external routine for the function.

The only valid specification is NO SQL.

You can specify the SQL Data Access and Language clauses in any order.

Optional Function Characteristics

The following set of clauses is optional. You can specify all or none of them, and in any order, but each clause defaults to a particular value if it is not specified.

- Specific function name (see [SPECIFIC Function Name Clause](#))
- Function class (see [Function Class Clause](#))
- Parameter style (see [Parameter Style Clause](#))
- Deterministic characteristics (see [Deterministic Characteristics Clause](#))
- Null call (see [Null Call Clause](#))
- External data access (see [External Data Access Clause](#))

If you specify any of these clauses, you must specify them sequentially prior to specifying the External Clause (see [EXTERNAL NAME Clause](#)), which can be followed by a Parameter Style Clause see [Parameter Style Clause](#)).

SPECIFIC Function Name Clause

This clause is mandatory if you are using function name overloading, but otherwise is optional.

The clause specifies the specific name of the function. The specific function name is the name placed in the *DBC.TVM* table and has the same name space as tables, views, macros, triggers, join indexes, hash indexes, and stored procedures. A specific function name must be unique within its containing database.

Function Class Clause

This optional clause specifies the class of function being defined. Specify it only for aggregate functions.

The aggregate function is invoked for each group in a SELECT statement that requires aggregation and computes a single value for each group it is called for. You can specify an aggregate UDF anywhere you can specify a built-in aggregate function.

A function of one class cannot have the same name as a function of another class within the same database, even if their parameters differ. Overloading function names is only supported within the same class of a function.

This clause is a Teradata extension to the ANSI SQL:2011 standard.

Optimizing the Aggregate Cache

Appropriate management of the aggregate cache used by aggregate UDFs is very important for optimizing their performance. Excessive cache paging can have an enormous negative effect on the performance of an aggregate UDF.

You can specify the size of the interim aggregate cache for a UDF, which is the maximum number of cache bytes used by that function, to a value between a minimum of 1 byte and a maximum of 64,000 bytes.

Allocating the optimal size for the aggregate cache is important because by doing so, you allocate the group cache used by the function. For example, if you specify an aggregate cache size of 64 Kbytes, then the maximum number of interim group cache entries is 15, which is determined as follows:

$$\text{Number of cache entries} = \frac{1 \text{ Mbyte}}{64 \text{ Kbytes}} = 15.625 \cong 15$$

If an aggregation requires more than 15 group cache entries, then those entries must be flushed to disk more frequently than would be necessary with a smaller aggregate cache allocation.

Aggregate UDFs use the same aggregate processing functionality as system-defined aggregate functions like SUM and MAX (see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145). Aggregate processing creates interim group entries to maintain the aggregation information between rows associated with the same aggregation group. The system caches these interim group entries to enhance performance, but in those cases where the number of interim groups exceeds the capacity of the aggregate cache to contain them, the least recently used entries are paged to a spool on disk. The system does not attempt to retrieve or combine interim group entries from spool until the last step of aggregate processing.

To achieve optimum performance, your goal is to keep all of the interim group entries in cache by minimizing the number of aggregation phases per group. The best way to improve aggregate cache usage for an aggregate UDF is to adjust the size of the aggregate interim group (the aggregate storage size) to exactly what is needed: no more and no less. Aggregation works correctly no matter what aggregate storage size you specify, but you can optimize its performance by adjusting the interim storage size using the CLASS AGGREGATE *interim_size* option when you create or replace an aggregate UDF.

The task is to optimize the trade off between maximizing the number of cache entries that can share the aggregate cache while at the same time allocating enough memory to handle the aggregation tasks required.

As long as the aggregate cache does not overflow, there are at most (*number_of_AMPs* - 1) aggregation phases for each group. But if the aggregate cache overflows, a large number of aggregation phases can occur, which has a negative impact on performance.

There are three variables to be considered in adjusting the aggregate storage size:

| Variable | Description |
|---------------------------------------|---|
| Aggregate cache size | This value is fixed in the database at 1 MB. |
| Aggregate storage size | This value is specified by the CLASS AGGREGATE <i>interim_size</i> option. The aggregate storage size can range from a minimum of 1 byte to a maximum of 64 KB, with a default value of 64 bytes. |
| Number of interim group cache entries | This value depends on the aggregate storage size, and is calculated as follows: $\text{Number of interim group cache entries} = \frac{\text{Aggregate cache size}}{\text{Aggregate storage size}}$ |

Note that the number of interim aggregation groups for an aggregate UDF is a direct function of the size of the aggregate storage specified by the UDF definition. The following table shows two extreme examples of this ranging over three orders of magnitude.

| Aggregate Storage Size | Number of Interim Groups That Can Be Contained in Cache |
|------------------------|---|
| 64 bytes | 15,625 |
| 64 KB | 15 |

As noted previously, when the number of interim groups exceeds the number that can be contained in the aggregate cache defined for the function, the cache overflows and some number of the least recently used entries are flushed to a spool. This paging of aggregation groups permits an unlimited number of groups to be processed, but at the cost of an additional aggregation phase for each aggregate cache entry written to spool *plus* the performance impact of writing to, and reading from, the spool on disk.

Parameter Style Clause

This optional clause specifies the parameter passing style to be used by the function. Vantage supports the following parameter passing styles for user-defined functions:

- SQL
- TD_GENERAL
- JAVA

The following rules apply to the parameter styles specified for row-level security policy functions.

- The SQL parameter style defines that each input and the output parameter for the UDF has a second indicator parameter.

The SQL parameter style enables the C or C++ code body to indicate null data.

The set of indicator parameters in the C or C++ definition of the UDF follows the set of corresponding value parameters. The indicator parameters must be in the same order as the value parameters.

| IF the indicator parameter value is ... | THEN its corresponding value parameter is ... |
|---|---|
| -1 | null. |
| 0 | is not null. |

No other values are acceptable for an indicator parameter.

- The TD_GENERAL parameter style defines that there are no indicator parameters for the UDF parameters.

The TD_GENERAL parameter style does not enable the code body to indicate null data.

- The JAVA parameter style is not supported for row-level security policy functions.

The SQL parameter style allows the code body to indicate null data. This cannot be done with the TD_GENERAL parameter style.

You can specify the parameter style clause in one of two places:

- SQL data access clause
- External function name

You cannot specify the parameter style both places in the same user-defined function definition statement. Only one parameter style clause is permitted per UDF definition.

See [General Rules for Row-Level Security Function Parameter Styles](#) for parameter style information that is specific to row-level security constraint UDFs.

Specific details of both options are described in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Deterministic Characteristics Clause

This optional clause declares whether the function returns the same results for identical inputs or not.

Vantage supports the following deterministic characteristics options:

- DETERMINISTIC
- NOT DETERMINISTIC

Vantage must evaluate each row selected by a non-deterministic UDF condition individually because the evaluation of the function can change for each row selected within a query (this qualification includes both specifying NOT DETERMINISTIC explicitly and *not* specifying DETERMINISTIC, because NOT DETERMINISTIC is the default option for the clause). A predicate that includes a non-deterministic UDF

specified on an index results in an all-AMP operation in spite of the index having been specified in the predicate.

Null Call Clause

This optional clause specifies how the function handles null results. The clause has two options:

- CALLED ON NULL INPUT
- RETURNS NULL ON NULL INPUT

The behavior of this clause is correlated with the specification you make for the parameter style:

| IF you specify this parameter style ... | THEN nulls ... |
|---|--|
| SQL | can be specified for both the input and the result. This is accomplished by providing an indicator value for each parameter as well as for the result parameter. |
| TD_GENERAL | raise an exception condition. |
| JAVA | <p>behave differently depending on whether the function uses an object map or a simple map (see Data Type Mapping Between SQL and Java for details), as follows.</p> <ul style="list-style-type: none"> • If an object map type is used, nulls can be specified for both the input and the result. • If a simple map type is used, nulls raise an exception condition. <p>A null can be explicitly returned from a scalar or aggregate Java UDF only when the return type of its Java method is defined as an object-mapped data type.</p> <p>If the return type of the Java method is defined as simple-mapped, then you cannot return a null at runtime because simple-mapped data types cannot represent nulls.</p> <p>You can pass a null as a parameter at runtime to a scalar or aggregate Java UDF only under the following conditions.</p> <ul style="list-style-type: none"> • If a parameter is object-mapped, then you can specify either RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT. If you specify RETURNS NULL ON NULL INPUT, Vantage detects a null for the parameter and returns null as the result of the UDF. The function does not evaluate the null parameter. If you specify CALLED ON NULL INPUT, Vantage passes the null for the parameter to the UDF to be evaluated appropriately. • If a parameter is simple-mapped, you can only specify RETURNS NULL ON NULL INPUT. In this case, the system detects the null for the parameter and returns it as the result of the UDF. The function does not evaluate the null parameter. While you can specify CALLED ON NULL INPUT for CREATE FUNCTION and REPLACE FUNCTION definitions, invocation of the UDF with a null always fails. |

For C and C++ procedures, if any of the passed arguments is null, then the action taken depends on the option you specify for handling nulls.

| IF you specify ... | THEN the function is ... |
|---|---|
| CALLED ON NULL INPUT | called and evaluated. |
| RETURNS NULL ON NULL INPUT and any of the parameter values passed to the function are null | not called. Instead, it always returns a null. This option is useful to avoid generating an exception condition for functions defined with the TD_GENERAL parameter style, which does not handle nulls. You <i>cannot</i> specify RETURNS NULL ON NULL INPUT for aggregate functions because they must always be called even if they are passed a null. |

External Data Access Clause

This optional clause defines the relationship between a UDF or external stored procedure and data that is external to Vantage.

The option you specify determines:

- Whether or not the external routine can read or modify external data
- Whether or not the database will redrive the request involving the function or procedure after a database restart

If Redrive protection is enabled, the system preserves responses for completed SQL requests and resubmits uncompleted requests when there is a database restart. However, if the External Data Access clause of an external routine is defined with the MODIFIES EXTERNAL DATA option, then the database will not redrive the request involving that function or procedure. For details about Redrive functionality, see:

- *Teradata Vantage™ - Database Administration*, B035-1093
- The RedriveProtection and RedriveDefaultParticipation DBS Control fields in *Teradata Vantage™ - Database Utilities*, B035-1102.

If you do not specify an External Data Access clause, the default is NO EXTERNAL DATA.

The following table explains the options for the External Data Access clause and how the database uses them for external routines.

| Option | Description |
|------------------------|--|
| MODIFIES EXTERNAL DATA | The routine modifies data that is external to the database. In this case, the word <i>modify</i> includes delete, insert, and update operations. Note: Following a database failure, the database does not redrive requests involving a function or external stored procedure defined with this option. |
| NO EXTERNAL DATA | The routine does not access data that is external to the database. This is the default. |
| READS EXTERNAL DATA | The routine reads, but does not modify data that is external to the database. |

External Body Reference Clause

The required external body reference clause declares that the function is external to Vantage and identifies the location of all the file components it needs to be able to run.

An external function name is optional. When it is specified, it must be the name of the C or C++ source code file on the client system to be retrieved and compiled as part of the CREATE FUNCTION or REPLACE FUNCTION process.

If the external function name alone is not sufficient to specify the location of all the function components, you must specify a string literal that explicitly specifies the path to each of those elements. See [External String Literal](#).

In all cases, *function_name* is the identifier you specify when you invoke the function from SQL statements.

You can optionally specify either or both of the following options for this clause:

- External function name
- Parameter style

You can specify an external function name in several different ways. See the following for details:

- [External String Literal](#)
- [Include Name Clause](#), [Library Name Clause](#), [Object File Name Clause](#), [Package Name Clause](#), and [Source File Name Clause](#)

The following table summarizes the options you can specify in this clause.

| IF CREATE FUNCTION specifies this clause ... | THEN ... |
|--|--|
| EXTERNAL | <ul style="list-style-type: none"> • If you specify the SPECIFIC clause, then the C or C++ function name must match the <i>specific_function_name</i>. • If you do not specify the SPECIFIC clause, then the C or C++ function name must match the <i>function_name</i>. <p>If the client is mainframe-attached, then the C or C++ function name must be the DDNAME for the source.</p> |
| EXTERNAL NAME <i>external_function_name</i> | <p>the C or C++ function name must match <i>function_name</i>.</p> <p>If the client is mainframe-attached, then <i>function_name</i> must be the DDNAME for the source.</p> |
| EXTERNAL NAME ' <i>string</i> ' | <p>'<i>string</i>' can specify the C or C++ function name by stipulating the F option with a <i>function_entry_name</i>, which must match the name of the C or C++ function.</p> <p>The maximum length of '<i>string</i>' is 1,000 characters.</p> <p>You cannot specify the F option in '<i>string</i>' without also specifying an include, library, object, package, or source file name. Vantage needs one or more of these file names to link to.</p> <p>If '<i>string</i>' does not stipulate a <i>function_entry_name</i>, then the following rules apply to the C or C++ function name.</p> |

| IF CREATE FUNCTION specifies this clause ... | THEN ... |
|--|--|
| | <ul style="list-style-type: none"> • If you specify the SPECIFIC clause, the C or C++ function name must match the <i>specific_function_name</i>. • If you do not specify the SPECIFIC clause, the C or C++ function name must match the <i>function_name</i>. |

For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Note that these specifications cannot be used for Java external procedures.

You can specify the parameter style for the function either in this clause or in the Optional Function Characteristics clause, but you can only specify the parameter style for a function one time in its definition. See [Parameter Style Clause](#) for more information.

EXTERNAL NAME Clause

You use the External Name clause to specify the locations of all components needed to create the external routine.

Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

All files included by, or linked to, an external routine must be accessible to the database. These files can reside on a client system or within Advanced SQL Engine. Best practice is to keep the source on the client system to ensure that the programmer writing the external routine can control who can access the code. To do this, specify the C option in the external string literal clause to notify the system that the source is stored on the client.

Source code files can be stored on the database server using the S option in the external string literal clause to notify the system that the source is stored on the server. There are specific default server directories for that purpose.

The system automatically copies any client-resident source, header, or object files specified in the External Name clause to the server. Note that the system does *not* transport external libraries and shared object files, which must be manually installed in Advanced SQL Engine.

You can specify the external function name in one of three ways:

- As an external routine name identifier with optional Parameter Style clause.
- As an external routine object name.
- As a coded string that specifies the explicit path to the specified code entity.

| IF the external routine name is . .. | THEN ... |
|---|--|
| an identifier | it is the name of the entry point for the external routine object. The identifier is case sensitive and must match the C or C++ external routine name. |
| a string | it is composed of one of the following: <ul style="list-style-type: none"> ◦ a C or C++ external routine entry point name specification ◦ an encoded path to the specified code entity For a Java external routine, you must specify an external Java reference string. The maximum length of the external name string is 1,000 characters. |

You can specify more than one encoded string per external routine definition, though some can only be specified once within a list of strings.

External String Literal

The external string literal specifies an encoded list of the components required to create a UDF. When decoded, the string specifies what each component is and where it is located.

Depending on the requirements of the specific UDF, you can specify the following components:

- Entry point name of the function to be run (see [Function Entry Name Clause](#))
- Include file (C or C++ header file) paths (see [Include Name Clause](#))
- System library paths (see [Library Name Clause](#))
- Object code paths (see [Object File Name Clause](#))
- Package (user-created library) paths (see [Package Name Clause](#))
- Source code paths (see [Source File Name Clause](#))

The system uses the code path to access the file it specifies, then adds an appropriate extension as it transfers that file to the UDF compile directory on the platform.

Do not use the following characters as the delimiter because they might be interpreted as a component of a path string rather than as a delimiter:

- / (SOLIDUS)
- \ (REVERSE SOLIDUS)
- : (COLON)

You must use the same delimiter character throughout the specified path string.

Vantage retrieves the source component from the specified client or server when it requests the source or object code. The system saves the function entry point name component in the dictionary and then uses that name to invoke the function.

You should name any include files (C/C++ header files) with the same name as specified in the C/C++ include directive of the C or C++ source file (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147).

For example for a z/OS client:

```
'CI;sqltypes_td;UDFDEFS;CS;myudf;UDFSRC;F;theudf'
```

where UDFDEFS and UDFSRC are the DDNAMEs for the files on the z/OS client system. Whenever you refer to an IBM client file in a code path, you must identify it by its client DDNAME.

This clause causes the system to do the following things:

- Retrieve the file UDFDEFS and rename it to `udfdefs.h` in the UDF compile directory for the platform.
- Retrieve UDFSRC and rename it to `myudf.c` in the UDF compile directory for the platform.

The C or C++ source `myudf.c` should contain the following code:

```
#include <udfdefs.h>
```

or

```
#include "udfdefs.h"
```

The clause also specifies the F option, which stipulates that the function entry name is `theudf`. The C or C++ file named `theudf` must then have the following skeletal code:

```
void the udf ( ... )
{
    ...
}
```

You cannot specify the F option unless you also specify a source, object, package, or library file in the specification string.

Suppose you specify the following skeletal CREATE FUNCTION statement, where the omitted details are denoted by an ELLIPSIS (...) character:

```
CREATE FUNCTION abc(...)
...
EXTERNAL NAME 'CS!matrix!matrix.c';
```

The name of the C or C++ source file in this example is `matrix.c` and the C or C++ function name, based on the function name `abc`, must have the following skeletal code:


```
void abc(...)
{
    ...
}
```

Suppose you specify the following skeletal CREATE FUNCTION statement, where, again, the omitted details are denoted by an ellipsis (...):

```
CREATE FUNCTION abc(...)
...
EXTERNAL NAME 'CO!matrix!matrix.o';
```

There is no source file for this function; instead, the code is an object file named `matrix.o`.

For this CREATE FUNCTION statement to compile successfully, there must be a function in the specified object file named `abc`.

See the table at the end of the [External String Literal Examples](#) topic for more information about function name usage.

You can specify the various options as many times as needed except for the package option, which cannot be used in combination with any of the other options except the C/C++ function entry point name option.

Function Entry Name Clause

The Function Entry Name clause specifies an identifier for the *function_entry_name* variable as it is defined in its C or C++ module. It must match the function entry point name when the source is compiled and linked into a shared library. The function entry point name is used when the function defined by this statement is called.

You must specify this clause if the external function entry point name is different than *function_name* or *specific_function_name*. The string specifies the entry point name of the function, which is its external name.

You can only specify one function entry name per UDF definition.

The syntax for this clause is as follows:

```
F | function_entry_name
```

The character `|` represents a user-defined delimiter.

Follow this procedure to specify a function entry name string:

1. Begin the clause with the character `F`, which indicates that the characters that follow are the function entry name for the UDF.
2. Specify an arbitrary delimiter character to separate the `F` code and the function entry name specified in the string.

3. Specify the function entry name for the UDF.

This is the name of the C or C++ function object code.

Client-Server UDF Code Specification

You must specify whether the UDF code for include files, object files, and source files is located on the client system or on the server. To do this, you specify C for client or S for server.

UDF code for library and package files is always located on the server, but you still must specify the S location code for any library or package file paths you specify.

The character used to separate entities in the path specification is platform-specific when the file is stored on a client system, but not when the file is on the server.

The following table provides more information about writing the client and server location specifications for include, object, or source files:

| IF you specify this location code ... | THEN you ... |
|---------------------------------------|--|
| C | must format the specification in the form required by the client application, for example, BTEQ. Refer to the appropriate client documentation for information about the required form of presentation. |
| S | can use either the SOLIDUS character (/) or the REVERSE SOLIDUS character (\) as the separator in the path specification for all platform operating systems. |

Include Name Clause

The Include Name clause specifies an explicit path to an include file that is to be used for this UDF definition.

The syntax for this clause is as follows.

```
CI | name_on_server | include_name
```

or

```
SI | name_on_server | include_name
```

The character | represents a user-defined delimiter.

Perform the following procedure for specifying an include file name string.

1. Begin the clause with the appropriate client or platform location code.

| IF you specify this code ... | THEN the source or object code for the function is stored on the ... |
|------------------------------|--|
| C | client. |
| S | server. |

2. Type the character I to indicate this is an include file specification.
3. Specify an arbitrary delimiter character to separate the I code and the *name_on_server* variable specified in the string.
4. Specify the name assigned to the include file, without the .h extension, on the server. The server adds the .h extension.

All *names on server* must be unique among the UDFs and external procedures created within the same database. If the CREATE/REPLACE FUNCTION definition includes a nonunique *name_on_server* specification, the system does not create it.

The C or C++ source must have an include statement in the following form:

```
#include <name_on_server.h>
```

5. Specify your delimiter character to separate the *name_on_server* from the *include_name*.
6. Specify the path and name of the include file.

| IF the include file is on the ... | THEN you ... |
|-----------------------------------|--|
| client | must format the specification in the form required by the client application, for example, BTEQ. Refer to the appropriate client documentation for information about the required form of presentation. |
| server | can use either the SOLIDUS character (/) or the REVERSE SOLIDUS character (\) as the separator in the path specification for all platform operating systems. |

Library Name Clause

The Library Name clause specifies the name of a non-standard library file on the server that would not normally be linked with the UDF being defined.

The syntax for this clause is as follows.

```
SL;library_name
```

The character ; represents a user-defined delimiter.

Perform the following procedure for specifying a library file name string.

1. Begin the clause with the character S to indicate this is a server specification.
2. Type the character L to indicate this is a library file specification.

- Specify an arbitrary delimiter character to separate the L code and the library name specified in the string.
- Specify the name assigned to the non-standard library file on the server. The server automatically adds prefix or suffix values as needed.

The path must name a server library that is already installed on the system.

You can use either \ or / characters to specify the path for all operating systems.

Object File Name Clause

The Object File Name clause specifies an explicit path to an object file that is to be used for this UDF definition.

The syntax for this clause is either of the following.

- CO_i*name_on_server*_i*object_name*
- SO_i*name_on_server*_i*object_name*

The character _i represents a user-defined delimiter.

Perform the following procedure for specifying an object file name string.

- Begin the clause with the appropriate client or server location code.

| IF you specify this code ... | THEN the source or object code for the function is stored on the ... |
|------------------------------|--|
| C | client. |
| S | server. |

- Type the character O to indicate this is an object file specification.
- Specify an arbitrary delimiter character to separate the O code and the *name_on_server* variable specified in the string.
- Specify the name assigned to the object file on the server. Do not specify an extension for this file name.

All *names on server* must be unique among the UDFs and external procedures created within the same database. If the CREATE/REPLACE FUNCTION definition includes a nonunique *name_on_server* specification, the system does not create it.

- Specify your delimiter character to separate the *name_on_server* from the *object_name*.
- Specify the path and name of the object file.

| IF the object file is on the ... | THEN you ... |
|----------------------------------|--|
| client | must format the specification in the form required by the client application, for example, BTEQ. Refer to the appropriate client documentation for information about the required form of presentation. |

| IF the object file is on the ... | THEN you ... |
|----------------------------------|--|
| server | can use either the SOLIDUS character (/) or the REVERSE SOLIDUS character (\) as the separator in the path specification for all platform operating systems. |

Package Name Clause

The Package Name clause specifies an explicit path to a package file that is to be used for this UDF definition. Packages are libraries that can contain UDFs and other functions to be called by a UDF.

A typical package includes a function library as well as a script that contains all the necessary SQL DDL and DCL statements to create the functions and make them available to users.

A package is a shared object file with a .so extension for Linux systems.

Package files must be distributed to all server nodes.

You cannot specify the package option with any other encoded path string clause, but you can specify it with a function entry point name string (see [EXTERNAL NAME Clause](#)).

To distribute a third-party package to a database node, use their documented installation procedure. If you are installing a package developed by your programming staff, you can use any of the following general procedures:

- Install the package in a specific directory and then use PCL to distribute it to all nodes.
- FTP the package from a client system.
- Use the Teradata-supplied procedure named *installsp*, which is stored in *SYSLIB*.

For instructions on how to use *installsp*, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The syntax for this clause is as follows.

```
SP j package_name
```

The character *j* represents a user-defined delimiter.

Perform the following procedure for specifying a package file name string.

1. Begin the clause with the character S to indicate this is a server specification.
2. Type the character P to indicate this is a package file specification.
3. Specify an arbitrary delimiter character to separate the P code and the package name specified in the string.
4. Specify the path and name assigned to the package file on the platform. You must specify the appropriate file extension for the platform operating system.

The package file extension must be .so for Linux systems.

The path must name a package that is already installed on the system and that has been distributed to all its nodes.

You can use either \ or / characters to specify the path for all platforms.

The maximum package path length is 256 characters.

Source File Name Clause

The Source File Name clause specifies the location of an object file that is to be used for this UDF definition.

The syntax for this clause is either of the following.

- CS *name_on_server* ; *source_name*
- SS *name_on_server* ; *source_name*

The character ; represents a user-defined delimiter.

Perform the following procedure for specifying a source file name string.

1. Begin the clause with the appropriate client or server location code.

| IF you specify this code ... | THEN the source or object code for the function is stored on the ... |
|------------------------------|--|
| C | client. |
| S | server. |

2. Type the character S to indicate this is a source file specification.
3. Specify an arbitrary delimiter character to separate the S code and the *name_on_server* variable specified in the string.
4. Specify the name assigned to the source file on the platform.

All *names on server* must be unique among the UDFs and external procedures created within the same database. If the CREATE/REPLACE FUNCTION definition includes a nonunique *name_on_server* specification, Vantage does not create it.

5. Specify your delimiter character to separate the *name_on_server* from the *source_name*.
6. Specify the path and name of the source file.

You can use either \ or / characters to specify the path for all platforms.

External String Literal Examples

The following examples demonstrate various UDF external string literals.

Example1: External String Literal Examples

The following example indicates that the source is to be obtained from the client, from absolute directory UDF Source. The file name on the platform and the source file name are both *sales.c*. The function entry name is *sales1* and the function is to be compiled with debug symbols.

```
'CS|sales|C:\UDF Source\sales.c|F|sales1|D'
```

Example 2: External String Literal Examples

In the following example, the object is to be obtained from the client. `udfdev/imagef.o` is the relative path to `udfdev/imagef.o` from the home or current client directory for the logged on user. The file name on the Teradata server is `img.o`. The name of the object it retrieves is `imagef.o`, and the function entry name for the UDF is `img_match`.

```
'CO|img|/udfdev/imagef.o|F|img_match'
```

Example 3: External String Literal Examples

The following example indicates that the header file `udf_types.h` and the C source file `stdvar.c` to be used for the UDF are to be found on the client. `/headers/udf_types.h` is the relative path from the home or current client directory for the logged on user to the header file and `/src/stdvar.c` is the relative path to the C source file. The function name in the C source code is called `stdvar`. Both files have the same name on the platform.

```
'CI|udf_types|headers/udf_types.h|CS:stdvar|src/stdvar.c|F|stdvar'
```

The following table summarizes the naming issues for the EXTERNAL NAME clause and its various components.

| Function Name | Source File Name | Function Name in DBC.TVM | C/C++ Function Name | Comments |
|---|---|-------------------------------|-------------------------------|--|
| <i>function_name</i> only | <i>function_name</i> | <i>function_name</i> | <i>function_name</i> | <i>function_name</i> must be unique within its database. If you add a new function that has the same <i>function_name</i> within the sam database, then you must specify a different <i>specific_function_name</i> to make the two functions distinct. |
| <i>function_name</i> and <i>specific_function_name</i> | <i>specific_function_name</i> | <i>specific_function_name</i> | <i>specific_function_name</i> | <i>specific_function_name</i> must be unique within its database. |
| <i>function_name</i> and <i>external_function_name</i> | <i>external_function_name</i> | <i>function_name</i> | <i>function_entry_name</i> | <i>specific_function_name</i> must be unique within its database. If you add a new function that has the same <i>function_name</i> within the sam database, then you must specify a different <i>specific_function_name</i> to make the two functions distinct. |
| <i>function_name</i> and <i>function_entry_name</i> as part of 'string' | <i>source_name</i> as specified in 'string' | <i>function_name</i> | <i>function_entry_name</i> | |

| Function Name | Source File Name | Function Name in DBC.TVM | C/C++ Function Name | Comments |
|---|---|-------------------------------|--|---|
| <i>function_name</i> and 'string' but no <i>function_entry_name</i> | | <i>function_name</i> | <i>function_name</i> | |
| <i>function_name</i> and <i>specific_function_name</i> and <i>function_entry_name</i> but not as 'string' | <i>external_function_name</i> | <i>specific_function_name</i> | <i>external_function_name</i> | <i>specific_function_name</i> must be unique within its database. |
| <i>function_name</i> and <i>specific_function_name</i> and <i>function_entry_name</i> as 'string' | <i>source_name</i> as specified in 'string' | | <i>function_entry_name</i> if F option is specified | |
| | | | <i>specific_function_name</i> if F option is not specified | |

UDF Default Location Paths

Listed below are the default path locations used by UDFs for the following purposes:

- Store UDF source files
- Compile UDF source files
- Store .so files
- Store shared memory files

The starting path is determined by the PDE config or temp directory:

- PDE Config directory: `pdepath -c`
- PDE Temp directory: `pdepath -S`

The following table documents the default directory paths for these resources and activities.

| Files/ Directories | Directory Path | Description |
|-----------------------|--|--|
| Header file | <code>/etc/opt/teradata/ tdconfig/Teradata/tlbs_ udf/usr/</code> | Header Files are located along with the source files. Header file <code>sqltypes_td.h</code> must be specified with an include directive in the UDF source. You can copy this file if you code or compile the UDF outside of the database. |

| Files/ Directories | Directory Path | Description |
|-------------------------|---|--|
| | | If you supply a header file, the file should be specified in the EXTERNAL NAME clause string encoded with the 'I' symbol. |
| Source directory path | /etc/opt/teradata/ tdconfig/Teradata/tdbs_ udf/usr/ | Default directory to search for source files. If the source or object file is on the server in this directory, you can specify the relative path from this directory for any server components specified in the external name string, such as: <ul style="list-style-type: none"> • Include • Object • Package • Source |
| Compiler path | /usr/bin/gcc | Default directory to search for the C/C++ compiler. |
| Linker path | /usr/bin/ld | Default directory to search for the C/C++ linker. |
| Compiler temporary path | /var/opt/teradata/tdtemp/ UDFTemp/ | Temporary directory where UDFs are compiled Any files needed for the compilation process are moved here. This includes source files from the server or client as well as object and header files, if needed. Temporary compilation directories only exist during the duration of a compilation. |
| UDF library path | /etc/opt/teradata/ tdconfig/udflib/ | Read-only directory where dynamically linked libraries are stored. |
| UDF server memory path | /var/opt/teradata/tdtemp/ udfsrv/ | Directory where shared memory files used for the execution of protected mode UDFs are stored. |

UDF .so File Linkage Information

There is only one UDF .so file per application category per database per node.

| Application Category | .so File Format |
|----------------------|--------------------------------|
| > 0 | libudf_dbid_librevno_AppCat.so |
| = 0 | libudf_dbid_librevno.so |

When you create, replace, or drop a UDF, the UDF .so file for that database must be relinked with all the other UDF object files and then redistributed to all the nodes. The 5607 warning message is normal.

External Security Clause

This clause is mandatory for all functions that perform operating system I/O. Not specifying this clause for a function that performs I/O can produce unpredictable results and even cause the database, if not the entire system, to reset. See [CREATE AUTHORIZATION](#) and [REPLACE AUTHORIZATION](#).

The *authorization_name* is an optional Teradata extension to the ANSI SQL:2011 standard.

- The external security authorization associated with the function must be contained within the same database as the function.
- When a function definition specifies EXTERNAL SECURITY DEFINER, then that function executes under the OS user associated with the specified external authorization using the context of that user.

| UDF Mode | OS User |
|-----------|---|
| Protected | <i>tdatuser</i> , which must be a member of the <i>tdatudf</i> OS group. |
| Secure | OS user assigned to an authorization name using the CREATE AUTHORIZATION statement. The specified OS user must belong to the <i>tdatudf</i> OS group. Contact your Teradata technical support representative if you need to change this for any reason. |

The following rules apply:

- If you do not specify an authorization name, you must create a default DEFINER authorization name before a user attempts to execute the function.
- If you have specified an authorization name, an authorization object with that name must be created before you can execute the function.

The system returns a warning message to the requestor when no authorization name exists at the time the UDF is being created.

Related Information

See the following documents for more information about coding and using external functions.

- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145
- *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210
- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- *Teradata Vantage™ - Data Types and Literals*, B035-1143
- *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146

See *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 for detailed information about row-level security and how it is enforced using scalar UDFs.

CREATE FUNCTION (Table Form)

Relationship Among UDFs, Table UDFs, and External Procedures.

UDFs, table UDFs, methods, and external procedures are specific variations of one another and share most properties in common.

Definition of a Table Function

A table function is a user-defined function that returns a multirow relational table, one row at a time to the SELECT request that calls it. You define the structure of the table to be returned in one of two ways:

- If you know the number of columns that will be returned before you run the function, then you define the return table structure similarly to the way you define the structure of a persistent table using the CREATE TABLE statement, by specifying its component column names and their respective data types. See [CREATE TABLE](#) and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The complete set of column names and their accompanying data types defines the structure of the table the function returns to the caller.

If one of the specified columns has a LOB data type, then you must define at least one other non-LOB column.

Each column name is limited to a maximum of 128 UNICODE characters with the same restrictions as other object identifiers. For information on object naming, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

- If you do not know the number of columns that will be returned before you run the function, then you define the return table structure using the TABLE VARYING COLUMNS clause and specify a maximum acceptable number of output columns to be returned. The system stops building return table columns if the specified limit is reached.

The maximum number of columns you can specify is 2,048 for the static and dynamic forms of table function.

The returned table is derived from an external source, typically a native operating system file, message queue, or input argument such as a LOB, and is semantically similar to a derived table (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).

Table functions are different from non-table UDFs because they return an entire table to the calling statement, while a non-table UDF returns only a single scalar result.

Typical Uses for Table Functions

The most common use for table functions is to analyze non-relational data or to convert non-relational data to relational data. Possible applications include the following:

- Converting flat file data into relational tables.
- Converting XML and web data into relational tables.
- Analyzing spreadsheet data or converting it into relational tables.

Very informally, you can think of table functions as a facility for creating a view-like mechanism that allows various SQL processing facilities to process and analyze numerous types of non-relational data.

General Limitations and Restrictions on Table Functions

These general restrictions apply to table functions:

- Table functions cannot execute against fallback data when an AMP is down. Once the AMP returns to service, the query can complete as normal.
- 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 external 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.
- You should always run in protected mode if a table function accesses data external to Vantage or causes the OS to consume system resources. This includes anything that causes the OS to allocate system context, including open files, pipes, semaphores, tokens, threads (processes), and so on.

You can expand the number of protected mode servers for table functions from the default value of 2 to a maximum of 20 per AMP or PE vprocs. For more information, see the *cufconfig* utility in *Teradata Vantage™ - Database Utilities*, B035-1102 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147. The minimum is 0.

Protected mode servers consume disk resources as follows:

Amount of disk used = (256 Kbytes)(number of vprocs)(number of protected mode servers)

- For functions running in protected mode, you need at least one protected mode server for each table function you want to run concurrently from separate sessions. For more information, see the *cufconfig* utility in *Teradata Vantage™ - Database Utilities*, B035-1102 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147. A table function reserves one protected mode UDF server per vproc for the duration of the table function step.
- A table function can be deadlocked if too few protected mode servers are configured.

The system does not detect or report this deadlock.

- You cannot specify more than one table UDF in a FROM clause.
- You cannot invoke a table function in a FROM clause join condition using the FROM ... JOIN ... ON syntax.

Suppose you create the following table and table function.

```
CREATE TABLE supp_indata (
  a FLOAT,
```

```

    b FLOAT;
CREATE FUNCTION c_mytable (a FLOAT, b FLOAT)
RETURNS TABLE (c3 FLOAT, c4 FLOAT )
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!c_mytable!c_mytable.c';

```

The following SELECT request is not valid because it uses the FROM ... JOIN ...ON syntax.

```

SELECT t1.c4
FROM TABLE (c_mytable(supp_indata.a, supp_indata.b)) AS t1
JOIN supp_indata AS t2 ON t1.c3 = t2.b;

```

However, the following SELECT request that uses slightly different syntax for the join is valid.

```

SELECT t1.c4
FROM TABLE(c_mytable(supp_indata.a, supp_indata.b)) AS t1,
           supp_indata AS t2
WHERE t1.c3 = t2.b;

```

- You cannot create or replace a table function from an embedded SQL application.
- A table function cannot reference a recursive view.
- The size of any row returned by a table function cannot exceed the system limit.

If a row does exceed this limit, the system aborts the request in Teradata session mode or its containing transaction in ANSI session mode, rolls it back, and returns an error message to the requestor.

The system defers the maximum row size check until you invoke the function because it cannot know whether the limit will be exceeded by a table function defined with dynamic result rows until the function runs.

- The maximum number of columns that can be returned by any table function is 2,048.
- When you run in protected mode, all instances of a table function run as *tdatuser*.
- You must troubleshoot code problems outside Vantage.

You can do a limited amount of code checking by building the trace facility into the code with the FNC_Trace_Write or FNC_Trace_Write_DL library calls. See [CREATE GLOBAL TEMPORARY TRACE TABLE](#). For details about these library calls and other troubleshooting methods you can build into table functions, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

- You can specify the TD_ANYTYPE data type as an input parameter for a table function.
- You cannot specify the TD_ANYTYPE data type as the return type for a table function.

- Any non-deterministic elements referenced in the RETURN expression of a UDF can be replaced by values predefined by Teradata Unity. For more information about Teradata Unity, see the Teradata Unity documentation.
- The mandatory way you must specify columns having a BLOB or CLOB data type depends on whether you are specifying a data type for a parameter column or for a RETURN TABLE clause column:

| Clause | LOB Type |
|---------------|-------------------------------|
| Parameter | With an AS LOCATOR phrase. |
| RETURNS TABLE | Without an AS LOCATOR phrase. |

Protected and Unprotected Modes

Vantage executes all table functions that are written in SAS in protected mode.

For further information about protected and unprotected modes, see [Protected and Unprotected Execution Modes](#).

Handling SQL Result Codes

See [UDF Handling of SQL Result Codes](#).

Mandatory Function Attributes

Note that return value data types do not apply to table UDFs.

Parameter Names and Data Types

See [Parameter Names and Data Types](#).

While you can specify the TD_ANYTYPE data type for table function parameters, you cannot specify TD_ANYTYPE as the return data type for a table function.

Note that unlike the case for the RETURNS TABLE clause, you *must* stipulate an AS LOCATOR phrase for any LOB parameter data types you specify in the parameters clause of a table function.

Managing UDF Memory Allocation

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 and *Teradata Vantage™ - Database Utilities*, B035-1102 for information about managing table UDF memory allocation.

Writing a UDF That Uses Dynamic UDT Parameters

See [Writing a UDF That Uses VARIANT_TYPE UDT Parameters](#).

RETURNS TABLE Clause

The basic RETURNS TABLE clause for a table UDF definition is a basic definition for the table to be returned and consists only of a list of the columns and their respective data types.

The dynamic results row specification RETURNS TABLE clause for a table UDF definition provides a method for dynamically defining the number of columns returned by the function when that number cannot be known before you invoke the function. The only parameter you specify for this clause is the maximum number of output columns. The system defines the column names and data types at run time.

Optional Function Characteristics

See [Optional Function Characteristics](#). Note that table functions do not support the specification of the RETURNS NULL ON NULL INPUT option.

Parameter Style Clause

See [Parameter Style Clause](#). Note that the only valid parameter style for table functions is SQL.

Null Call Clause

You can pass a null as an input argument to a table function or return a null in a result row of a table function only when the data type of the corresponding parameter to its Java method is object-mapped (see [Data Type Mapping Between SQL and Java](#)).

If the data type of the corresponding parameter to the Java method of the table function is defined as simple-mapped, then attempts to evaluate a null at runtime fail and return an error to the requestor. This is because simple-mapped data types cannot represent nulls.

See [Null Call Clause](#) for general information about the Null Call clause. Note that the only form of null call supported by table functions is CALLED ON NULL INPUT.

External Body Reference Clause

See the following for information about the components of the External Body Reference clause:

- [External Body Reference Clause](#)
- [EXTERNAL NAME Clause](#)
- [External String Literal](#)
- [Function Entry Name Clause](#)
- [Client-Server UDF Code Specification](#)
- [Include Name Clause](#)
- [Library Name Clause](#)
- [Object File Name Clause](#)
- [Package Name Clause](#)

- [Source File Name Clause](#)

Table UDF Default Location Paths

See [UDF Default Location Paths](#) for information about the default location of UDF-related information.

Table UDF .so Linkage Information

See [UDF .so File Linkage Information](#) for information about .so files for table UDFs.

Restrictions on Calling Table Functions

You can only call a table function in the FROM clause of an SQL SELECT request (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146), which includes calling table functions in the FROM clause of a subquery. Calling a table function from any other clause within a query causes the system to abort the request and returns an error to the requestor.

Restrictions on Declaring an External C++ Table Function

If you specify CPP in the LANGUAGE CLAUSE, then you must declare the main C++ function as extern “C” to ensure that the function name is not converted to an overloaded C++ name. For example:

```
extern "C"
void my_cpp(long int *input_int, long int *result, char sqlstate[6])
{
```

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details.

Restriction on Using a Table Function to Enforce Row-Level Security

You cannot use a table function to enforce row-level security for a security constraint.

About Table Operator UDFs

A table operator is a form of a UDF that can only be specified in the FROM clause of a SELECT request. The database treats table operator UDFs as derived table subqueries.

Name overloading is allowed, but because table operators have no explicit input, the table operator UDFs must have a unique name.

A table operator can be defined to operate in all UDF modes, protected, not protected and secure mode.

The table operator cannot execute on a PE.

You define the table operator using the SQLTABLE parameter style. SQLTABLE refers to the ANSI UDF parameter style. The ANSI UDF parameter style table operator is passed an input row in the indicdata format and returns the output row in the same format.

Note:

Table functions and table operators cannot execute against fallback data when an AMP is down. After the AMP returns to service, the query can complete as normal.

Differences Between Table Operators and Table Functions

The main differences between table functions and user-defined table operators are as follows.

- The input and output to table functions are row sets, or tables, rather than columns. The default format of a row is that of indicdata.
- For a table function, the row iterator is outside the function, and the iterator calls the function.

For a table operator, the iterator is the responsibility of the coder. The table operator itself is just called once.

- For a table operator, the input and output columns are determined by calling the parser function for the table operator at the time the operator is parsed. The operator has full flexibility to determine the output from input and return this information to the parser.
- A table operator can use custom argument clauses to make it more polymorphic.

Because the table operator itself has to do the iteration, it should be structured the way a simple AMP step is structured. The writer is provided with a row read/row write interface.

The parser function is similar to the standard scalar UDF in that it accesses one set of arguments (the input column types, and invocation metadata) and returns the list of output column types. However, like table functions, the output table format can also be determined by user specification at run time.

CREATE FUNCTION and REPLACE FUNCTION (SQL Form)

Relationship Among SQL UDFs, External UDFs, Table UDFs, Methods, and External Procedures

External UDFs, table UDFs, methods, and external procedures are specific variations of one another and share most properties in common. The generic term used to describe all these is *external routine*.

SQL UDFs are invoked in the same way as external and table UDFs, but are written in SQL, so they do not have external elements.

Naming Conventions: Avoiding Name Clashes Among UDFs, UDMs, and UDTs

For UDF, method, and UDT names not to clash, they must observe the following two rules:

- The following column pair must be unique within the *DBC.TVM* table:
 - *DatabaseID, TVMNameI*
- The signature of the following routine must be unique:
 - *database_name.routine_name(parameter_list)*.

UDFs, methods, and UDTs can have the same SQL names as long as their SPECIFIC names and associated routine signatures are different. In the case of UDTs, the SPECIFIC name reference is to the SPECIFIC names of any method signatures within a UDT definition, not to the UDT itself, which does not have a SPECIFIC name.

Rules for Using SQL UDFs

The rules for using SQL UDFs are as follows:

- SQL UDF names are restricted to 128 UNICODE characters in length.
For information on naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.
- The maximum number of parameters you can specify for an SQL UDF is 128.
- Each parameter you specify must have a name associated with it when you create or replace an SQL UDF.
- A parameter can be assigned any data type supported by Vantage except for VARIANT_TYPE or TD_ANYTYPE.
- You must specify an INLINE clause for an SQL UDF, and the inline specification must be TYPE 1.
- You must specify a COLLATION clause for an SQL UDF, and the collation specification must be INVOKER.
- You cannot specify an AS LOCATOR clause for an SQL UDF.

The AS LOCATOR clause is valid only when used with BLOB or CLOB data types specified as parameters in external UDFs.

- You cannot specify an explicit parameter mode for SQL UDFs.

The parameter mode for SQL UDFs always defaults implicitly to IN.

- SQL UDFs can specify both distinct and structured UDTs as input and return parameters.
- If you specify a UDT as either an input parameter type or as the function result type, the current user of the function must have either the UDTUSAGE privilege on the *SYSUDTLIB* database or the UDTUSAGE privilege on the specified UDT.
- SQL UDFs support the same RETURN types as Vantage supports for external UDFs with the exception of the following RETURN types.
 - TABLE
 - TD_ANYTYPE
 - VARIANT_TYPE
- You cannot specify the FORMAT column attribute for a RETURN expression data type. Otherwise, the system returns an error to the requestor.
- You cannot specify the CAST FROM RETURN clause option for SQL UDFs.
- The LANGUAGE clause of an SQL UDF must specify either LANGUAGE SQL or not be specified (the default value for the LANGUAGE clause of an SQL UDF is SQL).
- You cannot specify a PARAMETER STYLE clause for an SQL UDF.
- The SQL DATA ACCESS clause for an SQL UDF must specify CONTAINS SQL.
- You cannot specify a function CLASS clause for an SQL UDF.
- You cannot specify an EXTERNAL or EXTERNAL SECURITY clause for an SQL UDF.
- You can specify an SQL SECURITY clause for an SQL UDF.

This is the default.

The only valid option is DEFINER.

- You must specify an INLINE TYPE clause.

The only valid option is 1.

- You must specify a COLLATION clause.

The only valid option is INVOKER.

- The only SQL statement you can specify within the definition of an SQL UDF is RETURN.
- The value returned by the RETURN statement for an SQL UDF must be a scalar value.
- The SQL expression returned by a RETURN statement cannot contain any explicit table references, nor can it be or contain a scalar subquery.
- You can invoke an SQL UDF from any of the following SQL statement clauses, functions, expressions, operators, or logical predicates.

If an item is not listed, it cannot be used to invoke an SQL UDF.

7: CREATE FUNCTION - CREATE GLOBAL TEMPORARY TRACE TABLE

| Statement | Clause, Function, Expression, or Logical Predicate | | See this document for further information ... |
|----------------------|---|--|---|
| ABORT | WHERE | | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |
| DELETE | WHERE | | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |
| INSERT | VALUES | | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |
| INSERT ... SELECT | <ul style="list-style-type: none"> ◦ GROUP BY ◦ HAVING ◦ ON ◦ ORDER BY ◦ QUALIFY | <ul style="list-style-type: none"> ◦ SAMPLE ... WHEN ◦ select list ◦ WHERE ◦ WITH ... BY | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |
| MERGE | <ul style="list-style-type: none"> ◦ INSERT VALUES ◦ ON | <ul style="list-style-type: none"> ◦ SELECT ◦ UPDATE SET | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |
| ROLLBACK | WHERE | | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |
| SELECT | <ul style="list-style-type: none"> ◦ GROUP BY ◦ HAVING ◦ ON ◦ ORDER BY ◦ QUALIFY | <ul style="list-style-type: none"> ◦ SAMPLE ... WHEN ◦ select list ◦ WHERE ◦ WITH ... BY | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |
| UPDATE | <ul style="list-style-type: none"> ◦ SET ◦ WHERE | | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |
| Wherever valid | Argument to a method or external UDF. | | <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144 |
| Wherever valid | <ul style="list-style-type: none"> ◦ CASE (Searched Form) ◦ CASE (Valued Form) | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145 |
| Wherever valid | COALESCE | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145 |
| Wherever valid | NULLIF | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145 |

7: CREATE FUNCTION - CREATE GLOBAL TEMPORARY TRACE TABLE

| Statement | Clause, Function, Expression, or Logical Predicate | | See this document for further information ... |
|----------------|---|---|---|
| Wherever valid | Aggregate and Window Aggregate Functions | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145 |
| Wherever valid | <ul style="list-style-type: none"> ◦ AVG ◦ CORR ◦ COUNT ◦ COVAR_POP ◦ COVAR_SAMP ◦ GROUPING ◦ KURTOSIS ◦ MAX ◦ MIN ◦ REGR_AVGX ◦ REGR_AVGY ◦ REGR_COUNT | <ul style="list-style-type: none"> ◦ REGR_INTERCEPT ◦ REGR_R2 ◦ REGR_SLOPE ◦ REGR_SXX ◦ REGR_SXY ◦ REGR_SYY ◦ SKEW ◦ STDDEV_POP ◦ STDDEV_SAMP ◦ SUM ◦ VARPOP ◦ VAR_SAMP | |
| Wherever valid | Ordered Analytic Functions | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145 |
| | <ul style="list-style-type: none"> ◦ CSUM ◦ MAVG ◦ MDIFF ◦ MLINREG ◦ MSUM | <ul style="list-style-type: none"> ◦ PERCENT_RANK ◦ QUANTILE ◦ RANK ◦ ROW_NUMBER | |
| Wherever valid | Logical Predicates | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145 |
| | <ul style="list-style-type: none"> ◦ [NOT] BETWEEN ◦ [NOT] IN | <ul style="list-style-type: none"> ◦ [NOT] LIKE ◦ OVERLAPS | |
| Wherever valid | Attribute Functions | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145 |
| | <ul style="list-style-type: none"> ◦ BYTES ◦ CHARACTER_LENGTH ◦ CHARACTERS ◦ FORMAT | <ul style="list-style-type: none"> ◦ OCTET_LENGTH ◦ TITLE ◦ TYPE | |
| Wherever valid | Hash-Related Functions | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145 |
| | <ul style="list-style-type: none"> ◦ HASHAMP ◦ HASHBAKAMP | <ul style="list-style-type: none"> ◦ HASHBUCKET ◦ HASHROW | |
| Wherever valid | Arithmetic Operators | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145 |
| | <ul style="list-style-type: none"> ◦ ABS ◦ CASE_N ◦ EXP | <ul style="list-style-type: none"> ◦ NULLIFZERO ◦ RANGE_N ◦ SQRT | |

7: CREATE FUNCTION - CREATE GLOBAL TEMPORARY TRACE TABLE

| Statement | Clause, Function, Expression, or Logical Predicate | | See this document for further information ... |
|----------------|--|--|---|
| | <ul style="list-style-type: none"> LOG LN | <ul style="list-style-type: none"> WIDTH_BUCKET ZEROIFNULL | |
| Wherever valid | Trigonometric Functions | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates, B035-1145</i> |
| | <ul style="list-style-type: none"> ACOS ASIN ATAN ATAN2 | <ul style="list-style-type: none"> COS SIN TAN | |
| Wherever valid | Hyperbolic Trigonometric Functions: | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates, B035-1145</i> |
| | <ul style="list-style-type: none"> ACOSH ASINH ATANH | <ul style="list-style-type: none"> COSH SINH TANH | |
| Wherever valid | DateTime Functions <ul style="list-style-type: none"> ADD_MONTHS | <ul style="list-style-type: none"> EXTRACT | <i>Teradata Vantage™ - SQL Date and Time Functions and Expressions, B035-1211</i> |
| Wherever valid | String Operators and Functions | | <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates, B035-1145</i> |
| | <ul style="list-style-type: none"> CHAR2HEXINT COLUMN_NAME DATABASE_NAME INDEX LOWER POSITION SOUNDEX | <ul style="list-style-type: none"> STRING_CS SUBSTRING TABLE_NAME TRANSLATE TRANSLATE_CHK TRIM VARGRAPHIC | |

- An SQL UDF routine body cannot contain references to join indexes, macros, triggers, or stored procedures.
- Unlike the case for external UDFs, an ALTER FUNCTION request cannot be used with an SQL UDF.
- SQL expressions of any form are valid as arguments for an SQL UDF as long as the SQL expressions are none of the following:
 - Scalar subqueries
 - Boolean expressions
 - Nondeterministic functions (such as RANDOM) and nondeterministic UDFs when the corresponding parameter is specified more than once in the SQL UDF definition.

This differs from external UDFs, where there is no restriction on nondeterministic functions specified in SQL expressions that are passed as arguments to an external UDF.

- The data type of SQL any expression passed as an argument should match the data type specified for its corresponding parameter in the function definition.

If the types do not match, then they must be of the same class, and the data type of the argument passed by the invoker should be less than the size of the data type of the parameter specified in the UDF; otherwise, the request aborts and returns a function not found error to the requestor.

- Any arguments passed to an SQL UDF must be passed using the positional parameter method.
- An SQL UDF can be passed as an argument to another SQL UDF, external UDF, or method.
- An SQL UDF cannot be referenced by the partitioning expression for the primary index of a table, nor can it be referenced by any CHECK constraints declared for a table definition.

This differs from the case for external UDFs, which *can* be referenced by CHECK the constraints declared for a table definition.

- You *cannot* invoke an SQL UDF within the definition of a join index.
- You can invoke an SQL UDF within the definition of a trigger.
- You can invoke an SQL UDF within the definition of both external and SQL procedures.
- You can pass an SQL UDF as an argument to an external procedure, an SQL procedure, or a macro.
- You *cannot* reference an SQL UDF as an ordering, cast, or transform function in a UDT.

This contrasts with the case for an external UDF, which *can* be referenced inside the definition of a UDT.

You cannot invoke an SQL UDF from any of the following DDL statements:

- CREATE CAST (see [CREATE CAST and REPLACE CAST](#))
- CREATE ORDERING (see [CREATE ORDERING and REPLACE ORDERING](#))
- CREATE TRANSFORM (see [CREATE TRANSFORM and REPLACE TRANSFORM](#))
- You *cannot* trace an SQL UDF using a SET SESSION FUNCTION TRACE request.
- Apart from UDTs, there is no object dependency support in SQL UDFs, which means that if an SQL UDF is referenced inside another database object such as a view or macro, dropping the UDF invalidates the macro or view.

Vantage does not make a dependency check before dropping an SQL UDF. The same is true if a database object such as a UDF that is referenced inside an SQL UDF is dropped, which invalidates the SQL UDF.

However, you cannot drop a UDT that is referenced inside a SQL UDF definition unless the SQL UDF definition is altered in such a way that the dropped UDT is no longer referenced, or if you drop the SQL UDF referencing the UDT.

- Self-referencing, forward referencing, and circular referencing by SQL UDFs is not valid.
- You can invoke an SQL UDF within a derived table. The rules for doing this are the same as the rules for invoking an SQL UDF within a SELECT request (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details).

- You can invoke an SQL UDF within a view definition. The rules for doing this are the same as the rules for invoking an SQL UDF within a SELECT request (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details).
- An SQL UDF can be invoked in the WITH RECURSIVE request modifier of a recursive query. The rules for doing this are the same as the rules for invoking an SQL UDF within a SELECT request (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details).

Protected and Unprotected Execution Modes

SQL UDFs do not have execution modes.

How SQL UDFs Handle SQL Result Codes

The ANSI SQL:2011 standard defines a return code variable named SQLSTATE to accept status code messages. All condition messages are returned to this variable in a standard ASCII character string format.

All SQLSTATE messages are 5 characters in length. The first 2 characters define the message class and the last 3 characters define the message subclass.

For example, consider the SQLSTATE return code '22021'. The class of this message, 22, indicates a data exception condition. Its subclass, 021, indicates that a character not in the defined repertoire was encountered.

Be aware that SQL warnings do *not* abort a request, while SQL exceptions *do* abort a request.

You should ensure that your UDFs always return valid SQLSTATE codes. Vantage does not map SQLSTATE values returned by SQL UDFs to their equivalent SQLCODE values. All SQLSTATE codes generated by SQL UDFs are explicitly mapped to system messages as indicated by the following table:

| IF the UDF returns this type of SQL return code ... | THEN the system maps it to this error message ... | AND displays it in this format ... |
|---|---|---|
| Warning | 7505 | *** Warning 7505 in dbname.udfname: SQLSTATE 01Hxx: |
| Exception | 7504 | *** Error 7504 in dbname.udfname: SQLSTATE U0xxx: |

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for more information about SQL exception and warning codes.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for more information about SQLSTATE and SQLCODE result codes.

Differences Between CREATE FUNCTION and REPLACE FUNCTION

A function can be created or replaced using the same syntax except for the keywords CREATE and REPLACE.

If you specify CREATE, the function must not exist.

If you specify REPLACE, you can either create a new function or replace an existing function with the following restriction: if the function to be replaced was originally created with a specific function name, then that specific function name must be used for the REPLACE FUNCTION statement.

The advantage to using REPLACE is that you can avoid having to grant the EXECUTE privilege again to all users who already had that privilege on the function.

SQL UDFs and Embedded SQL

You can invoke an SQL UDF from embedded SQL requests in the same way you would invoke an SQL UDF from an interactive SQL request. Other than restrictions that apply to the use of SQL expressions in an embedded SQL program (see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148) you can invoke SQL UDFs from an embedded SQL request without any restrictions.

The following example shows one possible way to invoke an SQL UDF, in this case *common_value_expression*, from an embedded SQL request.

```
EXEC SQL FOR 19
  INSERT INTO t1
  VALUES (:var1, df2.common_value_expression(:var2, 2), :var3);
```

SQL UDFs and SQL Cursors

You can invoke SQL UDFs from within an SQL cursor. The rules for invoking an SQL UDF from within a cursor are the same as those for any other expression specified within an SQL cursor (see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details).

This example invokes an SQL UDF in the WHERE clause of a SELECT request in a cursor.

```
DECLARE ex1 CURSOR FOR
  SELECT *
  FROM t1
  WHERE df2.common_value_expression(t1.a1, t1.b1) > 1
  ORDER BY t1.a1;
```

This example invokes an SQL UDF in an UPDATE request in a cursor.

```
DECLARE ex3 CURSOR FOR
  UPDATE t1 SET b1 = df2.common_value_expression(t1.a1, t1.b1)
  WHERE c1 > 10;
```

Passing Parameters to an SQL UDF

When you pass parameters directly to a UDF, Vantage returns parameter metadata in a Statement Info parcel (see *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417 for details). The rules for passing parameters to an external UDF or method also apply to SQL UDFs (see [Rules for Using SQL UDFs](#) and [Function Calling Argument](#)).

In this example, the SQL UDF is not overloaded, so Vantage can resolve the UDF with just a ? parameter passed directly to the UDF. Vantage returns the parameter metadata regarding the UDF parameter in the target position.

```
SELECT myudf(1, ?)
FROM t1;
```

In this example, the SQL UDF is overloaded, so you must explicitly cast the ? parameter to an acceptable value to enable the UDF to be resolved in Prepare mode. If you do not specify such a cast, Vantage cannot resolve the UDF, so the request aborts and the system returns an error to the requestor. For the purposes of parameter metadata, the data type field of the returned metadata is the result type of the cast.

```
SELECT myudf(1, CAST(? AS INTEGER), col3)
FROM t1;
```

SQL User-Defined Functions and Large Objects

The usage characteristics for an SQL UDF with large object parameters or a large object return value are the same as the usage characteristics for any other UDF. In general, you can specify a UDF that accepts a LOB value as an argument in any context in which a UDF is otherwise allowed. You can also use an SQL UDF that returns a LOB value in any context in which a value of that type is appropriate and a UDF is otherwise allowed.

As with other functions and operators, Vantage can apply automatic type conversions to the arguments or the return values of an SQL UDF. You should be careful about the possible performance implications of automatic type conversions with large objects used as UDF parameters.

For example, a function whose formal parameter is a BLOB type could be passed a VARBYTE column as the actual argument. Vantage converts the VARBYTE value into a temporary BLOB and then passes that to the SQL UDF. Because even a temporary BLOB is stored on disk, the performance cost of the conversion is significant. To avoid this, you should consider creating an overloaded function that explicitly accepts a VARBYTE argument.

Another possible cause of undesired conversions is truncation. Declared length is part of the data type specification.

Function Identifiers

SQL UDF function names (see [Function Name](#)) are distinct from SQL UDF specific function names.

The following table briefly outlines the differences among the different function identifiers:

| Function Identifier | Syntax Variable Name | Definition |
|------------------------|-------------------------------|---|
| Function name | <i>function_name</i> | The identifier used to call the function from an SQL request. <ul style="list-style-type: none"> If a specific name is assigned to the function, <i>function_name</i> is not the name by which the dictionary knows the function as a database object, and it need not be unique within its containing database or user. If no specific name is assigned to the function, the function name is the data dictionary is <i>function_name</i>. |
| Specific function name | <i>specific_function_name</i> | The identifier used to define the function as a database object in the dictionary table <i>DBC.TVM</i> . |

Function Name

The function name is the identifier used to call the SQL function from an SQL request. It is not necessarily the database object name that is stored in *DBC.TVM*.

| IF a specific name ... | THEN the name stored in <i>DBC.TVM</i> is its ... |
|---------------------------------|--|
| is not assigned to the function | function name. In this case, the function name must be unique within its database. |
| is assigned to the function | specific function name. In this case, the function name need not be unique within its database. |

Though you can give an SQL function the same name as a column, you must be very careful to avoid specifying them ambiguously. For example, suppose *text_find* is the name of an SQL function and is also the name of a column in the *sales* table. If the column name is followed with a database-style data type specification as in the following example, then the system assumes that *text_find* is a reference to the function named *text_find*, and *not* a reference to the identically named column *text_find*.

```
text_find(FLOAT),
```

There are two ways to make your request unambiguous.

- Use the ANSI syntax for CAST to make an explicit declaration of data type rather than a function parameter. For example:

```
CAST (text_find as FLOAT)
```

- Qualify the column name fully. For example:

```
sales.text_find (FLOAT),
```

In this example, *sales* is the table that contains the column named *text_find*.

You can precede the SQL function name with its containing database or user name if the function is created in a different database or user than your default. The scope of the name is the database or user in which it is contained.

A function name need not be unique: several functions can have the same function name. This is referred to as function name overloading (see [Function Name Overloading](#)). If you overload function names, the parameter type specifications among the various overloaded function names must be sufficiently different to be distinguishable. See [Function Name Overloading](#) for a list of the rules the system uses to determine the uniqueness of a function by its parameters.

Function Calling Argument

The function calling argument is any simple SQL expression, including, but not limited to, constant values, column references, host variables, the NEW VARIANT_TYPE UDT constructor expression (see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210), or an expression containing any of these, including expressions containing UDFs.

When you call an SQL function, and that function is not stored in either your default database or in database *SYSLIB*, you must fully qualify the function call with a database name. If your default database and *SYSLIB* both contain functions matching the name of the called function, then the system references the UDF in your default database. This is the only exception that requires an explicit qualification for *SYSLIB*.

The argument types passed in the call must be compatible with the parameter declarations in the function definition of an existing function. If there are several functions that have the same name, and no qualifying database is specified, then the particular function that is picked is determined by the following process:

1. Vantage searches the list of built-in functions.

If the called function has the same name as a built-in function, the search stops and that function is used.

If a candidate function is not found, proceed to stage 2.

2. Vantage searches the list of SQL and external function names in the default user database.

Candidate functions are those having the same name and number of parameters as specified by the function call as well as the best fit based on their parameter type.

If a candidate function is not found, proceed to stage 3.

3. Vantage searches the list of function names in the SYSLIB database.

Candidate functions are those having the same name and number of parameters as specified by the function call as well as the best fit based on their parameter type.

If a candidate function cannot be located, the system returns an error to the requestor.

The rules for invoking an SQL UDF using an SQL expression are as follows:

- An SQL expression passed as an argument must not be a Boolean value expression, that is, a conditional expression.

This is also true for external UDFs.

- Nondeterministic SQL expressions, meaning expressions that use either random functions or nondeterministic UDFs, or both, that are passed as an argument must not correspond to a parameter that is used more than once in the RETURN clause of an SQL UDF.
- An SQL expression passed as an argument cannot be a scalar subquery.

The rules for selecting a best fit candidate user-defined function once its containing database has been determined are described in [Function Name Overloading](#).

Function Name Overloading

Function names need not be unique within a function class; however, functions from two different function classes cannot have the same name within the same database or user. Vantage uses the parameter types of identically named functions to distinguish among them, so it is imperative that the parameter types associated with overloaded function names be sufficiently different to be distinct.

The system uses the precedence order for compatible type parameters to determine which function is to be invoked when several functions having the same name must be differentiated by their parameter types.

Vantage follows a set of parameter rules to determine the uniqueness of a function name. These rules are provided in the following list:

- The following numeric parameter types listed in order of precedence for determining function uniqueness. For example, a BYTEINT fits into a SMALLINT and a SMALLINT fits into an INTEGER. Conversely, a FLOAT does not fit into an INTEGER without a possible loss of information.

The types are distinct and compatible. Types sharing a number are synonyms and are not distinct from one another.

- BYTEINT
- SMALLINT
- INTEGER
- DECIMAL
- NUMERIC

The size specification for DECIMAL and NUMERIC types does not affect the distinctiveness of a function. For example, DECIMAL(8,3) and DECIMAL(6,2) are identical with respect to determining function uniqueness.

- FLOAT, DOUBLEPRECISION, REAL

- The following character parameter types are listed in order of precedence for determining function uniqueness.

The types are distinct and compatible. Types sharing a character are synonyms and are not distinct from one another. The length specification of a character string does not affect the distinctiveness of a function. For example, CHARACTER(10) and CHARACTER(5) are identical with respect to determining function uniqueness. CHARACTER SET clauses also have no effect on the determination of function uniqueness.

- CHARACTER
 - VARCHAR, CHARACTERVARYING, LONGVARCHAR
 - CHARACTER LARGE OBJECT
- The following graphic parameter types are distinct and compatible. Types sharing a bullet are synonyms and are not distinct from one another.
 - GRAPHIC
 - VARGRAPHIC
 - LONG VARCHAR CHARACTER SET GRAPHIC
 - The following byte parameter types are distinct and compatible:
 - BYTE
 - VARBYTE
 - BINARY LARGE OBJECT
 - All date, time, timestamp, and interval parameter types are distinct.
 - If the number of parameters in identically named existing functions is different or if the function parameter types are distinct from one another in at least one parameter, then the function being defined is considered to be unique.
 - If more than one function has the same *function_name*, then you must supply a *specific_function_name*.
 - You can only overload function names within the same class within a given database. For example, you cannot have a scalar function and an aggregate function with the same *function_name* within the same database.

Parameter Names and Data Types

The parameter list contains a list of variables to be passed to the function.

Function parameters must be explicitly named. Parameter names are standard SQL identifiers.

The data types of SQL expressions passed as arguments must match those of their corresponding parameter data types in the definition of the SQL function. If the types do not match, the system returns an error to the requestor.

You cannot specify TD_ANYTYPE, VARIANT_TYPE, or TABLE as parameter data types for an SQL function.

The only exception to this is the case where the sizes of the parameter and argument data types differ, but the types themselves belong to the same category of data types such as numeric types, character types, DateTime types, and so on. In this case, a mismatch is tolerated as long as the argument size is less than the parameter size. For example, when the parameter type is INTEGER and the argument type is BYTEINT, the types are treated as being compatible. However, if the parameter type were BYTEINT and the argument type were INTEGER, the mismatch would abort the request and return an error to the requestor.

For example, the argument and parameter types are a mismatch in the following SQL UDF definition, but because the size of the argument type is less than the corresponding parameter type, the types are compatible.

```
CREATE FUNCTION df2.myudf (a INTEGER, b INTEGER, c INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a*b*c;

CREATE TABLE t1 (
  a1 BYTEINT,
  b1 INTEGER);
SELECT df2.myudf(t1.a1, t1.b2, 2)
FROM t1;
```

The contrary result is seen for the following example, where the only difference is that column *a1* is defined with an INTEGER type rather than a BYTEINT type. Because the argument type is greater than its corresponding UDF parameter type in the following example, the SELECT request aborts and returns an error to the requestor.

```
CREATE FUNCTION df2.myudf (a BYTEINT, b INTEGER, c INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a*b*c;

CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER);
```

```
SELECT df2.myudf(t1.a1, t1.b2, 2)
FROM t1;
```

To make this example work correctly, you would need to explicitly cast *t1.a1* as `BYTEINT` in the `SELECT` request, because an `INTEGER` argument does not fit into the `BYTEINT` parameter *a* as defined by the UDF *df2.myudf*.

Parameter names are used by the `COMMENT` statement (see [COMMENT \(Comment Placing Form\)](#)) and are reported by the `HELP FUNCTION STATEMENT` statement (see [HELP FUNCTION](#)). Parameter names, with their associated database and function names, are also returned in the text of error messages when truncation or overflow errors occur with a function call.

Each parameter type is associated with a mandatory data type to define the type of the parameter passed to or returned by the function. The specified data type can be any valid data type, including UDTs (see *Teradata Vantage™ - Data Types and Literals*, B035-1143 for a complete list of data types). Character data types can also specify an associated `CHARACTER SET` clause.

For character string types like `VARCHAR` that might have a different length depending on the caller, the length of the parameter in the definition indicates the longest string that can be passed. If there is an attempt to pass a longer string, the result depends on the session mode.

You cannot specify a character data type that has a server character set of `KANJI1`.

The following table summarizes the standard Vantage session mode semantics with respect to character string truncation:

| IF the session mode is ... | THEN ... |
|----------------------------|---|
| ANSI | any pad characters in the string are truncated silently and no truncation notification is returned to the requestor. A truncation exception is returned whenever non-pad characters are truncated. If there is a truncation exception, then the system does not call the function. The relevant indicator values are not set to the number of characters truncated. |
| Teradata | the string is truncated silently and no truncation notification message is returned to the requestor. |

RETURNS Clause

This mandatory clause specifies the data type of the parameter value returned by the function. You can specify any valid data type except `TD_ANYTYPE`, `VARIANT_TYPE` and `TABLE`, including UDTs.

The defined return data type is stored as an entry in `DBC.TVFields` using the name `RETURN0[n]`, where *n* is a sequentially defined integer used to guarantee the uniqueness of the value. This ensures that user-defined parameter names are not duplicated.

SQL Data Access Clause

This mandatory clause specifies whether SQL statements are permitted within the user-written external routine for the function.

The only valid specification is `CONTAINS SQL`.

You can specify the SQL Data Access and Language clauses in any order.

Optional Function Characteristics

The following set of clauses is optional. You can specify all or none of them, and in any order, but each clause defaults to a particular value if it is not specified.

- Specific function name (see [SPECIFIC Function Name Clause](#))
- Deterministic characteristics (see [Deterministic Characteristics Clause](#))
- Null call (see [Null Call Clause](#))

SPECIFIC Function Name Clause

This clause is mandatory if you are using function name overloading, but otherwise is optional.

The clause specifies the specific name of the function. The specific function name is the name placed in the `DBC.TVM` table and has the same name space as tables, views, macros, triggers, join indexes, hash indexes, and stored procedures. A specific function name must be unique within its containing database.

Deterministic Characteristics Clause

This optional clause declares whether the function returns the same results for identical inputs or not.

Vantage supports the following deterministic characteristics options:

- `DETERMINISTIC`
- `NOT DETERMINISTIC`

Vantage must evaluate each row selected by a nondeterministic UDF condition individually because the evaluation of the function can change for each row selected within a query. This qualification includes both specifying `NOT DETERMINISTIC` explicitly and *not* specifying `DETERMINISTIC`, because `NOT DETERMINISTIC` is the default option for the clause.

A predicate that includes a nondeterministic UDF specified on an index results in an all-AMP operation in spite of the index having been specified in the predicate.

Null Call Clause

This optional clause specifies how the function handles null results. The clause has two options:

- `CALLED ON NULL INPUT`

- RETURNS NULL ON NULL INPUT

See [Null Call Clause](#) for more information about what these options signify.

SQL SECURITY Clause

This optional clause specifies the type of security to be used for the function.

The only valid option is DEFINER

INLINE Clause

This mandatory clause indicates the environmental setting for the output of the function.

TYPE 1 is the only valid option.

The TYPE 1 option indicates that the output of the SQL function is controlled by the environment settings of the system.

COLLATION Clause

This mandatory clause indicates the collation setting the function uses to return output when it is invoked.

INVOKER, which means that the collation used to return output is the default collation for the session from which it is invoked, is the only valid option.

RETURN Statement

The RETURN statement, which is an SQL control statement similar to the SQL procedure control language statements LEAVE, LOOP, WHILE, REPEAT, and so on, defines the return expression for an SQL UDF. See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

The following set of rules applies to coding the RETURN statement for an SQL UDF.

- You cannot specify more than one RETURN statement per SQL UDF.
- The SQL expression in the RETURN statement must not contain any references to tables, nor can it contain scalar subqueries.
- The SQL expression in the RETURN statement can specify references to parameters, constants, SQL UDFs, external UDFs, and methods.

It cannot specify a FORMAT attribute.

- If you specify a reference to an SQL UDF in the RETURN statement, it cannot be any of the following types of reference:
 - Self-referencing

A self-referencing SQL UDF references itself within its own definition.

The following self-referencing SQL UDF is not valid. The portion of the RETURN expression that causes the error is highlighted in boldface type.

```

CREATE FUNCTION df2.myudf1 (a INTEGER, b INTEGER)
RETURNS FLOAT
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
SPECIFIC df2.myudf1
CALLED ON NULL INPUT
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN CASE WHEN a < 1
            THEN myudf1 (a + 1 + b)
            ELSE a + b
END;

```

Failure 3807 Object 'myudf1' does not exist.

- Circular referencing

A circular referencing SQL UDF refers to a second SQL UDF, which in turn refers back to the first SQL UDF.

Neither of the following circular referencing SQL UDFs is valid. The portions of the RETURN expressions that cause the errors are highlighted in boldface type.

```

CREATE FUNCTION df2.myudf2 (a INTEGER, b INTEGER)
RETURNS FLOAT
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
SPECIFIC df2.myudf2
CALLED ON NULL INPUT
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a + b * myudf1 (a, b-1);

```

Failure 3807 Object 'myudf1' does not exist.

```

CREATE FUNCTION df2.myudf1 (a INTEGER, b INTEGER)
RETURNS FLOAT
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
SPECIFIC df2.myudf1

```

```

CALLED ON NULL INPUT
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a + b * myudf2 (a, b-1);

```

Failure 3807 Object 'myudf2' does not exist.

- The SQL expression in the RETURN statement must not be a conditional expression whose return type is Boolean because Boolean return types are not supported for SQL UDFs.
- The SQL expression in the RETURN statement can contain any form of arithmetic, string, DateTime functions, and operators that define a scalar result.
- The SQL expression in the RETURN statement can *not* contain aggregate or OLAP function.
- The return expression data type either must match the data type specified in the RETURNS clause of the UDF definition or it must be possible to implicitly cast the return expression to the data type specified in the RETURNS clause; otherwise, the system returns an error to the requestor.

Restriction on Using an SQL Function to Enforce Row-Level Security

You cannot use an SQL function to enforce row-level security for a security constraint.

Related Information

See the following statements for more information about SQL user-defined functions:

- “CREATE FUNCTION (SQL Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- [DROP FUNCTION](#)
- [HELP FUNCTION](#)
- [RENAME FUNCTION \(SQL Form\)](#)
- [SHOW object](#)

CREATE GLOBAL TEMPORARY TRACE TABLE

Function of Global Temporary Trace Tables

Like global temporary tables, global temporary trace tables have a persistent definition, but do not retain rows across sessions.

Global temporary trace tables are not hashed. Instead, each table row is assigned a sequential hash sequence. You can perform an INSERT ... SELECT operation on the table to copy its rows into a normal

hashed table with a primary key, indexes, or additional attributes common to other base table types if you need to do so.

Define the information you want to trace as the input to the UDF. The UDF can then call the `FNC_Trace_Write_DL` function that places the information in a global temporary trace table. If the trace is not turned on, the system does no function traceback, though there is still some overhead because the system still calls the UDF. See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 and [SET SESSION FUNCTION TRACE](#).

Run the trace UDF in non-protected mode once it has been debugged to ensure that it executes in-line with the procedure. A premature end of the procedure or a rollback of the transaction has no impact on the global temporary trace table, and its contents are not deleted until the session logs off. Enabling function traceback makes your procedure run more slowly because each trace call forces the data to be written into the trace table. Nothing is buffered to ensure that nothing is lost while you are testing a function.

Rules and Limitations for Global Temporary Trace Tables

Because global temporary trace tables are not hashed, they have many definition and manipulation restrictions that distinguish them from ordinary global temporary tables (see [Global Temporary Tables](#)).

The following rules and limitations apply to global temporary trace table columns. Any other columns are optional and user-defined.

- You cannot define global temporary trace tables as SET tables. All global temporary trace tables are restricted to the MULTiset table type and cannot be altered to be SET tables.
- You cannot specify fallback characteristics for a global temporary trace table. They are always created as NO FALLBACK by default and cannot be altered to be FALLBACK tables.
- You cannot define *any* indexes for the table, including primary, primary AMP, secondary, join, and hash indexes.

This restriction explicitly includes both partitioned and nonpartitioned primary indexes. You also cannot define a global temporary trace table using the NO PRIMARY INDEX option.

- You cannot specify partitioning for a global temporary trace table.
- You cannot specify default column values for the table.

The following list describes the other rules and limitations for global temporary trace tables.

- If you try to use the same trace table to simultaneously trace a UDF that runs on an AMP and an external procedure or UDF that runs on a PE, the sequence number written to the second column is not in true sequential order. This is because the AMP bases the next sequence number on the sequence of the last row of the trace table and adds one to it no matter where the row originated.

The rows inserted into the trace table from the PE are hashed to one AMP based on the PE vproc number and the current sequence number. Therefore, if the current sequence number for the PE is 5 and the trace row is added to AMP 1, then a trace write into that table from AMP 1 has the sequence number 6.

The best practice is to avoid simultaneously tracing on an AMP and PE.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for more information.

- If you specify `ON COMMIT PRESERVE ROWS`, then the system preserves the rows of the materialized trace table after the transaction commits or after it aborts; otherwise, all rows are deleted from the table.

If the trace table was materialized for the session by the transaction that aborts, it is deleted, so in that case, no rows are preserved. Once the trace table is successfully materialized, however, the system retains any rows a UDF had written to it before the abort occurred.

- You cannot join a global temporary trace table with another table.
- You cannot update a global temporary trace table. In other words, you cannot perform `DELETE`, `INSERT`, or `UPDATE` statements against a global temporary trace table, though you can specify `DELETE ALL`.
- You can use `INSERT ... SELECT` operations to insert rows from a global temporary trace table into another table.
- You can select the contents of the trace table for output to a response spool for examination.

The `SELECT` request used for this purpose can specify a `WHERE` clause to determine the criteria used to select the rows. It can also specify an `ORDER BY` clause.

- You can perform `DROP TEMPORARY TABLE` on a global temporary trace table.
- Because there is no primary index, all requests that are normally primary index retrievals become full-table scans.

CREATE HASH INDEX - CREATE ORDERING

These topics provide supplemental usage information about selected SQL DDL statements alphabetically from CREATE HASH INDEX through CREATE ORDERING.

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE HASH INDEX

Use Single-Table Join Indexes Instead of Hash Indexes

You should use equivalent single-table join indexes rather than hash indexes.

Specifying the Primary Index for a Hash Index

A hash index must have a primary index. Unlike other base tables, you do not use the keywords PRIMARY INDEX to specify the primary index for a hash index.

The following table explains how the primary index for a hash index is defined:

| Hash Index | Primary Index for the Hash Index Uses this Column Set |
|--|---|
| Does <i>not</i> define a BY clause | Primary index of the base table to which the hash index refers. |
| Specifies a BY clause and an ORDER BY clause | Column set specified in the <i>column_name_2</i> set in the BY clause. These columns must also be defined in the <i>column_name_1</i> set. Rows in a hash index defined with this type of primary index are ordered either by row hash or by numeric value, depending on what you specify in the ORDER BY clause. |

Maximum Number of Hash, Join, and Secondary Indexes Definable Per Data or Join Index Table

Up to 32 secondary, hash, and join indexes can be defined for one user data base table. Each multicolumn NUSI defined with an ORDER BY clause counts as two consecutive indexes in this calculation. See [Why Consecutive Indexes Are Important For Value-Ordered NUSIs](#).

Permitting all secondary index operations for one table to be performed at the same time allows the action to be treated as a single transaction; that is, either all of the indexes are created, or none of them are created.

Fallback and Hash Indexes

Fallback is very important when a system needs to reconstruct data from fallback copies if a hardware read error occurs when it attempts to read the primary copy of the data. When a read error occurs in this case, the file system reads the fallback copy of the hash index subtable rows and reconstructs a memory-resident image of them on their home AMP. This is referred to as Read From Fallback. See *Teradata Vantage™ - Database Design*, B035-1094. Without this feature, the file system fault isolation logic would abort the transaction and, depending on the error, possibly mark the index as being down. See [SET DOWN and RESET DOWN Options](#).

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. To avoid the overhead of this substitution, you should drop and recreate the hash index.

To enable the file system to detect all hardware read errors for hash indexes, you should also set CHECKSUM to ON.

Comparison of Hash and Single-Table Join Indexes

The reasons for using hash indexes are similar to those for using single-table join indexes. Hash indexes optionally be specified to be distributed in such a way that their rows are AMP-local with their associated base table rows and provide an alternate direct access path to those base table rows. In this way, hash indexes similar in function to secondary indexes. Hash indexes are also useful for covering queries without having to access the base table.

The following list summarizes the similarities of hash and single-table join indexes:

- Primary function of both is to improve query performance.
- Both are maintained automatically by the system when the relevant columns of their base table are updated by a delete, insert, or update operation.
- Both can be the object of the following SQL statements: COLLECT STATISTICS (Optimizer Form), DROP STATISTICS, HELP INDEX, or SHOW HASH INDEX.
- Both receive their space allocation from the permanent space of their creator and are stored in distinct tables.
- Both can be hash-ordered or value-ordered.
- Both can be row-compressed, though the method of compression is different for each, and both are different from the method of multivalue compression used for base tables.

For a description of the methods used to compress hash index rows, join index rows, and base table column values, see [Compression of Hash Index Rows](#), [Row Compression of Join Indexes](#), and [Compressing Column Values Using Only Multivalue Compression](#).

Although join index columns can inherit the multivalue compression characteristics of their parent tables, hash index columns cannot. See [Compressing Hash Index Column Values](#).

- Both can be FALLBACK protected.
- Both can be used to transform a complex expression into a simple index column. The transformation permits you to collect statistics on the expressions, which can then be used by the Optimizer to make single-table cardinality estimates when those expressions are specified on base table columns in the predicate of a query. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.
- A join index can specify expressions in its select list, while a hash index has no select list.

- Neither can be queried or directly updated.
- A hash index cannot have a partitioned primary index or primary AMP index, but a single-table join index can.
- Neither can be used to partially cover a query that specifies the TOP *n* or TOP *m* PERCENT option.
- Both share the same restrictions for use with the MultiLoad, Teradata Parallel Transporter, and FastLoad utilities.

The following table summarizes the differences between hash and join indexes:

| Hash Index | Join Index |
|---|---|
| Indexes one table only. | Can index multiple tables. |
| A logical row corresponds to one and only one row in its referenced base table. | A logical row can correspond to either of the following, depending on how the join index is defined: <ul style="list-style-type: none"> • One and only one row in the referenced base table. • Multiple rows in the referenced base tables. |
| Column list cannot specify aggregate or ordered analytical functions. | Select list can specify aggregate functions. |
| Cannot specify an expression in its select list. | Can specify an expression in its select list. |
| Cannot have a secondary index. | Can have a secondary index. |
| Supports transparently added, system-defined columns that point to the underlying base table rows. | Does not add underlying base table row pointers implicitly. Pointers to underlying base table rows can be created explicitly by defining one element of the column list using the keyword ROWID. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. |
| Cannot be specified for a NoPI table. | Can be specified for a NoPI table. |
| Primary index cannot be partitioned. | Primary index of noncompressed row forms can be partitioned. |
| Cannot be defined on a table that also has triggers. | Can be defined on a table that also has triggers. |
| Cannot be defined on a table that also has triggers. | Can be defined on a table that also has triggers. |
| Column multivalue compression, if defined on a referenced base table, is not added transparently by the system and cannot be specified explicitly in the hash index definition. | Column multivalue compression, if defined on a referenced base table, is added transparently by the system with no user input, but cannot be specified explicitly in the join index definition. |
| Index row compression is added transparently by the system with no user input. | Index row compression, if used, must be specified explicitly in the CREATE JOIN INDEX request by the user. |

Hash indexes provide a functional subset of the capabilities of join indexes. A hash index has a functionally equivalent join index. The functionally equivalent join indexes include only the uniqueness part of the ROWID. See CREATE HASH INDEX examples in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Compressing Hash Index Column Values

The database does not automatically transfer column multivalue compression defined on the base table to the hash index definition. You cannot compress hash index column values.

Compression of Hash Index Rows

The Teradata system uses a variety of compression methods, including:

- Logical row compression
- Compression of column values

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

When describing compression of column values, compression refers to the storage of those values one time only in the table header, not in the row itself, and pointing to them by means of an array of presence bits in the row header. The method is called Dictionary Indexing. For additional information, see *Teradata Vantage™ - Database Design*, B035-1094.

This topic is in regard to logical row compression.

When the following things are true, the system automatically compresses the rows of a hash index:

- The ordering column set is the same as the primary index column set of the base table.
- The primary index of the base table is not unique.

Rows having the same values for the order key are compressed into a single physical row having fixed and repeating parts. If the columns do not fit into a single physical row, they spill over to additional physical rows as is necessary.

The fixed part of the row consists of the explicitly-defined columns for the *column_1* list. The repeating part is composed of the remaining columns, whether defined implicitly or explicitly. See *Teradata Vantage™ - Database Design*, B035-1094 for a more detailed description of hash index row compression.

The system only compresses row sets together if they are inserted by the same INSERT statement. This means that rows that are subsequently inserted are *not* appended as logical rows to existing compressed row sets, but rather are compressed into their own self-contained row sets.

Note that while you can compress hash index *rows*, you *cannot* compress individual column values for a hash index. Furthermore, unlike join index column values, hash index column values cannot inherit the compression characteristics of their parent base table.

Collecting Statistics on a Hash Index

You should collect statistics on appropriate columns of a hash index frequently just as you would for any base table or join index. For most applications, you should collect the statistics on base table columns rather than on hash index columns. See [Collect Statistics on Base Table Columns Instead of Hash Index Columns](#).

An important exception to this guideline is the case where a hash index is defined on a complex expression that a predicate expression specifies on the relevant base table column.

A complex expression is defined as any expression that specifies something other than a simple column reference on the right hand side of a predicate. The Optimizer can only use statistics from those complex expressions that can be mapped completely to a hash index expression.

Note:

You cannot collect statistics on a UDT column. This includes UDT columns that are components of an index. The Optimizer uses dynamic AMP sampling information for equality predicates on UDT columns and default selectivity for other predicates on UDT indexes for costing. The dynamic AMP sampling provides limited statistics information about the index. For example, it cannot detect nulls or skew. If a UDT index access path does not show any improved performance, you should consider dropping the index to avoid the overhead involved in its storage and maintenance.

Collecting statistics on the hash index for those expressions enhances the ability of the Optimizer to estimate single-table cardinalities for a query that specifies the base table expressions in its predicate. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Always consider using the SAMPLE options when you collect and recollect statistics on a hash index. See [Reducing the Cost of Collecting Statistics by Sampling](#) for further information about these options and recommendations on how to use them.

When you create a hash index with a BY clause, you can collect and drop statistics on those columns using an INDEX clause to specify the column set.

For example, suppose a hash index has the following definition:

```
CREATE HASH INDEX ord_hidx_8 (o_custkey, o_orderdate) ON orders
  BY (o_custkey, o_orderdate)
  ORDER BY (o_orderdate);
```

Then you can collect statistics on the partitioning columns as shown in the following example:

```
COLLECT STATISTICS ON ord_hidx_8 INDEX (o_custkey, o_orderdate);
```

Note that you can *only* use columns specified in a BY clause definition if you use the keyword INDEX in your COLLECT STATISTICS statement.

A poorer choice would be to collect statistics using the column clause syntax. To do this, you must perform two separate requests.

```
COLLECT STATISTICS ON ord_hidx_8 COLUMN o_custkey;

COLLECT STATISTICS ON ord_hidx_8 COLUMN o_orderdate;
```

You should collect the statistics for a multicolumn index on the index itself rather than on the individual columns because you have more selectivity when you collect statistics on the index columns as a set.

For example, a query with a condition like `WHERE x=1 AND y=2` is better optimized if statistics are collected on `INDEX (x,y)` than if they are collected individually on column `x` and column `y`.

The same restrictions hold for the `DROP STATISTICS` statement.

Collect Statistics on Base Table Columns Instead of Hash Index Columns

The Optimizer substitutes base table statistics for hash index statistics when no demographics have been collected for its hash indexes. Because of the way hash index columns are built, it is generally best not to collect statistics directly on the index columns and instead to collect them on the corresponding columns of the base table. This optimizes both system performance and disk storage by eliminating the need to collect the same data redundantly.

See [Collecting Statistics on a Hash Index](#) for an exception to this recommendation.

You need to collect statistics on the hash index columns if you do not collect the statistics for the relevant base table columns.

Guidelines for Collecting Statistics On Hash Index Columns

The guidelines for selecting hash index columns on which to collect statistics are similar to those for base tables and join indexes. The primary factor to consider in all cases is whether the statistics provide better access plans. If they do not, consider dropping them.

If the statistics you collect produce do not provide better access plans, report the incident to Teradata support.

When you are considering collecting statistics for a hash index, you can consider the index as a special kind of base table that stores a derived result. For example, any access plan that uses a hash index must access it with a direct probe, a full table scan, or a range scan.

When deciding which columns to include in statistics collection, consider the following:

- Consult the following table for execution plan issues related to collecting statistics on a hash index:

| IF an execution plan might involve ... | THEN you should collect statistics on the ... |
|---|---|
| search condition keys | column set that constitutes the search condition. |
| joining the hash index with another table | join columns to provide the Optimizer with the information it needs to best estimate the cardinalities of the join. |

- Consult the following table for index definition issues related to collecting statistics on a hash index:

| IF the hash index is defined ... | THEN you should collect statistics on the ... |
|---|---|
| with a BY clause or ORDER BY clause (or both) | primary index and ordering columns specified by those clauses. |
| without a BY clause | primary index column set of the base table on which the index is defined. |
| without an ORDER BY clause and the ordering column set from the base table is not included in the <i>column_name_1</i> list | order columns of the base table on which the index is defined. This action provides the Optimizer with several essential baseline statistics, including the cardinality of the hash index. |

- If a hash index column appears frequently in WHERE clauses, you should consider collecting statistics on it as well, particularly if that column is the sort key for a value-ordered hash index.
- If a hash join index is defined on complex expressions (complex expression being defined as any predicate expression that specifies something other than a simple column reference on the right hand side of a predicate) that are frequently specified as query predicates written using base table column references, collecting statistics on those columns enhances the ability of the Optimizer to estimate single-table cardinalities for a query that specifies the base table columns because it can use those statistics directly to estimate the selectivity of complex expressions on base table columns specified in the query. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Hash Indexes Are Maintained by Vantage

Hash indexes are automatically maintained by the system when update operations, such as UPDATE, DELETE, or INSERT, are performed on columns in their underlying base tables that are also defined for the hash index. Additional steps are included in the execution plan to regenerate the affected portion of the stored result.

Creating Hash Indexes for Tables With Row-Partitioned Primary Indexes

You cannot create a partitioned primary index for a hash index, but you can create a hash index on a table that has a row-partitioned primary index in some cases. The rules for creating hash indexes on tables defined with a row-partitioned PI are as follows:

- To define a hash index on a row-partitioned PI table, you must also specify an ORDER BY clause that includes an explicit column list.
- The column list cannot contain BLOB, CLOB, Period, or Geospatial columns.
- You cannot define a hash index on a row-partitioned PI table if the partitions for the table are defined on PERIOD bound functions.

For example, suppose you create the following row-partitioned PI table with a partitioning expression based on the BEGIN bound function.

```
CREATE TABLE hash_idx_ppi (
  i  INTEGER
  j  INTEGER,
  vt PERIOD(DATE))
PARTITION BY (CASE_N(BEGIN(vt) < DATE,
                     NO CASE));
```

Suppose you then attempt to create a hash index on *hash_idx_ppi* such as the following.

```
CREATE HASH INDEX hx(i)
ON hash_idx_ppi
ORDER BY VALUES(i);
```

The system returns an error to the requestor indicating that you cannot create an index on a PERIOD column.

- If 3710 errors occur when running queries against a row-partitioned PI table with a hash index, you can increase the amount of memory that is available to the Optimizer by updating the MaxParseTreeSegs DBS Control flags to a value that ensure that it can process any request on the row-partitioned PI table.

| Problem Occurs with Row-Partitioned PI Tables | Suggested Action |
|---|--|
| Without hash or join indexes | Change the value for MaxParseTreeSegs to this value. <ul style="list-style-type: none"> ◦ 2000 (byte-packed format systems) ◦ 4000 (byte-aligned format systems) |
| With hash or join indexes | Contact Teradata support. |

You might find that your query workloads are such that these values need to be increased still further. For information on changing the setting for the MaxParseTreeSegs DBS Control field, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Restrictions and Limitations for NoPI and Column-Partitioned Tables

You cannot create a hash index on a NoPI table or column-partitioned table.

Restrictions and Limitations for Error Tables

You cannot create a hash index on an error table. See [CREATE ERROR TABLE](#).

Restrictions and Limitations for Load Utilities

You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have hash indexes because those indexes are not maintained during the execution of these utilities. If you attempt to load data into base tables with hash indexes using these utilities, an error message returns and the load does not continue.

To load data into a hash-indexed base table, you must drop all defined hash indexes before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Load utilities like Teradata Parallel Data Pump, BTEQ, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, *are* supported for hash-indexed tables.

Restrictions and Limitations for Ordered Analytical Functions

When an ordered analytical function is specified on columns that are also defined for a compressed hash index, the Optimizer does not select the hash index to process the query.

Restrictions and Limitations for Structured UDT Columns

A hash index can contain structured UDT columns, but you must ensure that the ordering function for the UDT never returns a null. If it does, the system returns an error to the requestor.

Restrictions and Limitations for Large Objects

Hash indexes cannot contain BLOB or CLOB columns.

Restrictions and Limitations for Triggers

You cannot define triggers and hash indexes on the same table.

Considerations After a System Reconfiguration

Value-ordered hash and join indexes must be rebuilt after a system reconfiguration. Reconfigurations are performed by Teradata Services personnel.

This is unnecessary for hash-ordered hash indexes.

Restrictions on Creating a Hash Index on a Table Concurrent With Dynamic AMP Sample Emulation on That Table

You cannot create a hash index for a table while that table is subject to dynamic AMP sample emulation. To disable dynamic AMP sampling, contact the Teradata Support Center.

To use dynamic AMP sampling on the table with a new hash index, use the following general procedure:

1. Create the new hash index on the table on the target system.
2. Extract a fresh dynamic AMP sample from the target system.
3. Apply the fresh sample to the source system.

Permanent Journal Recovery of Hash Indexes Is Not Supported

You can use ROLLBACK or ROLLFORWARD statements to recover base tables that have hash indexes defined on them; however, the hash indexes are *not* rebuilt during the recovery process. Instead, they are marked as not valid. You must drop and recreate any such hash indexes before the Optimizer can use them again to generate a query plan.

When a hash index has been marked not valid, the SHOW HASH INDEX statement displays a special status message to inform you that the hash index has been so marked.

Block-Level Compression and Hash Indexes

Hash index rows can be compressed at the data block level or not, depending on whether they have been compressed or decompressed using the Ferret utility. For information about the Ferret utility, see *Teradata Vantage™ - Database Utilities*, B035-1102.

All limits on data block sizes apply to the noncompressed size of a hash index. Block compression does not raise any of these limits, nor does it enable more data to be stored in a single data block than can be stored in an noncompressed data block of the same size.

CREATE INDEX

Locks and Concurrency

The type of lock you specify with a LOCKING modifier does not always match the type of lock applied by the Lock Manager, as the following table describes:

| Lock type you specify for the table containing the index columns | Lock type placed by the system on the table |
|---|---|
| <ul style="list-style-type: none"> • None • WRITE | WRITE |
| <ul style="list-style-type: none"> • ACCESS • READ • SHARE | READ |
| EXCLUSIVE | EXCLUSIVE |

For more information about lock types, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

For more information about the LOCKING modifier, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

The READ or WRITE lock is upgraded to an EXCLUSIVE lock after the index subtables are created, but prior to locking the dictionary and modifying the headers. This improves concurrency by reducing the time SELECT requests against the table are blocked.

If you do not specify an explicit EXCLUSIVE lock, then a CREATE INDEX request on a table can run concurrently with a SELECT request that has an ACCESS or READ lock on the same table until the lock is upgraded to a severity of EXCLUSIVE. When that occurs, the statement that requests the lock first blocks the other.

A CREATE INDEX request holding an EXCLUSIVE lock on a table can run concurrently with a COLLECT STATISTICS statement on that table until the CREATE INDEX statement requests that the lock be upgraded to EXCLUSIVE. When that occurs, the CREATE INDEX statement is blocked until the COLLECT STATISTICS request completes. Any subsequent COLLECT STATISTICS statements on the table are blocked until the CREATE INDEX process completes.

A CREATE INDEX request cannot run concurrently against the same table with an ALTER TABLE statement or another CREATE INDEX statement. The statement that first requests the lock blocks the other.

Collecting Statistics on a Secondary Index

You should collect statistics on your secondary indexes to enable the Optimizer to make the best use of them.

Note:

You cannot collect statistics on a UDT column. This includes UDT columns that are components of an index. The Optimizer uses dynamic AMP sampling information for equality predicates on UDT columns and default selectivity for other predicates on UDT indexes for costing. The dynamic AMP sampling provides limited statistics information about the index. For example, it cannot detect nulls or skew. If a UDT index access path does not show any improved performance, you should consider dropping the index to avoid the overhead involved in its storage and maintenance.

Maximum Number of Indexes

Up to 32 secondary indexes can be defined for one CREATE INDEX statement. Also, up to 32 secondary, hash, and join indexes can be defined for one table. Each multicolumn NUSI created with an ORDER BY clause counts as two consecutive indexes in this calculation (see [Why Consecutive Indexes Are Important For Value-Ordered NUSIs](#)).

Permitting all secondary index operations for one table to be performed at the same time allows the action to be treated as a single transaction; that is, all of the indexes are created, or none of them are created. You can define up to 64 columns for each secondary index you create.

You can also create secondary indexes on user data base tables and join indexes as part of the CREATE TABLE and CREATE JOIN INDEX statements, respectively. See [CREATE TABLE](#) and [CREATE JOIN INDEX](#) for details.

Fallback and Secondary Indexes

You cannot explicitly specify fallback for secondary index subtables. Instead, any secondary indexes defined on a table or join index inherit the fallback properties of their parent table or join index.

Fallback is very important if a system needs to reconstruct data from fallback copies when a hardware read error occurs when it attempts to read the primary copy of the data. When a read error occurs in this case, the file system reads the fallback copy of USI subtable rows and reconstructs a memory-resident image of them on their home AMP. This is referred to as Read From Fallback. See *Teradata Vantage™ - Database Design*, B035-1094. Without this feature, the file system fault isolation logic would abort the transaction and, depending on the error, possibly mark the index as being down. See [SET DOWN and RESET DOWN Options](#). 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 USI subtable data blocks. Read From Fallback does not support NUSI 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. To avoid the overhead of this substitution, you should drop the USI and recreate it.

To enable the file system to detect all hardware read errors for USIs, set CHECKSUM to ON.

System-Defined Unique Secondary Indexes

When you define a PRIMARY KEY or UNIQUE constraint, the system may implement that constraint as a unique secondary index or unique join index.

All system-defined USIs and unique join indexes count against the limit of 32 secondary, hash, and join indexes per user-defined data base table.

Why Consecutive Indexes Are Important For Value-Ordered NUSIs

Value-ordered multicolumn NUSIs created with an ORDER BY clause consume two consecutive index numbers of the allotted 32 index numbers. One index of the consecutively numbered pair represents the column list, while the other index in the pair represents the ordering column.

Consider the following scenario:

1. You create 32 indexes on a table, none of which is value-ordered.
2. You drop every other index on the table, meaning that you drop either all the odd-numbered indexes or all the even-numbered indexes.

For example, if you had dropped all the even-numbered indexes, there would now be 16 odd-numbered index numbers available to be assigned to indexes created in the future.

3. You attempt to create a multicolumn NUSI with an ORDER BY clause.

The system returns an error message because two consecutive index numbers were not available for assignment to the ordered multicolumn NUSI.

You are still able to create 16 additional value-ordered single column NUSIs, non-value-ordered NUSIs, USIs, hash indexes, or join indexes, but you cannot create any ordered multicolumn NUSIs.

To work around this problem, perform the following procedure:

1. Drop any index on the table.
This action frees 2 consecutive index numbers.
2. Create the ordered multicolumn NUSI that failed previously.
3. Recreate the index you dropped to free the consecutive index numbers.

Storage and Performance Considerations for Secondary Indexes

Secondary indexes are stored in subtables separately from their associated data and join index tables. This means that each secondary index subtable that is created requires additional disk space.

Additionally, insert, delete, and update operations require more processing time for tables with secondary indexes because the index subtables must be updated each time an indexed field in an associated row in their parent data or join index table is updated.

Secondary Indexes Can Shorten Search Time

Secondary index values can shorten table search time for multirow selection. Multirow selection is often referred to as set selection. Each row in a secondary index subtable is made up of the index value and one or more row IDs that point to the data row set containing that value. Therefore, when secondary index values are used as conditional expressions in SQL statements, only the row set that contains the specified value is accessed.

CREATE INDEX and the QITS Column of Queue Tables

You cannot create a USI on the QITS column of a queue table. See [CREATE TABLE \(Queue Table Form\)](#).

UNIQUE Versus Nonunique Secondary Indexes

The UNIQUE option creates a USI which, perhaps redundantly, prevents the occurrence of duplicate index values in the indexed base table. Only one row can have a value set in the column set that corresponds to an entry in the subtable of a unique secondary index. The subtable of a unique secondary index is a hashed table, which is very efficient for retrieving a single row.

When the values of nonunique secondary indexes are used to apply complex conditional expressions to very large tables, efficient retrieval is achieved by bit mapping the subtable row IDs.

Use EXPLAIN to Analyze the Usefulness of Secondary Indexes

Before a request is processed, the Optimizer estimates comparative costs to determine whether the use of a secondary index offers savings or is plausible. If not, it uses a different access method. You should test the performance of a secondary index before implementing it.

Use the SQL EXPLAIN request modifier to generate a description of the processing sequence to help determine whether a secondary index should be retained. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Named Indexes

The index name must conform to the usual rules for a Teradata SQL identifier and be unique in the table.

Relationships Between Secondary Indexes and Primary Indexes or Primary AMP Indexes

You can create a USI on the same column set that defines a nonunique primary index or primary AMP index for a table to enforce uniqueness.

The system can use the USI to validate that the primary index or primary AMP index columns are unique. You should only use such a USI as a temporary solution for making a NUPI unique. When you alter a table

in this way, the system returns a warning message and the CREATE TABLE SQL text returned by a SHOW TABLE request is not valid.

You should immediately alter the table using the MODIFY PRIMARY INDEX clause to change its primary index from a NUPI to a UPI. See [ALTER TABLE \(Basic Table Parameters\)](#). The system drops the USI that had been defined on the old NUPI because it is no longer needed to enforce the uniqueness of the primary index.

When this redundant USI has been created, you can then use ALTER TABLE to change the primary index on the table from a NUPI to a UPI. You cannot make this conversion for row-partitioned PIs unless all partitioning columns are also defined in the primary index column list.

You can create a value-ordered NUSI on the same column set that defines the primary index or primary AMP index. This is valid for a row-partitioned PI, regardless of whether the primary index column set includes all the columns used in the partitioning expression.

You can create a *non*-value-ordered NUSI on the same column set that defines the primary index for a row-partitioned PI table when the primary index column set does not include all the columns specified for the partitioning expression.

Defining a USI or NUSI on the same column set as the primary index or primary AMP index for a PPI table may provide better performance for accesses to that table.

Disk I/O Integrity Checking

Vantage assigns secondary index subtables the same level of disk I/O integrity checking that is defined for their associated user data tables or join indexes, so you need not, and *cannot*, create any disk I/O integrity checks for them.

Restrictions and Limitations for Error Tables

You can use the CREATE INDEX statement to add a secondary index to an error table to facilitate scanning its captured data for analysis. See [CREATE ERROR TABLE](#).

You must then drop any of the secondary indexes you have defined before you perform an INSERT ... SELECT or MERGE operation that specifies error logging on the data table associated with the error table.

Restrictions and Limitations for Large Object, Period, and Geospatial Columns Used in a Secondary Index Definition

Secondary indexes cannot contain Period, Geospatial, BLOB, or CLOB columns.

Restrictions and Limitations for Teradata Row-Level Security Columns Used in a Secondary Index Definition

There is no restriction on creating either USI or NUSIs on row-level security columns.

Restrictions on Creating a Secondary Index on a Table Concurrent With Dynamic AMP Sample Emulation on That Table

You cannot create a secondary index for a table while that table is subject to dynamic AMP sample emulation. To disable dynamic AMP sampling, contact the Teradata Support Center.

To use dynamic AMP sampling on the table with a new secondary index, use the following general procedure:

1. Create the new secondary index on the table on the target system.
2. Extract a fresh dynamic AMP sample from the target system.
3. Apply the fresh sample to the source system.

Block Compression and Secondary Indexes

Secondary index subtable rows cannot be block compressed.

CREATE JOIN INDEX

Locks and Concurrency

Join index maintenance is optimized to use localized rowhash or rowkey-level locking whenever possible. Table-level locks are applied when the maintenance is performed using all-AMPs operations such as spool merges.

When the table being updated contains a join index, an EXPLAIN of the UPDATE request illustrates whether an all-AMPs operation and table-level locking are used at the time of the update, or a single-AMP operation with rowhash or rowkey-level locking.

The following is a list of the conditions that support single-AMP updating and rowhash or rowkey-level locking on join indexes when the base table is updated a row-at-a-time. These optimizations may not be applied to complicated indexes when cardinality estimates are made with low confidence or when the index is defined on three or more tables with all its join conditions based on nonunique and unindexed columns.

When a row-at-a-time INSERT on the base table is being performed, inserts that specify an equality constraint on the primary index column set of a table with joins between some non-primary index columns of the table and the primary index columns of another table are optimized to use rowhash-level locks.

As long as the number of rows that qualify is within a 10% of the number of AMPs threshold, and the number of rows resulting from join operations is also within this threshold, this INSERT request does not incur any table-level locks.

If a single-table non-covering join index is defined on a table, the join that is processed by duplicating the qualified rows to join with another table can be processed as follows:

1. A single-AMP retrieve from the first table by way of an equality condition into Spool 2. Spool 2 is hash redistributed by the expression on the right-hand side of the equality condition and qualifies as a group-AMPs spool.
2. A few-AMPs join from Spool 2 to the join index is based on an equality condition, with the result going into Spool 3. Spool 3 is redistributed by a ROWID condition on the first table and also qualifies as a group-AMPs spool.
3. A few-AMPs join back from Spool 3 to the first table on ROWID.

As long as the number of rows that qualify the join is within the 10% threshold, no table-level locks are incurred for the DELETE request.

For more information, see *Teradata Vantage™ - Database Design*, B035-1094.

When a row-at-a-time UPDATE request on the base table is being performed, the following restrictions apply:

- The value for the primary index of the join index must be specified in the WHERE clause predicate of the request.
- The UPDATE request cannot change the primary index of the join index.
- When it is cost effective to access the affected join index rows by means of a NUSI, it is done using rowhash locks and a direct update step. If only a few rows are updated (a few-AMPs operation),

rowhash READ locks are placed on the NUSI subtable for the index rows that are read. Rowhash locks are also applied to the base table using the rowID values extracted from the index rows.

For more information, see *Teradata Vantage™ - Database Design*, B035-1094.

When a row-at-a-time DELETE request on the base table is performed, the following restrictions apply:

- The value for the primary index of the join index must be specified in the WHERE clause predicate of the DELETE request.
- The deleted row must not be from the inner table of an outer join in the CREATE JOIN INDEX statement with the following exceptions:
 - The outer join condition in the join index is specified on a UPI column from the inner table.
 - The outer join condition in the join index is specified on a NUPI column from the inner table.
- When it is cost effective to access the affected join or hash index rows by means of a NUSI, it is done using rowhash-level locks and a direct delete step. If only a few rows are deleted (a few-AMPs operation), rowhash-level READ locks are placed on the NUSI subtable for the index rows that are read. Rowhash-level locks are also applied to the base table using the rowID values extracted from the index rows.

Under all other conditions, a single-row UPDATE operation causes a table-level WRITE lock to be placed on the join index.

If table-level locks are reported in the EXPLAIN text, then consider using set processing approaches with one or more secondary indexes as an alternative.

For more information, see *Teradata Vantage™ - Database Design*, B035-1094.

For information about how locks are placed on join indexes for tactical queries, see *Teradata Vantage™ - Database Design*, B035-1094.

Be aware that you cannot drop a join index to enable Teradata MultiLoad or Teradata FastLoad batch loads until any requests that access that index complete processing. Requests place READ locks on any join indexes they access, and the database defers processing of any DROP JOIN INDEX requests against locked indexes until their READ locks have all been released.

Maximum Number of Indexes Definable Per Data, Hash, or Join Index Table

Up to 32 secondary, hash, and join indexes, in any combination, can be defined for a table.

Each multicolumn NUSI defined with an ORDER BY clause counts as two consecutive indexes in this calculation. See [Why Consecutive Indexes Are Important For Value-Ordered NUSIs](#).

Fallback and Join Indexes

Fallback is very important when a system needs to reconstruct data from fallback copies if a hardware read error occurs when it attempts to read the primary copy of the data. When a read error occurs in this case, the file system reads the fallback copy of the join index subtable rows and reconstructs a memory-resident

image of them on their home AMP. This is referred to as Read From Fallback. See *Teradata Vantage™ - Database Design*, B035-1094. Without this feature, the file system fault isolation logic would abort the transaction and, depending on the error, possibly mark the index as being down. See [SET DOWN and RESET DOWN Options](#).

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. To avoid the overhead of this substitution, you should drop the join index and recreate it.

To enable the file system to detect all hardware read errors for join indexes, set CHECKSUM to ON.

Defining a Unique Primary Index for a Join Index

You can define noncompressed and non-value-ordered single-table join indexes with a unique primary index. A join index that has a unique primary index is referred to as a unique join index. The primary index of a compressed join index must be based on the *column_1* column set, which means there is only one instance of the *column_2* column set for a given instance of *column_1*, so if the primary index column set in the *column_1* column set were unique, there can be no row compression for that join index. Because of this, the ability to define a UPI for a compressed join index does not provide any advantages, so it is not supported.

The row hash value of a value-ordered join index corresponds to the ORDER BY column set instead of the primary index column set, and therefore does not support checking for the uniqueness of a UPI.

You can define a UPI on a join index. Because Vantage checks for uniqueness when new rows are inserted into the index, an INSERT or an UPDATE operation on the base table fails if the inserted or updated rows violate the uniqueness of the join index. Join index creation fails if rows of the base table or tables being indexed violate the uniqueness constraint defined by the UPI of the join index.

The value that you specify in the equality condition of the query forms the primary index key for accessing the unique join index.

For a user-defined unique join index whose primary index is explicitly defined as unique, the coverage testing and the presence of an equality condition on the primary index of the join index guarantee that the join index returns at most a single row, and therefore are sufficient to qualify the index to be used for single-row access by the Optimizer.

For information about how the Optimizer uses unique join indexes as an access path in two-AMP query plans, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

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

Partitioned Tables and Join Indexes

You can create join indexes on a table that is row-partitioned, column-partitioned, or multilevel column-partitioned with one or more row partitioning levels and you can also row partition or column partition a join index. See [Rules and Restrictions for Column-Partitioned Join Indexes](#).

The partitioning expressions for a join index can be based on either numeric or character data. See [Partitioned and Nonpartitioned Primary Indexes](#).

Row partitioning for a join index functions as a constraint on the rows that can be updated in its underlying base table, for example with an SQL DELETE, INSERT, MERGE, or UPDATE operation. Define partitioning expressions for a join index that do not prevent rows from being inserted into, updated, or deleted from the base tables.

You can create a partitioning for a join index that specifies the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions in one or more of its partitioning expressions. See [Partitioning Expressions Using DATE or TIMESTAMP Built-In Functions](#). Use the ALTER TABLE TO CURRENT statement to reconcile the date and timestamp values of these partitioning expressions periodically. See [ALTER TABLE TO CURRENT](#).

Row partitioning is valuable for optimizing the performance of range queries on the partitioning column set, but can be neutral or even suboptimal for non-range queries or for range queries on columns that are not part of the partitioning expression for the join index, so you should test the performance of row-partitioned join indexes carefully before introducing them to your production environment.

Depending on the application, you can use the NO RANGE and UNKNOWN partitions (see [Purpose and Behavior of the NO RANGE and UNKNOWN Partitions](#)) to gracefully handle INSERT, UPDATE, and DELETE requests that cause one or more of the partitioning expressions in the join index to evaluate to null without otherwise aborting such problematic update requests. Update is used here in its generic sense, meaning any delete, insert, update, or merge operation.

You can specify a WHERE clause in the CREATE JOIN INDEX statement to create a sparse join index for which only those rows that meet the condition of the WHERE clause are inserted into the index, or, for the case of a row in the join index being updated in such a way that it no longer meets the conditions of the WHERE clause after the update, cause that row to be deleted from the index. See [Sparse Join Indexes](#).

The process for this activity is as follows:

1. Vantage checks the WHERE clause condition for its truth value after the update to the row.

| Condition Result | Description |
|------------------|---|
| FALSE | Deletes the row from the sparse join index. |
| TRUE | Retains the row in the sparse join index and proceeds to stage b. |

2. Vantage evaluates the new result of the partitioning expression for the updated row.

| Partitioning Expression Result | Description |
|---|--|
| Null or to a value outside the range 1 through 65,035, inclusive, for 2-byte partitioning or outside the range 1 through 9,223,372,036,854,775,807, inclusive, for 8-byte partitioning. | Base table or the sparse join index not updated, error returned. |
| Value between 1 and 65,035, inclusive for 2-byte partitioning or to a value between 1 and 9,223,372,036,854,775,807, inclusive, for 8-byte partitioning. | Stores the row in the appropriate partition, which might be different from the partition in which it was previously stored, and continues processing requests. |

A sparse join index is also helpful because Vantage checks its WHERE clause first when index maintenance is needed and the partitioning expression constraint is only checked if the WHERE clause condition can be satisfied.

You must consider all of the following factors when you are deciding what join indexes to create to support your workloads:

- Data model
 - Table definitions
 - Foreign Key-Primary Key constraints
 - Dimensional hierarchies
- Workload
 - Query patterns
 - Query frequencies
 - Query priorities
- Data maintenance
 - Frequency of maintenance
 - Constraints specified by data maintenance statements and commands

Consider the advantages and the disadvantages of partitioning and join indexes to ensure that you define appropriate indexes for your workloads. Use EXPLAIN request modifiers to verify whether your join index is actually used and to determine whether the Optimizer applies any partition-related optimizations to the query.

Expression evaluation errors, such as divide by zero, can occur during the evaluation of a partitioning expression. The system response to such an error varies depending on the session mode in effect at the time the error occurs.

| Session Mode | Roll Back Performed |
|--------------|--|
| ANSI | Request that contains the aborted statement. |
| Teradata | Transaction that contains the aborted request. |

When you design your partitioning expressions, you should construct them in such a way that expression errors either cannot, or are very unlikely to, occur.

Depending on the circumstances, you can use either an ALTER TABLE statement to revalidate the partitioning of a join index or an ALTER TABLE TO CURRENT statement. See [General Rules for the MODIFY PRIMARY Clause](#) and [Comparing ALTER TABLE TO CURRENT and ALTER TABLE ... REVALIDATE](#).

The ALTER TABLE TO CURRENT statement enables you to refresh the content of a join index without first dropping it and then recreating it. The relative efficiency of ALTER TABLE TO CURRENT requests when compared to the drop and create ALTER TABLE alternative depends on how often you must submit ALTER TABLE requests to update your join indexes as well as the type of DATE, CURRENT_DATE, or CURRENT_TIMESTAMP conditions that are defined in the partitioning.

If you refresh the join index infrequently and the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP condition requires a large volume of old rows to be removed and a large volume of new rows to be inserted, it can be more efficient to drop and recreate the join index.

For obvious cases, such as when the partitioning column set of a join index is updated as a result of a new DATE, CURRENT_DATE, or CURRENT_TIMESTAMP value, ALTER TABLE TO CURRENT requests internally delete all rows from the partitioning expression and then rebuild it using the new DATE, CURRENT_DATE, or CURRENT_TIMESTAMP value.

For the less obvious cases that require a DELETE ALL operation followed by rebuilding the index, the method is more efficient, but ALTER TABLE TO CURRENT requests do not use it, so you should consider using the drop-and-recreate alternative.

When you drop and then rebuild a join index, the existing security privileges and any statistics that have been collected on the join index are also dropped when you drop the join index.

When you must submit an ALTER TABLE request on both a base table and its join indexes, you should consider submitting ALTER TABLE requests for those join indexes that only specify a lower bound DATE, CURRENT_DATE, or CURRENT_TIMESTAMP condition. A lower bound DATE, CURRENT_DATE, or CURRENT_TIMESTAMP condition is a condition that results in rows from the join index being deleted when the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP is refreshed to the latest date or timestamp value, for example, when $j > \text{CURRENT_DATE}$.

Using this method, any join index maintenance required after submitting an ALTER TABLE request against the base table need not be done unnecessarily on those join index rows that would have been deleted by an ALTER TABLE request on the join index.

For additional information about partitioned primary indexes and the rules and recommendations for using them, see [Partitioned and Nonpartitioned Primary Indexes](#) and [Rules and Usage Notes for Partitioned Tables](#), and *Teradata Vantage™ - Database Design*, B035-1094.

Vantage uses both fast path DELETE and fast path deferred partition DELETE operations for the following join index maintenance cases.

- Fast path deferred partition DELETE operations for the following scenarios.
 - Tables that are defined with a nonpartitioned join index

- Nonpartitioned tables defined with a partitioned join index.
- PPI tables defined with a PPI join index.
- Fast path DELETE operations to multiple DELETE ALL steps for implicit multistatement request transactions or multistatement transactions that contain multiple DELETE ALL requests. Vantage uses fast path DELETE operations in both ANSI and Teradata session modes to process the DELETE ALL *table_name* requests if they are the last requests in the transaction that reference *table_name*.
- Fast path DELETE operations for the following cases when tables have a simple or aggregate join index and the index is one of the following:
 - Single-table join index.
 - Multitable join index defined with an inner join.
 - Multitable join index where the table being deleted is the outer table of an outer join.

The preceding criteria apply to the following cases.

- DELETE ALL operations on a table that has a simple or aggregate join index. Vantage uses fast path DELETE operations to process both the base table and its join index.
- Conditional DELETE operations on a table that has a simple or aggregate join index. Vantage uses fast path DELETE operations to process only the join index if the DELETE condition covers the entire index.

The join index partition DELETE operations and DELETE ALL operations greatly facilitate the performance of join index maintenance.

Working Around Optimizer Memory Problems Encountered When Creating Join Indexes for Partitioned Tables

If 3710 errors occur when running queries against a partitioned table with a join index, you can increase the amount of memory that is available to the Optimizer by updating the MaxParseTreeSegs DBS Control flag to a value that ensures that it can process any request on the partitioned table.

The recommended initial setting for the MaxParseTreeSegs field is as follows.

| System Format | MaxParseTreeSegs Setting |
|---------------|--------------------------|
| Byte-packed | 2000 |
| Byte-aligned | 4000 |

You might find that your query workloads are such that these values need to be increased still further.

See *Teradata Vantage™ - Database Utilities*, B035-1102 for information about how to change the setting for the MaxParseTreeSegs field in the DBS Control record. You must contact a Teradata support representative to complete this process.

See *Teradata Vantage™ - Database Design*, B035-1094 for information about some special applications of join indexes to partitioned tables.

Distribution of Join Index Rows

With the exception of column-partitioned NoPI join indexes, the hash distribution of join index rows across the AMPs is controlled by the specified PRIMARY INDEX or PRIMARY AMP INDEX clause. For a description of how Vantage assigns the rows of column-partitioned NoPI database objects to the AMPs, see *Teradata Vantage™ - Database Design*, B035-1094.

For most join indexes, a primary index must be nonunique. UPIs are only permitted for single-table join indexes. If you do not define a primary index or primary AMP index explicitly, the first column defined for the join index is assigned as the primary index, but only if the column is not partitioned. A primary AMP index is always nonunique.

By default, for a primary index, rows are sorted locally on each AMP by the hash code of the primary index column set. To sort by a single column of your choice, use the optional ORDER BY clause in the join index definition. With the ORDER BY clause, you can sort by raw data values rather than the default sort on the hash codes for the values.

Sorting a join index NUSI by data values, instead of hash codes, is especially useful for range queries that involve the sort key. For a comparison of value-ordered NUSIs versus row partitioning for join index range query support, see [Choosing Between a Row-Partitioned Join Index and a Value-Ordered NUSI For Covering Range Queries](#).

Value-ordered NUSI storage provides better performance for queries that specify selection constraints on the value ordering column. NUSI value ordering is limited to a single four-byte numeric or DATE column.

For example, suppose a common task is to look up sales information by sales date. You can create a join index on the sales table and order it by sales date. The benefit is that queries that request sales by sales date only need to access those data blocks that contain the value or range of values that the queries specify.

In the following example, the join index rows are hash-distributed across AMPs using *c_name* as the nonunique primary index and are value-ordered on each AMP using *c_custkey* as the sort key:

```
CREATE JOIN INDEX ord_cust_idx AS
SELECT (o_custkey, c_name), (o_status, o_date, o_comment)
FROM Orders
LEFT JOIN Customer ON o_custkey = c_custkey
ORDER BY o_custkey
PRIMARY INDEX (c_name);
```

If the join index you are creating is not row compressed, then you can define a row partitioning for it by specifying one or more partitioning expressions, which also optimizes the access to the index for range queries on the partitioning column set. See [Partitioned Tables and Join Indexes](#).

Rules and Restrictions for Join Indexes

You can create a join index with column partitioning and a primary AMP index, primary index, or no primary index, with some restrictions such as the join index must be noncompressed and based on a single table.

Join indexes can improve query response times in the following ways:

- Multitable join indexes prejoin tables so that the join result is fetched from the join index directly to answer applicable queries instead of calculating the joins dynamically.
- Single-table join indexes effectively redistribute the rows in their underlying base table by choosing a different partitioning or primary index than the base table to make the joins between the join index and other tables or indexes more efficient.

You can also use a single-table join index to establish an alternate access opportunity.

- Aggregate join indexes preaggregate results so they can be used to answer aggregation queries directly.
- Sparse join indexes store a subset of the rows from their underlying base table set based on a WHERE clause condition. This makes it possible for a smaller join index to be used in answering queries when applicable.

A join index can be a combination of the individual types, such as a join index that has all of the following properties.

- Sparse
- Aggregate
- Multitable

Guidelines for Row-Partitioned PI Join Indexes

You can create a row-partitioned primary index for a noncompressed join index, with some restrictions, such as the join index cannot include the ORDER BY option. A row-partitioned PI join index can improve query performance by leveraging join index and row-partitioned PI benefits. The following two scenarios indicate some possibilities in which an noncompressed join index with a row-partitioned PI might be beneficial.

Scenario A

Assume a star schema with the following tables in the physical data model:

| Dimensional Model Table Type | Corresponding Physical Table and Column Sets | |
|------------------------------|---|-------------------------------------|
| Fact | <i>sales</i> • <i>sale_date</i> • <i>store_id</i> | • <i>prod_id</i> • <i>amount</i> |
| Dimension | • <i>calendar</i> <i>dayofmth</i> | • <i>yr</i> |

| Dimensional Model Table Type | Corresponding Physical Table and Column Sets | |
|------------------------------|--|---|
| | <i>month</i> | |
| | PRIMARY INDEX(<i>yr</i> , <i>month</i>) | |
| | <ul style="list-style-type: none"> <i>product</i> <i>prod_id</i> | <ul style="list-style-type: none"> <i>prod_category</i> |
| | <ul style="list-style-type: none"> <i>org</i> <i>store_id</i> <i>area</i> | <ul style="list-style-type: none"> <i>division</i> <i>business_unit</i> |

You might want to create a single-level row-partitioned aggregate join index with daily summary data to answer some ad hoc queries for this database schema such as the following:

```
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
  GROUP BY sale_date, prod_category, division
  PRIMARY INDEX(sale_date, prod_category, division)
  PARTITION BY RANGE_N(sale_date BETWEEN DATE '1990-01-01'
                        AND      DATE '2005-12-31'
                        EACH INTERVAL '1' MONTH);
```

A wide range of queries can make use of this join index, especially when there exists a foreign key-to-primary key relationship between the fact table and the dimension tables that enables the join index to be used broadly to cover queries on a subset of dimension tables. See [Query Coverage by Join Indexes](#), [Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#), and [Rules for Whether Join Indexes With Extra Tables Cover Queries](#). For information about broad join indexes, see *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

A significant performance gain is achieved with this join index when it is used to answer the queries that have equality or range constraints on the *sale_date* column.

Optimizations for row-partitioned base tables, such as row partition elimination, are applied to a join index in the same way as they are applied to a base table.

For example, the execution of the following query only needs to access 12 out of 192 partitions, saving up to 93.75 percent on disk reads, with proportional elapsed time reductions compared with a corresponding nonpartitioned join index, which must perform a full-table scan for the same query:

```

SELECT prod_category, SUM(amount) AS daily_sales
FROM calendar AS c, product AS p, sales AS s
WHERE c.dayofmth = s.sale_date
AND   p.prod_id = s.prod_id
AND   sale_date BETWEEN DATE '2005-01-01' AND DATE '2005-12-31'
GROUP BY prod_category;

```

An EXPLAIN of the SELECT statement includes SUM step to aggregate from 12 partitions of SALES_SUMMARY.

Scenario B

With the same schema used in Scenario A, you create a single-level row-partitioned PI single-table join index on the fact table that includes the foreign key of one or more dimension tables so it can be used to join to the corresponding dimension table set and roll up to higher aggregate levels. The following is an example of a single-table join index:

```

CREATE JOIN INDEX sji AS
  SELECT sale_date, prod_id, SUM(amount) AS sales_amount
  FROM Sales
  GROUP BY sale_date, prod_id
PRIMARY INDEX(sale_date, prod_id)
PARTITION BY RANGE_N(sale_date BETWEEN DATE '1990-01-01'
                     AND      DATE '2005-12-31'
                     EACH INTERVAL '1' MONTH);

```

For the total sales amount at the month level, the Optimizer can join this join index to the *calendar* table and roll up:

```

SELECT yr, mth, SUM(amount)
FROM sales, Calendar
WHERE sale_date=dayofmth
AND sale_date BETWEEN DATE '2005-01-01' AND DATE '2005-12-31'
GROUP BY yr, mth;

```

An EXPLAIN of the SELECT statement includes a RETRIEVE step from 12 partitions of the join index, sji. Row partition elimination is applied to sji, reducing the number of row partitions accessed to 12 out of 192 row partitions, making the join much faster than it would be with a nonpartitioned single-table join index, which requires a full-table scan. The database can apply other row-partition-related join optimizations, such as dynamic row partition elimination and rowkey-based merge join, to row-partitioned join indexes just as they are applied to row-partitioned base tables.

While a join index can greatly improve the query response time, you should consider the overhead of maintaining it when inserts, deletions, and updates are made to the base table on which the index

is defined. A row-partitioned join index might experience significant improvement in its maintenance performance due to row-partition-related optimizations. For example, for the join indexes defined in scenarios A and B, the database can apply row partition elimination when there is an equality or range constraint on the row partitioning column in the data maintenance statement.

For example:

```
DELETE sales
      WHERE sale_date BETWEEN DATE '1999-01-01'
                        AND    DATE '1999-12-31';
```

An EXPLAIN of the DELETE statement includes steps for:

- DELETE from 12 partitions of SALES_SUMMARY with a condition of ("(SALES_SUMMARY.sale_date ≥ DATE '1999-01-01') AND (SALES_SUMMARY.sale_date ≤ DATE '1999-12-31')")
- DELETE from 12 partitions of SJI with a condition of ("(SJI.sale_date ≥ DATE '1999-01-01') AND (SJI.sale_date ≤ DATE '1999-12-31')")

Partitioning on the DATE column in the join index helps insert performance if the transaction data is inserted into *sales* according to a time sequence. Note that Teradata MultiLoad and Teradata FastLoad are not supported for base tables with join indexes, so other batch loading methods, such as Teradata Parallel Data Pump or Teradata FastLoad into a staging table followed by either an INSERT ... SELECT or MERGE batch request with error logging must be used to load data 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 with the nonpartitioned join index case where the inserted rows scatter among all the data blocks.

You cannot drop a join index to enable Teradata MultiLoad or Teradata FastLoad batch loads until any requests that access that index complete processing. Requests place READ locks on any join indexes they access, and the database defers processing of any DROP JOIN INDEX requests against locked indexes until their READ locks have all been released.

The maintenance of a row-partitioned join index may be less efficient than an otherwise identical nonpartitioned join index. For example, if a row-partitioned join index has a primary index that does not include the partitioning column set, and a DELETE or UPDATE statement specifies a constraint on the primary index, the maintenance of the row-partitioned PI join index must probe all partitions within the AMP as opposed to the fast primary index access for a nonpartitioned join index.

Choosing Between a Row-Partitioned Join Index and a Value-Ordered NUSI For Covering Range Queries

Row-partitioned join indexes and value-ordered NUSIs are designed to optimize the performance of range queries. These guidelines apply to choosing between a value-ordered NUSI and a single-level partitioned join index only. Value-ordered NUSIs are not an alternative to multilevel partitioned join indexes.

Because you cannot define both a partitioning expression and a value-ordered NUSI on the same column set for a join index, nor can you define partitioning for a row-compressed join index, you must determine which is more likely to enhance the performance of a given query workload.

You should consider the following guidelines when determining whether to design a join index for a particular workload using a row-partitioned join index or using a nonpartitioned join index and adding a value-ordered NUSI to create a value-ordered access path to the rows:

- In general, if a partitioned join index suits your purposes, you should use a partitioned join index rather than using a value-ordered NUSI and a nonpartitioned join index.
- If row compression is defined on the join index, then you cannot define partitioning for it.

For this scenario, a value-ordered NUSI is your only join index design choice possibility for optimizing range queries.

- Each time an underlying base table for a join index is updated, the join index, too, must be updated.

If a value-ordered NUSI is defined on a join index, then it, too, must be updated each time the base table and join index rows are updated.

This additional maintenance is unnecessary when you define the join index with a row-partitioned PI, and row partition elimination makes updating a row-partitioned join index even faster than the equivalent update operation with a nonpartitioned join index. In this context, update includes the delete and update operations performed by DELETE, MERGE, and UPDATE SQL requests, but not insert operations.

- A PPI provides direct access to join index rows, while a value-ordered NUSI does not.

Using a NUSI to access rows is always an indirect operation, accessing the NUSI subtable before using the join index to access a row set. However, if the NUSI covers the query, there is no need to access the rows in the join index subtable.

A row-partitioned PI also provides a means for attaining better join and aggregation strategies on the primary index of the join index.

- If the primary index is specified as a query condition, a row-partitioned PI offers the additional benefit of enhanced direct join index row access using row partition elimination rather than the all-AMPs access approach required by NUSIs.

However, the comparative row access time depends on the selectivity of a particular value-ordered NUSI, the join index row size, and whether a value-ordered NUSI covers a particular query or not.

- If a join index column has more than 65,535 unique values, and the query workload you are designing the index for probes for a specific value, then a value-ordered NUSI is likely to be more selective than a join index partitioned on that column.

Column-Partitioned Join Indexes

The principal application of column-partitioned join indexes is to provide an alternative physical database design of a primary base table that is not column-partitioned. This enables you to maintain the identical data, but organized very differently to support different query workloads. By having these very different

organizations of the same data, you can present different types of queries to a table, one set that performs better on a table with a standard row structure, and another set that performs better on a column-partitioned physical table. For this second case, the Parser can substitute a column-partitioned join index for the table with a standard row structure, and achieve better performance than could be realized with the row-oriented storage of the primary data table.

Of course, you could also create the primary data table using column partitioning and a supporting join index on the same columns, but using a row-oriented physical design. The two methods are equally applicable, with neither having any particular advantage over the other.

Rules and Restrictions for Column-Partitioned Join Indexes

Column-partitioned join indexes have the following rules and restrictions.

- You can only specify column-partitioning, and optionally, column-partitioning mixed with row-partitioning, for a join index if the following assertions are all true.
 - The join index is defined on a single table.
It cannot be a multitable index.
 - The join index is a non-aggregate index.
 - The join index is not row-compressed.
- If you specify the COLUMN option in a PARTITION BY clause, the join index is by definition a column-partitioned join index.
- If specify NO PRIMARY INDEX or PRIMARY AMP INDEX, you must also specify a PARTITION BY clause and the partitioning must specify a COLUMN partitioning level.
- When you do not specify a column grouping for a COLUMN clause, Vantage defines a separate column partition for each column and column group specified for a CREATE JOIN INDEX request.
- You can only specify column or constraint grouping that is done in the select list if you also specify COLUMN partitioning in the PARTITION BY clause.

Note:

You cannot group columns or constraints in the select list of join indexes that are not column-partitioned.

- You cannot specify column or constraint grouping in both the select list and in the COLUMN option of the PARTITION BY clause of the same CREATE JOIN INDEX request.
- You cannot specify a column more than once within a column partition.
Otherwise, the system returns an error to the requestor.
- You cannot specify a column to be in more than one column partition.
Otherwise, the system returns an error to the requestor.
- The number of defined column partitions for a column partitioning level is the number of user-specified column partitions plus two column partitions reserved for internal use.

Vantage uses one of the reserved partitions as a sparse bit map to indicate deleted rows and the other is reserved for future use. There is always at least one column partition number that is not assigned to a column partition.

- The number of defined partitions for a row partitioning level is the number of user-defined row partitions specified by the PARTITION BY clause.

If you do not specify a RANGE_N or CASE_N function in the PARTITION BY clause, Vantage uses 65,535 row partitions for the join index.

- For a row partitioning level, the maximum partition number is the same as the maximum number of partitions for that level.
- For a column partitioning level, the maximum partition number is one more than the maximum number of column partitions for that level.

This ensures that there is at least one unused column partition number that is always available for altering a column partition.

- For each row partitioning level that does not specify the ADD option in level order, the maximum number of partitions for the row partitioning level is, if using the number of row partitions defined as the maximum for this and any lower row partitioning level without an ADD clause, the partitioning would be 2-byte partitioning, and is the largest value that does not cause the partitioning to be 8-byte partitioning.

Otherwise, the maximum number of partitions for this level is the largest value that does not exceed 9,223,372,036,854,775,807. If there is no such largest value, the system returns an error to the requestor.

- You can specify ADD 0 for a partitioning level to specify explicitly that the maximum number of partitions for this level is the same as the number of defined partitions.

This can be useful for a column partitioning level if you want to override the default of ADD 10 so that other levels can have more partitions.

This can also be useful for a row partitioning level if you want a lower level that does not specify the ADD clause to have any excess partitions.

- The maximum number of partitions for a row partitioning level must be at least two.

Otherwise, the system returns an error to the requestor.

This error occurs when only one partition is defined for a row partitioning level with an ADD 0 or with no ADD option and the maximum is not increased to at least two.

- Specifying a maximum number of partitions for a partitioning level that is larger than the number of defined partitions for that level can enable the number of defined partitions for that level to be increased using an ALTER TABLE request.

Note:

This does not mean that you can increase the maximum number of partitions for a level using an ALTER TABLE request.

- If you define a partitioning expression using only a single CASE_N function, the number of partitions defined must be less than or equal to 2,147,483,647 because the CASE_N function has INTEGER data type. (otherwise, existing error 5715 occurs).

Otherwise, the system returns an error to the requestor.

- If you define the partitioning expression for single-level partitioning using only a single RANGE_N function with INTEGER data type, the total number of partitions defined must be less than or equal to 2,147,483,647.

Otherwise, the system returns an error to the requestor.

For 2-byte single-level partitioning, the RANGE_N function cannot define more than 65,533 range partitions because by definition, more range partitions would require 8-byte partitioning.

The total number of partitions you can define can be as many as 65,535 if you define both a NO RANGE and an UNKNOWN partition.

- If you define a partitioning expression using only a single RANGE_N function with BIGINT data type, the number of ranges defined must be less than or equal to 9,223,372,036,854,775,805.

Otherwise, the system returns an error to the requestor.

For single-level partitioning, the total number of partitions you can define can be as many as 9,223,372,036,854,775,807 if you define both a NO RANGE and an UNKNOWN partition.

- The number of partitions defined for a row partitioning level is the number of row partitions specified by the RANGE_N or CASE_N function, or 65,535 if you specify neither a RANGE_N or a CASE_N function.
- You can specify an ADD option for a partitioning level. The BIGINT number following the ADD keyword plus the number of partitions defined for the level is the maximum number of defined partitions allowed for that level.

If this maximum exceeds 9,223,372,036,854,775,807, the system returns an error to the requestor.

For single-level partitioning, the maximum combined partition number is the same as the maximum partition number for the partitioning level.

The ADD option allows the number of partitions to be increased for that level up to this maximum using ALTER TABLE requests.

- If you do not specify an ADD option for a column partitioning level, and the level is the only partitioning level, the maximum number of partitions, including the two for internal use, is 65,534.
- If you do not specify an ADD option for a column partitioning level and the following list of items is also true, the partitioning is stored using 2 bytes in the row header and the maximum number of partitions for the column partitioning level is the number of column partitions defined plus 10. In other words, the default for this case is ADD 10.
 - There is row partitioning
 - At least one of the row partitioning levels does not specify the ADD option

The maximum number of partitions for the column partitioning level is the largest value that does not cause the partitioning to be 8-byte partitioning; otherwise, the maximum number of partitions for the column partitioning level is the largest value that does not exceed 9,223,372,036, 854,775,807.

For single-level partitioning, the maximum combined partition number is the same as the maximum partition number for this partitioning level.

- The number of combined partitions for a join index is the product of the number of partitions defined by each partitioning level.
- The maximum number of combined partitions for a join index is the product of the maximum number of partitions defined for each partitioning level.

For single-level partitioning, the maximum number of combined partitions is the same as the maximum number of partitions for the partitioning level.

If there is column partitioning, the maximum number of combined partitions is smaller than the maximum combined partition number. Otherwise, it is the same.

- If the maximum combined partition number is greater than 65,535, Vantage stores the partition number in an 8-byte field in the row header. This is referred to as 8-byte partitioning.

If the maximum combined partition number is less than or equal to 65,535, Vantage stores the partitioning in a 2-byte field in the row header. This is referred to as 2-byte partitioning.

Single-level column partitioning with no ADD option is stored in a 2-byte field.

- The maximum combined partition number for a join index is the product of the maximum partition numbers for each partitioning level.

The maximum cannot exceed 9,223,372,036, 854,775,807.

For single-level partitioning, the maximum number of combined partitions is the maximum number of partitions for this partitioning level.

- For 2-byte partitioning, the maximum number of partitions for the first level is the largest value that does not cause the maximum combined partition number to exceed 65,535. This is repeated for each of the other levels, if any, from the second level to the last level.
- For 2-byte partitioning, the maximum number of partitioning levels is 15.
- For 8-byte partitioning, the maximum number of partitions for the first level is the largest value that does not cause the maximum combined partition number to exceed 9,223,372,036,854,775,807. This is repeated for each of the other levels from the second level to the last level.
- For 8-byte partitioning, the maximum number of partitioning levels is 62.

If the maximum column partition number is more than two at one or more levels, the number of partitioning levels may be further limited because of the limit placed by the rule that the product of the maximum combined partition numbers at each level cannot exceed 9,223,372,036,854,775,807.

- Vantage derives a partitioning constraint from the partitioning level definitions for a column-partitioned join index.

The text for this partitioning constraint cannot exceed 16,000 characters; otherwise, the system returns an error to the requestor.

The following definitions apply to this rule.

| Variable | Definition |
|-----------|---|
| <i>cs</i> | Number of user-specified column partitions. |
| <i>cm</i> | Maximum partition number for the column partitioning level. |

Vantage initially assigns the user-specified column partitions numbers starting at 1 in increments of 1 up to *cs*.

Vantage assigns an internal partition a partition number of *cm*-1 and the delete column internal partition is assigned a partition number of *cm*.

At first, no column partitions are assigned to column partition numbers *cs*+1 to *cm*-2. Initially, there is at least one unused column partition number because *cm*-*cs*-2 is greater than 0.

As you drop or alter partitions, there can be gaps in the numbering. As you add or alter column partitions, unused column partition numbers between 1 and *cm*-2 can be assigned to the newly added or altered column partitions as long as one column partition number remains unused. This is necessary because at least one column partition number must be available for use by ALTER TABLE to alter a column partition.

Vantage uses the column partition number to compute the combined partition number for a column partition value of a join index row. Apart from that, there is no significance to the column partition number assigned to a column partition.

- If a join index is partitioned, its fallback rows are partitioned identically to its primary data rows.
If the primary data rows have a primary index, the fallback data rows have the same primary index.
If the primary data rows do not have a primary index, the fallback data rows also do not have a primary index.

The following table summarizes to which partitioning level any excess partitions are added.

| IF partitioning is ... | AND ... | THEN Vantage adds as many left over combined partitions as possible to ... |
|------------------------|---|---|
| single-level | | the only row or column partitioning level. If you specify an ADD clause, Vantage overrides it. |
| multilevel | all the row partitioning levels have an ADD clause, but there is a column partitioning level without an ADD clause, | the column partitioning level. This does not need to be added at the first partitioning level. |
| | a column partitioning level and at least one of the row partitioning levels does not have an ADD clause, including the case where none of the | the first row partitioning level without an ADD clause after using a default of ADD 10 for the column partitioning level. |

| IF partitioning is ... | AND ... | THEN Vantage adds as many left over combined partitions as possible to ... |
|------------------------|---|--|
| | row partitioning levels has an ADD clause specified, | This is repeated for each of the other row partitioning levels without an ADD clause, if any, in level order. |
| | a column partitioning level has an ADD clause and at least one of the row partitioning levels does not have an ADD clause | the first row partitioning level without an ADD clause. This is repeated for each of the other row partitioning levels without an ADD clause, if any, in level order. |
| | there is no column partitioning level and at least one of the row partitioning levels does not have an ADD clause. This includes the case where none of the row partitioning levels has an ADD clause specified. | the first row partitioning level without an ADD clause. |
| | all of the partitioning levels have an ADD clause or after applying left over combined partitions. | the first row or column partitioning level and overrides the ADD clause for the first partitioning level if one is specified. Vantage repeats this for each of the other levels, if any, from the second level to the last level. |

You can define a large variety of partitioning expressions and column groupings for column partitioning with a large range in the number of combined partitions. You must consider the usefulness of defining a particular partitioning and its impact on performance and storage.

See [Performance Issues for Column-Partitioned Tables](#) for performance considerations.

Column grouping in the COLUMN option of a PARTITION BY clause enables more flexibility in specifying which columns belong to which partitions while still being able to specify the display order in the column list.

Column grouping in the select list enables a simpler, but less flexible, specification of column groupings.

- A column partition value consists of the values of the columns in the column partition for a specific join index physical row.
- If you specify a column grouping clause for a COLUMN specification of a PARTITION BY clause, the columns specified for the grouping columns must have been specified in the select list for the join index.
- If you specify a column grouping clause for a COLUMN specification of a PARTITION BY clause, Vantage defines column partitions as specified by that column grouping clause.

- If you specify a column or constraint grouping or both with COLUMN format in the column list, the grouping defines a column partition and Vantage stores one or more column partition values in a physical container using COLUMN format.

If you specify a column or constraint grouping or both with ROW format in the column list, the grouping defines a column partition that stores only one column partition value in a physical subrow using ROW format.

If you specify neither a COLUMN format nor a ROW format for a column or constraint grouping, the grouping defines a column partition and Vantage determines whether to use a COLUMN or ROW format for the column partition.

- You can specify either AUTO COMPRESS or NO AUTO COMPRESS as the default for column partitions. If you do not explicitly specify either, AUTO COMPRESS is the default..
- If you specify a column or constraint grouping (or both) with AUTO COMPRESS or NO AUTO COMPRESS in the column list, Vantage applies the appropriate autocompression for the specified physical rows and applies any user-specified compression to the column partition.

Vantage also applies row header compression for column partitions with COLUMN format.

If you specify neither AUTO COMPRESS nor NO AUTO COMPRESS, Vantage uses the default that you specify explicitly for COLUMN in the PARTITION BY clause.

If you do not define a default for COLUMN, Vantage uses AUTO COMPRESS.

- If you specify neither PRIMARY INDEX (*column_list*) nor NO PRIMARY INDEX explicitly in your CREATE JOIN INDEX request, but do specify a PARTITION BY clause, Vantage creates the join index without a primary index regardless of the setting of the DBS Control flag PrimaryIndexDefault. For details and exceptions, see *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - Database Utilities*, B035-1102.
- If a COLUMN option that is a partitioning level of a PARTITION BY clause neither specifies column grouping nor specifies a column or constraint definition delimited by parentheses (which defines a group), Vantage treats the specified individual column as a single-element group, defining a single-column partition with AUTO COMPRESS and with a system-determined COLUMN or ROW format.
- To specify either a COLUMN or ROW column partition format or to specify NO AUTO COMPRESS for a single-column column partition, you must delimit the column or constraint definition within parentheses to denote it as a single-column group.

You can group one or more column or constraint definitions into a column partition by delimiting the definitions with parentheses.

You can achieve the same result by specifying the same options and grouping in a COLUMN option in the PARTITION BY clause.

- Column partitions have either a COLUMN format or a ROW format. Formats cannot mixed within the same column partition.

However, different column partitions of a column-partitioned join index *can* have different formats. For example, the column partitions of a column-partitioned join index can have all COLUMN format (and

be stored in containers), all ROW format (and be stored in subrows), or some partitions in COLUMN format and others in ROW format.

- You can specify a different column grouping for a target table than the source table has in the column list of a CREATE TABLE ... AS request. The rules listed here also apply.

COLUMN Partitioning in the PARTITION BY Clause of a Join Index

You can create a single-table, nonaggregate, noncompressed join index with column partitioning and, optionally, row partitioning. If you specify COLUMN partitioning in the PARTITION BY clause of a join index, the following rules and restrictions apply.

- You can specify a PRIMARY INDEX, PRIMARY AMP INDEX, or NO PRIMARY INDEX clause.
- You cannot specify a UNIQUE PRIMARY INDEX clause.
- If you do not specify a PRIMARY INDEX, PRIMARY AMP INDEX, or a NO PRIMARY INDEX clause, the default is NO PRIMARY INDEX regardless of the setting of the DBS Control field PrimaryIndexDefault.
- You cannot specify an ORDER BY clause.
- You can define column-partitioning for a sparse join index.
- You must specify an alias for the system-derived column ROWID in the select list of the CREATE JOIN INDEX request used to create a column-partitioned join index.

You can then use the alias you specify for ROWID in a column grouping clause specified in a COLUMN clause in the PARTITION BY clause.

- Any expression in the select list of the join index definition that is not a column name must be specified with an alias.
- None of the columns specified in the select list for the join index definition can have a server character set of CHARACTER SET KANJI1.

Instead, you should change the server character set to CHARACTER SET UNICODE.

Specifying a Column Grouping for the COLUMN Option

Grouping columns in a COLUMN specification of the PARTITION BY clause enables flexibility in specifying the partitions to which columns belong.

The following rules apply when you specify a column grouping for the COLUMN option.

- When you specify a column grouping for a COLUMN clause, you can only specify the name of a column that is defined in the select list for the join index.

If you attempt to specify a column by something other than its name, the system returns an error to the requestor.

- Vantage defines a column partition for each non-group and group column partition specified in the select list of the join index definition.
- If you specify COLUMN for a column partition, Vantage stores one or more column-partition values in a physical container.

- If you specify ROW for a column partition, Vantage stores only one column partition value in a physical subrow.

A subrow is just a standard Teradata row format, and the term subrow is used to emphasize that it is part of a column partitioning.

- If you specify neither COLUMN nor ROW for a column partition, Vantage determines whether to use COLUMN or ROW format for the column partition.

A column partition value consists of the values of the columns in the column partition for a specific join index row.

- The following table provides the rules for using the ALL BUT option in the PARTITION BY clause of a join index definition.

| IF ... | THEN Vantage ... |
|------------------------|--|
| you specify ALL BUT | <ul style="list-style-type: none"> defines a single-column partition with autocompression. defines a system-determined COLUMN or ROW format for any column that is not specified in select list. |
| do not specify ALL BUT | groups any columns that you do not specify in the select list into one column partition with autocompression and a system-determined COLUMN or ROW format. |

- The following table provides the rules for using the NO AUTO COMPRESS option in the PARTITION BY clause of a join index definition.

| IF you ... | THEN Vantage ... |
|--|---|
| specify NO AUTO COMPRESS for a column partition | does not apply autocompression for physical rows. |
| do not specify NO AUTO COMPRESS for a column partition | applies autocompression for physical rows. |

- A column partition either has COLUMN format or ROW format. It cannot have a mix of both formats. Different column partitions of a column-partitioned join index can have different formats. In other words, the column partitions of a column-partitioned join index can have all COLUMN format, all ROW format, or some COLUMN format and others ROW format.
- If you do not specify an explicit COLUMN or ROW format for a partition, Vantage makes the determination for you.

When the COLUMN or ROW format is system-determined, Vantage bases its choice of format on the size of a column partition value for the column partition and other factors such as whether a column-partition value for the column partition has fixed or variable length and whether the column partition is a single-column or multicolumn partition.

In general, Vantage assigns a COLUMN format to a narrow column partition and assigns a ROW format to a wide column partition.

You can submit a `HELP COLUMN` request or select the `ColumnPartitionFormat` column from the `DBC.ColumnsV(X)` view to determine which format Vantage elected to use for a column partition.

Note that you can explicitly specify the format if you want to use a specific format for a column partition.

Options for Column-Partitioned Join Indexes

You can define the following options for column-partitioned join indexes.

- Column partitioning
- Fallback

If a table is column-partitioned, its fallback rows are also column-partitioned using the same partitioning as its primary data rows.

- Secondary indexes
- BLOB, CLOB, ARRAY, VARRAY, and Geospatial columns.

The validity of UDTs as index elements does *not* include a NUSI defined on a UDT that contains one or more LOB elements.

There is a limit of approximately 64K rows per row hash value for LOB columns. Because there is normally only one row hash value per AMP for column-partitioned join indexes, there is also a limit of approximately 64K rows per AMP for column-partitioned join indexes that contain columns typed as BLOBs or CLOBs.

BLOCKCOMPRESSION

Use this option to set the block compression state of a join index.

The following table lists the available options for BLOCKCOMPRESSION.

| Option | Description |
|----------|--|
| AUTOTEMP | The compressed state of the data in the join 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 definition of whether the join index uses the MANUAL, AUTOTEMP or NEVER compression options is determined by the DBS Control flag <code>DefaultTableMode</code> . For more information, see <i>Teradata Vantage™ - Database Utilities</i> , B035-1102. |
| MANUAL | The compressed state of the data in the join index does not change unless you take specific action to do so using Ferret commands. |
| NEVER | The join index is not compressed even if the DBS Control block compression settings indicate otherwise. This means that Vantage rejects Ferret commands to manually compress the join index, but Ferret commands to decompress the index are valid. |

See *Teradata Vantage™ - Database Utilities*, B035-1102 for information about how to use the Ferret utility.

Similarities and Differences Between Join Indexes and Base Tables

In many respects, a join index is similar to a base table. For example, you can create NUSIs on a join index and you should collect statistics on appropriate columns and indexes of both to ensure that the Optimizer has accurate statistical summaries of their demographics.

Note:

You cannot collect statistics on a UDT column. This includes UDT columns that are components of an index.

You can also perform DROP, HELP and SHOW statements on join indexes. However, you *cannot* directly query or update a join index. You *can* do this by creating a view that has a similar definition to that of the join index, which can then be queried like any other view. Such a view provides a *logical* view of the data, and you can maintain it for as long as you need it. A join index, in contrast, is a *physical* database object that you create or delete for performance reasons. It does not affect the logical view of the data in any way.

For example, the following query is not valid because `ord_cust_idx` is a join index, not a base table.

```
SELECT o_status, o_date, o_comment
FROM ord_cust_idx;
```

Because join indexes are not part of the logical definition of a database, you can use them as summary and prejoin tables, both of which would otherwise violate the rules of normalization. The update anomalies presented by denormalized structures like summary and prejoin tables are avoided because Vantage implicitly performs all necessary updates to join indexes to ensure that no information is ever lost and that semantic integrity is always enforced (see [Join Index Updates Are Maintained by Vantage](#) for details).

Join Index Updates Are Maintained by Vantage

Most values of a join index are automatically maintained by Vantage when update operations (DELETE, INSERT, MERGE, UPDATE) are performed on columns in their underlying base tables that are also defined for the join index. Additional steps are included in the execution plan to regenerate the affected portion of the stored join result.

The exception to this update rule is the values for any join index partitioning expression that specifies a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP built-in function. In this case, you must submit an appropriate ALTER TABLE TO CURRENT request to reconcile the values stored for the date or timestamp for the partitioning expression. Note that you *cannot* use an ALTER TABLE TO CURRENT request to reconcile a CURRENT_TIME function that is specified within a partitioning expression.

Similarly, you *cannot* reconcile the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP values specified in the WHERE clause of a join index. For more information about updatable date and timestamp partitioning functions, see [Join Indexes, Expressions, and Built-In Functions](#) and [ALTER TABLE TO CURRENT](#).

Simple Join Indexes

Define a simple join index as a join index that does not specify aggregate operations.

The primary function of a simple join index is to provide the Optimizer with a high-performing, cost-effective means for satisfying any query that specifies a frequently performed join operation. The simple join index permits you to define a permanent prejoin table without violating schema normalization.

You can also define a simple join index on a single table. This enables you to hash a subset of the columns of a large base table on a foreign key that hashes rows to the same AMP as another large table to which it is frequently joined. This is most commonly done by projecting a proper subset of the columns from the base table into a single-table join index, which is often referred to as *vertical partitioning*. Note that this is not related to the range partitioning performed with a partitioned join index. But if your design requires it, you can also project *all* of the columns from the base table into a second physical table that differs from its underlying base table only in the way its rows are hashed to the AMPs.

In some situations, this is more high-performing than building a multitable join index on the same columns due to less internal update maintenance on the single table form of the index.

Only prototyping can determine which is the better design for a given set of tables, applications, and hardware configuration.

The following describes a general procedure for defining a single-table join index:

1. Determine whether the partitioning for the join index should be a nonpartitioned, row-partitioned, column-partitioned, or column-partitioned with a mix of row partitioning.

If you decide to define the index using partitioning, determine whether that partitioning should be a single-level or multilevel.
2. Define a join index on the frequently joined columns of the table to be distributed on a new partitioning.
3. Define a *column_1_name* for each *column_name* in the primary base table to be included in the single-table join index.

Include the keyword ROWID as a value in the *column_name* list to enable the Optimizer to join a partial covering index to its base table to access any non-covered columns. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement.

Even though you do not explicitly specify this join when you write your query, it counts against the 64 table restriction on joins.

4. For a primary-indexed join index, define the primary index on the column set on which the join is made. This is typically a foreign key from the table to which the index table is to be joined.
5. If performance suggests it, use CREATE INDEX to create one or more NUSIs on the join index.
6. If performance suggests it, collect the appropriate statistics on the join columns, the indexes, and the ORDER BY column. In most applications, you should collect statistics on the base table columns on which the index is defined rather than on the index columns themselves. See [Collecting Statistics on a Single-Table Join Index](#) and [Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns](#) for clarification.

Aggregate Join Indexes

An aggregate join index is a join index that specifies SUM, COUNT, MIN, or MAX aggregate operations. No other aggregate functions are permitted in the definition of a join index.

The primary function of an aggregate join index is to provide the Optimizer with a high-performing, cost-effective means for satisfying any query that specifies a frequently made aggregation operation on one or more columns. The aggregate join index permits you to define a summary table without violating schema normalization.

Optimizer Rules for Using Aggregate Join Indexes in a Query Plan

The Optimizer analyzes all queries for the use of an aggregate join index. In determining if an aggregate join index can be used to process a query, the Optimizer applies the following rules:

- An aggregate join index cannot be used to substitute for the base tables of a non-aggregate query.
- Like a simple join index, an aggregate join index qualifies for inclusion in the plan if it specifies either the same joins or a subset of the joins of the query.
- If an aggregate join index specifies a subset of joins of the query, then all the joined fields of the remaining join conditions on the tables of the aggregate join index must be specified in the GROUP BY clause *and* in the select list of the aggregate join index definition.
- Even if an aggregate join index specifies the same joins as the query, it does not qualify if its GROUP BY columns are neither the same nor a superset of the GROUP BY columns for the query.
- An aggregate join index must contain all the columns needed by the query for the tables that it replaces. When trying to match expressions from a query to the select list for a join index, the Optimizer applies the following rules:
 - Only the SUM, COUNT, MIN, and MAX aggregate functions are valid for an aggregate join index.
 - Addition and multiplication are commutative.
 - Vantage implicitly adds a COUNT(*) or COUNT(*non_null_expression*) term to the definition of an aggregate join index if neither term is not explicitly defined.
 - Vantage uses COUNT(*) and COUNT(*non-null_expression*) interchangeably.
 - An aggregate expression must be assigned an alias.
 - You cannot base the primary index of an aggregate join index on the alias of an aggregate expression.
 - You cannot specify a DISTINCT operator to qualify the operand of an aggregate expression. For example, you cannot specify MIN(DISTINCT *column_expression*) in the definition of an aggregate join index.
 - A numeric expression from the join index that is converted to FLOAT can be used to match any expression of the same type defined with or without the conversion from the query.
 - An AVERAGE aggregate expression in a query is converted to a SUM/COUNT expression when an aggregate join index substitutes for it.

- A SUM(0) expression in a query is converted to a constant having value 0.
- A SUM(*constant*) expression in a query matches with *constant* * COUNT(*) in a aggregate join index definition.
- The SUM or COUNT function from SUM(CASE *expression*) or COUNT(CASE *expression*) can be pushed to all the resulting expressions of the CASE expression to find a match. For example, the following expression can be converted to the expression following it to find a match.

```
SUM(CASE WHEN x1=1
      THEN 1
      ELSE 0)

CASE WHEN x1=1
      THEN SUM(1)
      ELSE SUM(0)
```

- Conditions are converted according to the following conversion table.

| THIS condition ... | IS converted to this equivalent expression ... |
|--|--|
| <i>expression</i> ≥ 'YYYY0101' | 'EXTRACT(YEAR FROM <i>expression</i>) ≥ YYYY' |
| <i>expression</i> > 'YYYY1231' | 'EXTRACT(YEAR FROM <i>expression</i>) > YYYY' |
| <i>expression</i> < 'YYYY0101' | 'EXTRACT(YEAR FROM <i>expression</i>) < YYYY' |
| <i>expression</i> ≤ 'YYYY1231' | 'EXTRACT(YEAR FROM <i>expression</i>) ≤ YYYY' |
| <i>expression</i> ≥ 'YYYYMM01' | 'EXTRACT(YEAR FROM <i>expression</i>) > YYYY' or '(EXTRACT(YEAR FROM <i>expression</i>) = YYYY AND EXTRACT(MONTH FROM <i>expression</i>) ≥ MM)' |
| <i>expression</i> > 'YYYYMMLD' where LD is the last day of the month specified by MM | 'EXTRACT(YEAR FROM <i>expression</i>) > YYYY' or '(EXTRACT(YEAR FROM <i>expression</i>) = YYYY AND EXTRACT(MONTH FROM <i>expression</i>) > MM)' |
| <i>expression</i> ≤ 'YYYYMMLD' where LD is the last day of the month specified by MM | 'EXTRACT(YEAR FROM <i>expression</i>) < YYYY' or '(EXTRACT(YEAR FROM <i>expression</i>) = YYYY AND EXTRACT(MONTH FROM <i>expression</i>) ≤ MM)' |
| <i>expression</i> < 'YYYYMM01' | 'EXTRACT(YEAR FROM <i>expression</i>) < YYYY' or '(EXTRACT(YEAR FROM <i>expression</i>) = YYYY AND |

| THIS condition ... | IS converted to this equivalent expression ... |
|--------------------|--|
| | EXTRACT(MONTH FROM <i>expression</i>) < MM' |

Sparse Join Indexes

You can create join indexes that restrict the number of rows in the index to those that are accessed when, for example, a frequently run request references only a small, well known subset of the rows of a large base table. By using a constant expression in the WHERE clause to filter the rows included in the join index, you can create what is known as a sparse join index.

For example, the following CREATE JOIN INDEX request creates an aggregate join index containing only the *sales* records from 2007. The filtering is done by the constant predicate expression EXTRACT(year, sales_date)=2007 in the WHERE clause.

```
CREATE JOIN INDEX j1 AS
SELECT storeid, deptid, SUM(sales_dollars)
FROM sales
WHERE EXTRACT(year, sales_date)=2007
GROUP BY storeid, deptid;
```

When a request is made against an indexed base table, the Optimizer determines if accessing *j1* provides the correct answer and is more efficient than accessing the base tables. The Optimizer then selects this sparse join index only for queries that restricted themselves to data from the year 2007.

For example, a query might require data from departments 42, 43, and 44, but only for the year 2007. Because the join index *j1* contains all of the data for year 2007, it might be used to satisfy the following query.

```
SELECT storeid, deptid, SUM(sales_dollars)
FROM sales
WHERE EXTRACT(year FROM sales_date) = 2007
AND deptid IN(42, 43, 44)
GROUP BY storeid, deptid;
```

Be aware of the following complication: When a base table has a CHECK constraint on a character column, values inserted into or updated for that column are tested based on the collation for the current session. This means that character values that pass a CHECK constraint for one session collation might fail the same check when a different session collation is in effect. This can have a particularly important effect on the values contained in a sparse join index, which, in turn, can affect the result of a request that the Optimizer uses that index to cover. Furthermore, such a scenario can also affect the result of the same request submitted when the sparse join index exists versus the same request submitted for the identical data when that sparse join index does not exist.

See *Teradata Vantage™ - Database Design*, B035-1094 for applications of sparse join indexes that take advantage of a partitioned primary index on the base table they support.

Join Indexes, Expressions, and Built-In Functions

When you create a join index that specifies a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP built-in function in its partitioning expression, the default is for Vantage to resolve the function at the time the join index is created and then to store the result as part of the index definition rather than evaluating it dynamically at the time the Optimizer would use the index to build a query plan. Note that you *cannot* reconcile a CURRENT_TIME or TIME function that is specified as part of the partitioning expression for a join index using ALTER TABLE TO CURRENT requests.

This need not be the case, however, because you can reconcile the value of any DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function specified as part of a partitioning expression using the ALTER TABLE TO CURRENT statement (see [ALTER TABLE TO CURRENT](#)). A DateTime expression used in this way in a partitioning expression is referred to as an updatable date or updatable timestamp.

When you create a join index that specifies a built-in, or system, function in its WHERE clause, Vantage resolves the function at the time the join index is created and then stores the result as part of the index definition rather than evaluating it dynamically at the time the Optimizer would use the index to build a query plan. For more information about built-in functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

You can use ALTER TABLE TO CURRENT requests to reconcile DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions that are specified in the WHERE clause of a join index definition. The Optimizer use the join index if the query specifies an explicit value in its WHERE clause that matches the resolved value stored in the index definition.

The following built-in functions are relevant to this issue:

- ACCOUNT
- DATABASE
- PROFILE
- ROLE
- SESSION
- TIME
- USER

For example, suppose you decide to define a join index using the CURRENT_DATE built-in function on January 4, 2007 as follows:

```
CREATE JOIN INDEX curr_date AS
  SELECT *
  FROM orders
  WHERE order_date >= CURRENT_DATE;
```

On January 7, 2007, you perform the following SELECT request:

```
SELECT *
FROM orders
WHERE order_date >= CURRENT_DATE;
```

When you EXPLAIN this query, you find that the Optimizer does not use join index *curr_date* because the date stored in the index definition is the explicit value '2007-01-04', not the current system date '2007-01-07'.

On the other hand, if you were to perform the following SELECT request on January 7, 2007, or any other date, the Optimizer *does* use join index *curr_date* for the query plan because the statement explicitly specifies the same date that was stored with the join index definition when it was created:

```
SELECT *
FROM orders
WHERE order_date = DATE '2007-01-04';
```

You can define join indexes, including aggregate join indexes, on Period and UDT columns; however, you cannot collect statistics on UDT columns. You can also define join indexes that specify other non-aggregate, non-OLAP, and non-UDF expressions in the select list, as well as single-table conditions in the WHERE and ON clauses of the index definition. This includes allowing a join index to be defined using a method that is associated with a UDT column.

The following set of rules defines the restrictions and limitations of specifying expressions and columns with Period and UDT data types in join index definitions.

- Any expression that you specify in the select list of a join index definition must be defined on at least one column.

You cannot specify a constant expression in the select list.

For example, the following request is not valid because it specifies a constant CASE expression in its select list.

```
CREATE JOIN INDEX test AS
  SELECT a1, CASE WHEN 1=1
                THEN 1
                ELSE 0
            END AS c
FROM t1;
```

- An expression specified in the select list of a join index definition can reference a set of columns from a single table or from multiple tables.

Note that Vantage does not treat a WHERE clause condition that is composed of a multicolumn expression as a join condition.

If you are creating a multitable join index, the tables must be joined by an equality condition on a column from each table. Otherwise, the system returns an error to the requestor.

For example, the following request is not valid because only satisfiable conditions that have constant or inequality conditions ANDed to at least one equality join between columns from different tables of the same type are valid in the WHERE clause.

```
CREATE JOIN INDEX test AS
  SELECT *
  FROM t1,t2
  WHERE a1+b1=a2;
```

- An expression specified in the select list of a join index definition must be aliased.

For example, the following request is not valid because the expression BEGIN(d1) is not aliased.

```
CREATE JOIN INDEX test AS
  SELECT a1, BEGIN(d1)
  FROM t1;
```

- An expression defined in the select list of an aggregate join index definition must also be specified as part of the GROUP BY clause.

For example, the following request is not valid because it does not specify a GROUP BY clause that includes the aggregate expression SUM(c1) AS s.

```
CREATE JOIN INDEX test AS
  SELECT (a1+b1) AS a, SUM(c1) AS s
  FROM t1;
```

- You can define either the primary or secondary for a join index on a UDT column. For example, assume that column *t1.d1* has a UDT data type.

The following CREATE JOIN INDEX request is valid because its primary index is defined on column *t1.d1*.

```
CREATE JOIN INDEX test AS
  SELECT d1, a1
  FROM t1
  PRIMARY INDEX (d1);
```

- You can define a compressed join index that specifies a UDT column.

For example, assume that column *t1.d1* has a UDT data type. The following CREATE JOIN INDEX request is valid.

```
CREATE JOIN INDEX test AS
  SELECT (b1), (c1,d1)
  FROM t1;
```

For examples of how to specify UDT and Period data type columns, BEGIN expressions, and P_INTERSECT expressions correctly in the select list of a join index definition, see CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Outer Joins and Simple Join Indexes

Because join indexes generated from inner joins do not preserve unmatched rows, you should always consider using outer joins to define simple join indexes. This practice empowers the join index to satisfy queries with fewer join conditions than those used to generate the index.

You can define a join index using both left and right outer joins, but full outer joins are prohibited.

Be aware that when you define a join index using an outer join, you must reference all the columns of the outer table in the select list of the join index definition. If any of the outer table columns are not referenced in the select list for the join index definition, the system returns an error to the requestor.

See [Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#) and [Rules for Whether Join Indexes With Extra Tables Cover Queries](#) for some special considerations about how to define outer joins to maximize the coverage possibilities of a join index.

Collecting Statistics on Join Indexes

Single- and multitable join indexes are fundamentally different database objects in many ways, and they are used for different purposes.

As with hash indexes, column statistics for single-table join indexes are generally best collected directly on the base table columns rather than separately as is always required for multitable join indexes.

Note:

You cannot collect statistics on a UDT column. This includes UDT columns that are components of an index. The Optimizer uses dynamic AMP sampling information for equality predicates on UDT columns and default selectivity for other predicates on UDT indexes for costing. The dynamic AMP sampling provides limited statistics information about the index. For example, it cannot detect nulls or skew. If a UDT index access path does not show any improved performance, you should consider dropping the index to avoid the overhead involved in its storage and maintenance.

An important exception to this guideline is the case where a single-table join index is defined on a complex expression that is also frequently specified as a term in predicate expressions for queries made on the relevant base table column. In this context, a complex expression is defined as an expression that specifies something other than a simple column reference on the left hand side of a predicate.

Always consider using the SAMPLE options when you collect and recollect statistics on a join index. See [Reducing the Cost of Collecting Statistics by Sampling](#) for further information about these options and recommendations on how to use them.

The Optimizer can only use statistics collected on complex expressions that can be mapped completely to a single-table join index or hash index expression. Collecting statistics on the join index column for

those expressions enhances the ability of the Optimizer to estimate single-table cardinalities for a query that specifies the base table expressions in its predicate. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

The following table points to the various topics that explain the different recommendations for collecting statistics for single-table and multitable join indexes:

| FOR this type of join index ... | The following topics explain how best to collect statistics on the index ... |
|---------------------------------|--|
| single-table | <ul style="list-style-type: none"> • Collecting Statistics on a Single-Table Join Index. • Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns. • Guidelines for Collecting Statistics On Single-Table Join Index Columns. |
| multitable | Collecting Statistics on Multitable Join Indexes. |
| sparse | <ul style="list-style-type: none"> • Collecting Statistics on a Single-Table Join Index. • Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns. |

For general information about collecting statistics, see [COLLECT STATISTICS \(Optimizer Form\)](#).

Collecting Statistics on a Single-Table Join Index

As is true for hash indexes, you should collect the statistics on base table columns rather than on single-table join index columns for most applications.

An important exception to this guideline is the case where a single-table join index is defined on a complex expression that is also specified as a predicate expression on the relevant base table column. Collecting statistics on the single-table join index for those expressions enhances the ability of the Optimizer to estimate single-table cardinalities for a query that specifies the base table expression in its predicate. See [Collecting Statistics on Join Indexes](#) and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Note:

If the join index is sparse (see [Sparse Join Indexes](#)), you should collect statistics on the join index itself rather than on its underlying base table.

See [Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns](#) for further information.

You can collect and drop statistics on the primary index columns of a single-table join index using an INDEX clause to specify the column set.

For example, suppose a join index has the following definition.


```
CREATE JOIN INDEX OrdJIdx8 AS
  SELECT o_custkey, o_orderdate
  FROM Orders
PRIMARY INDEX (o_custkey, o_orderdate)
ORDER BY (o_orderdate);
```

Then you can collect statistics on the index columns as a single object as shown in the following example:

```
COLLECT STATISTICS ON OrdJIdx8 INDEX (o_custkey, o_orderdate);
```

Note that you can *only* use columns specified in a multicolumn PRIMARY INDEX clause definition if you specify the keyword INDEX in your COLLECT STATISTICS request.

A poorer choice would be to collect statistics using the column clause syntax. To do this, you must perform two separate statements:

```
COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_custkey);

COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_orderdate);
```

It is always better to collect the statistics for multicolumn indexes on the index itself rather than individually on its component columns because its selectivity is much better when you collect statistics on the index columns as a set. Note that you can collect statistics on multiple unindexed columns as well. See [Collecting Statistics on Multiple Columns](#).

For example, a query with a condition like WHERE x=1 AND y=2 is better optimized if statistics are collected on INDEX (x,y) than if they are collected individually on column x and column y.

The same restrictions hold for the DROP STATISTICS statement.

See [Comparison of Hash and Single-Table Join Indexes](#) for a list of the similarities between single-table join indexes and hash indexes.

Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns

The Optimizer substitutes base table statistics for single-table join index statistics when no demographics have been collected for its single-table indexes. Because of the way single-table join index columns are built, it is generally best *not* to collect statistics directly on the index columns and instead to collect them on the corresponding columns of the base table. This optimizes both system performance and disk storage by eliminating the need to collect the same data redundantly.

Note:

This recommendation does not apply to sparse join indexes. If the join index is sparse, you should collect statistics on the join index itself rather than on its underlying base table.

Also, this recommendation does not apply to the case where a single-table join index is defined using a complex expression in its select list that is frequently specified in predicates used in queries made against the mapped base table columns in the expression. See [Collecting Statistics on Join Indexes](#) and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

There are three reasons why you might need to collect statistics on single-table join index columns instead of their underlying base table columns.

- The single-table join index is sparse (see [Sparse Join Indexes](#)).
- If you decide not to collect the statistics for the relevant base table columns for some reason, then you should collect them directly on the corresponding single-table join index columns.
- If the primary index column set consists of more than one column, and there is no primary or secondary index on the base table that includes those columns, then you have two options:
 - The better option is to collect multiple column statistics on the base table (see [Collecting Statistics on Multiple Columns](#)).
 - A much poorer option is to collect the statistics on the column set for the single-table join index. If you do this, you *must* use the COLLECT STATISTICS ON ... INDEX syntax to collect the statistics for that column set on the single-table join index.

Guidelines for Collecting Statistics On Single-Table Join Index Columns

The guidelines for selecting single-table join index columns on which to collect statistics are similar to those for base tables and hash indexes. The primary factor to consider in all cases is whether the statistics provide better access plans. If they do not, consider dropping them. If the statistics you collect produce *worse* access plans, then you should always report the incident to Teradata Services personnel.

When you are considering collecting statistics for a single-table join index, it might help to think of the index as a special kind of base table that stores a derived result. For example, any access plan that uses a single-table join index must access it with a direct probe, a full table scan, or a range scan.

With this in mind, consider the following factors when deciding which columns to collect statistics for:

- Always consider collecting statistics on the primary index if the join index has one. This is particularly critical for accurate cardinality estimates.
- Consult the following table for execution plan cases that suggest collecting specific statistics.

| IF an execution plan might involve ... | THEN collect statistics on the ... |
|---|---|
| search condition keys | column set that constitutes the search condition. |
| joining the single-table index with another table | join columns to provide the Optimizer with the information it needs to best estimate the cardinalities of the join. |

- Consult the following table for single-table join index cases that suggest collecting specific statistics.

| IF a single-table join index is defined ... | THEN you should collect statistics on the ... |
|--|--|
| with an ORDER BY clause | order key specified by that clause. |
| without an ORDER BY clause and the order key column set from the base table is not included in the <i>column_name_1</i> list | order key of the base table on which the index is defined. This action provides the Optimizer with several essential baseline statistics. |

- If a single-table join index column appears frequently in WHERE clauses, you should consider collecting statistics on it as well, particularly if that column is the sort key for a value-ordered single-table join index.
- If a single-table join index is defined with a complex expression in its select list that is also specified in a predicate expression written on a mapped base table column, collecting join index statistics on the expression enhances the ability of the Optimizer to estimate single-table cardinalities for a query that specifies the expression on the base table because it can use those statistics directly to estimate the selectivity of complex expressions on base table columns specified in the query (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for details).

Collecting Statistics on Multitable Join Indexes

Statistics for multitable join indexes and the base tables they index are not interchangeable. Whenever a column is defined both for the base table and for a multitable join index defined on it, you must collect statistics separately on both columns.

The demographics for column values stored in a base table are often very different than those stored for a derived join result in a multitable join index. When statistics have not been collected on a multitable join index, the Optimizer does *not* derive corresponding statistics from the statistics collected on its underlying base tables.

You can collect statistics on a simple data column or secondary index column set of a join index. It is instructive to think of a multitable join index as a special base table representing a derived join result. You might need to collect statistics on search condition keys to help the Optimizer to evaluate alternative access paths.

Execution plans can also involve joining a multitable join index table with yet another table that is not part of the multitable join index. When your EXPLAIN statements indicate that this is happening, you should also collect statistics on those join columns to help the Optimizer to estimate cardinalities.

Before creating a new multitable join index, you should consider performing the following procedure to improve the performance of its creation as well as its update maintenance.

1. Collect statistics on the base tables underlying the planned multitable join index.
2. Immediately create the new multitable join index.

Compression of Join Index Column Values

When you create a join index, the database automatically transfers any column multivalue compression defined on its base table set, with a few exceptions, to the join index definition. This is different from the situation for hash indexes, which *cannot* inherit the multivalue compression characteristics of their parent base tables.

The following rules and restrictions apply to automatically transferring the column compression characteristics of a base table to its underlying join index columns. All of these rules and restrictions must be met for base table column multivalue compression to transfer to the join index definition.

- Base table column multivalue compression transfers to a join index definition even if there is an alias name specified for the columns in the join index definition.
- Base table column multivalue compression transfers to a multitable join index definition only up to the point that the maximum table header length of the join index is exceeded.

The CREATE JOIN INDEX request does not abort at that point, but the transfer of column multivalue compression from the base table to the join index definition stops.

- Base table column multivalue compression does *not* transfer to a join index definition if the column is a component of the primary index for the join index.
- Base table column multivalue compression does *not* transfer to a join index definition for any of the columns that are components of a partitioned primary index or the partitioning expression for the join index.
- Base table column multivalue compression does *not* transfer to a join index column if the column is specified as the argument for aggregate or EXTRACT functions.
- Base table column multivalue compression does *not* transfer to a column in a compressed join index if the column is a component of a NUSI defined on the *column_1* and *column_2* indexes.
- If you use an ALTER TABLE request to modify the compression characteristics of any base table columns that are also components of a join index, the database returns a warning message advising you that you must recreate the join index to reflect those compression changes. See [ALTER TABLE \(Basic Table Parameters\)](#).
- Base table column multivalue compression does *not* transfer to a join index definition if the column is a component of an ORDER BY clause in the join index definition.

Row Compression of Join Indexes

The term *compression* has multiple meanings for the Teradata system. Multivalue compression and row compression forms are lossless, meaning that the original data can be reconstructed exactly from their compressed forms.

- The storage organization for join indexes supports a compressed format to reduce storage space.

If you know that a join index contains groups of rows with repeating information, then its definition DDL can specify repeating groups, indicating the repeating columns within a separate set of parentheses from the fixed columns. In other words, the column list is specified as two groups of columns, with each

group delimited within parentheses. The first group contains the repeating columns and the second group contains the fixed columns.

You cannot define both row compression and a row-partitioned primary index for the same join index.

When describing compression of join index rows, compression refers to a logical row compression in which multiple sets of non-repeating column values are appended to a single set of repeating column values. Vantage stores the repeating value set only once, while any non-repeating column values are stored as logical segmental extensions of the base repeating set.

For example, suppose you have a join index *ji_no_comp* that has five rows. The join index table might look something like this.

| column_1 | column_2 | column_3 | column_4 | column_5 |
|----------|----------|----------|----------|----------|
| 1 | 2 | 0 | 2 | 4 |
| 2 | 4 | 8 | 7 | 7 |
| 1 | 2 | 2 | 3 | 1 |
| 1 | 2 | 5 | 3 | 3 |
| 2 | 4 | 3 | 4 | 6 |

When you examine the rows of this join index, you notice several repetitions of the value 1 in *column_1* paired with the value 2 in *column_2*. These repeating pairs are highlighted in teal. You also notice that there are several repetitions of the value 2 in *column_1* paired with a 4 in *column_2*. These repeating pairs are highlighted in tan. None of the values for *column_3*, *column_4*, or *column_5* repeat.

By defining a new compressed join index for these rows, you can reduce the number of rows stored from five down to two. To do this, you would define the index something like this.

```
CREATE JOIN INDEX ji_comp AS
  SELECT (column_1, column_2), (column_3, column_4, column_5)
  FROM table_zee
  PRIMARY INDEX (column_3);
```

Because of the way the columns are parenthetically grouped in this request, Vantage knows that the first group of columns represents a column set whose values repeat, while the second group of columns represents a column set whose values do not repeat.

You can think of the way Vantage stores the data from the original 5 rows as being something like the following 2 rows, reducing the storage space required for the index from five rows to two. Join index *ji_comp* uses less disk space than *ji_no_comp*, but logically represents the same 5 rows. In this representation the repeating column values are highlighted in teal and their accompanying fixed column values are highlighted in tan:

| | | | |
|-------|---------|---------|---------|
| (1,2) | (0,2,4) | (2,3,1) | (5,3,3) |
|-------|---------|---------|---------|

| | | | |
|-------|---------|---------|--|
| (2,4) | (8,7,7) | (3,4,6) | |
|-------|---------|---------|--|

For more information about the syntax you must use to specify join index row compression, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- When describing compression of column values, compression refers either to multivalued compression, which is the storage of column values one time only in the table header, not in the row itself, and pointing to them by means of an array of presence bits in the row header, or algorithmic compression, which is a user-defined column multivalued compression method using UDFs. See [Compression of Join Index Column Values](#). For information about creating algorithmic compression UDFs, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The method Vantage uses for multivalued compression is called Dictionary Indexing, and it can be viewed as a variation of Run-Length Encoding. For additional information, see *Teradata Vantage™ - Database Design*, B035-1094.

For more information about join index row compression, see [Physical Join Index Row Compression](#).

Physical Join Index Row Compression

A physical join index row has two parts:

- A required fixed part that is stored only once.
- An optional repeated part that is stored as often as needed.

For example, if a logical join result has n rows with the same fixed part value, then there is one corresponding physical join index row that includes n repeated parts in the physical join index. A physical join index row represents one or more logical join index rows. The number of logical join index rows represented in the physical row depends on how many repeated values are stored.

Row compression is only done on rows inserted by the same INSERT statement. Newly inserted rows are not added as logical rows to existing compressed rows.

When the number of repeated values associated with a given fixed value exceeds the system row maximum size, the join index row is automatically split into multiple physical rows, each having the same fixed value but different lists of repeated values. A logical join index result row cannot exceed the system row maximum size of 1 megabyte.

Guidelines for Using Row Compression With Join Index Columns

A column set with a high number of distinct values cannot be row compressed because it rarely (and in the case of a primary key or unique index, never) repeats. Other column sets, notably foreign key and status code columns, are generally highly non-distinct: their values repeat frequently. Row compression capitalizes on the redundancy of frequently repeating column values by storing them once in the fixed part of the index row along with multiple repeated values in the repeated part of the index row. Typically, primary key table column values are specified as the repeating part and foreign key table columns are specified in the fixed part of the index definition.

See *Teradata Vantage™ - Database Design*, B035-1094 for more information about row compression for join indexes.

Query Coverage by Join Indexes

When the columns requested by a query can be retrieved from an index instead of scanning the base table that index supports, the index is said to *cover* the query. Some vendors refer to this as index-only access. Although secondary indexes can sometimes cover a minor query, only hash and join indexes are generally called upon to respond to queries of any consequence. If a join index is defined correctly, it can also cover a query that requests columns it does *not* carry. This situation is called partial query coverage, and when a join index partially covers the query, the Optimizer can weigh the cost of using it against the cost of scanning the base table set it supports. The criteria for defining a join index to make it eligible for consideration as a partial query cover are described in the following paragraphs.

Other, more complicated, criteria for determining whether a join index can cover a query are described in [Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#), and [Rules for Whether Join Indexes With Extra Tables Cover Queries](#).

Be aware that a join index defined with an expression in its select list has less coverage than a join index that is specified using base columns in its select list.

For example, the Query Rewrite Subsystem can use the following join index, *ji_f1*, to rewrite a query that specifies any character function on column *f1*,

```
CREATE JOIN INDEX ji_f1 AS
  SELECT b1, f1
  FROM t1
  WHERE a1 > 0;
```

However, the Query Rewrite Subsystem can only use join index *ji_substr_f1* to rewrite a query that specifies the same SUBSTR function on column *f1* as join index *ji_substr_f1* specifies in its select list.

```
CREATE JOIN INDEX ji_substr_f1 AS
  SELECT b1, SUBSTR(f1,1,10) AS s
  FROM t1
  WHERE a1 > 0;
```

See *Teradata Vantage™ - Database Design*, B035-1094 for information about using global join indexes with tactical queries.

A join index that has the capability of accessing base table columns via a prime access key specified in its definition DDL is eligible for partial query covering. A join index defined in this way is sometimes referred to as a global index or a global join index.

Support for global join indexes is provided by the following substitution logic:

| IF the definition for a single-table join index ... | THEN that join index ... |
|--|--|
| <p>specifies any one or more of the following items:</p> <ul style="list-style-type: none"> • ROWID keyword as a column name in the <i>column_name</i> list You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. • Column set that defines the UPI of the underlying base table • Column set that defines the NUPI of the underlying base table plus the ROWID keyword • Column set that defines the NUPI of the underlying base table plus a column set that defines a USI on the underlying base table | <p>qualifies as a partial covering index if it also contains a subset of the columns specified by a query.</p> <p>Partial coverage is not restricted to single-table join indexes.</p> <p>Join indexes defined with one of these column sets are sometimes referred to as global indexes or global join indexes. See <i>Teradata Vantage™ - Database Design</i>, B035-1094 for some specific applications of global join indexes.</p> <p>The Optimizer specifies a join from the join index to the base table to pick up columns that are requested by the query but not included in the single-table join index definition.</p> |
| does not specify the ROWID keyword, base table UPI columns, or base table NUPI columns plus either the ROWID keyword or a base table USI column set | cannot be used to partially cover a query. |

The Optimizer specifies a join from the join index to the base table to pick up columns that are requested by the query but not included in the join index.

| IF this item is specified in the join index definition ... | THEN the join from the single-table join index to the base table is made using this join type ... |
|--|---|
| <p>the base table ROWID only</p> <p>You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement.</p> | Row ID |
| the base table primary index | <p>Merge</p> <p>In this case, both the primary index column set and the ROWID, if present, are used to make the join.</p> |

Even though you do not explicitly specify this join when you write your query, it counts against the 128 tables and views per query block restriction on joins.

| IF this item is specified in the join index definition ... | THEN it is ... |
|--|---|
| UPI | <p>not necessary to include ROWID in the index definition to enable it as a partial cover because a UPI is sufficient to identify any base table row.</p> <p>You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement.</p> |
| NUPI | <p>necessary to include either of the following specifications to make it eligible for partial coverage of a query.</p> <ul style="list-style-type: none"> • The base table NUPI column set <i>and</i> ROWID in the index definition. |

| IF this item is specified in the join index definition ... | THEN it is ... |
|---|--|
| | <ul style="list-style-type: none"> The base table NUPI column set <i>and</i> a base table USI column set in the index definition. <p>The NUPI plus ROWID option is the preferable choice.</p> |

The Optimizer specifies a partially covering join index in a join plan only if the cost of using it is less than the cost of *not* using it, just as it does for an index that fully covers the query.

For example, consider the following table and join index definitions:

```
CREATE TABLE t1 (
  x1 INTEGER,
  y1 INTEGER,
  z1 INTEGER)
PRIMARY INDEX (x1);

CREATE TABLE t2 (
  x2 INTEGER,
  y2 INTEGER,
  z2 INTEGER)
PRIMARY INDEX (x2);

CREATE JOIN INDEX j1 AS
  SELECT y1
  FROM t1
PRIMARY INDEX (y1);
```

The Optimizer selects single-table join index *j1* as a partial covering index for its join plan for the following query.

```
EXPLAIN
SELECT x1, y1, z2
FROM t1, t2
WHERE y1 = x2
AND   y2 = 1;
```

Explanation

-
- 1) First, we lock MyDB.t2 for read on a reserved RowHash to prevent global deadlock.
 - 2) Next, we lock MyDB.t1 for read on a reserved RowHash to prevent

global deadlock.

- 3) We lock MyDB.t2 for read, and we lock MyDB.t1 for read.
 - 4) We do an all-AMPs RETRIEVE step from MyDB.t1 by way of an all-rows scan with a condition of ("NOT (MyDB.t1.y1 IS NULL)") into Spool 2 (all_amps), which is redistributed by the hash code of (MyDB.t1.y1) to all AMPs. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated with low confidence to be 4 rows (84 bytes). The estimated time for this step is 0.03 seconds.
 - 5) We do an all-AMPs JOIN step from MyDB.t2 by way of a RowHash match scan with a condition of ("MyDB.t2.y2 = 1"), which is joined to Spool 2 (Last Use) by way of a RowHash match scan. MyDB.t2 and Spool 2 are joined using a merge join, with a join condition of ("y1 = MyDB.t2.x2"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 4 rows (132 bytes). The estimated time for this step is 0.11 seconds.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.13 seconds.

In this query, joining tables *t1* and *t2* would require redistributing all the *t1* rows by *t1.y1*. This is an expensive operation if *t1* is very large.

An alternative, more cost-effective solution is to perform a local join between join index *j1* and base table *t2* and then to redistribute the intermediate join result to join with base table *t1*. Since there is a precise constraint condition on base table *t2* that *t2.y2 = 1*, redistributing the join result of *j1* and *t2* and then joining the result to *t1* is less expensive than the redistribution cost of *t1*. Therefore, the Optimizer selects *j1* to process the *t1-t2* join.

By contrast, consider a similar query without the *t2.y2 = 1* constraint. In this case, the Optimizer does not select *j1* to cover the query even though it qualifies as a partially covering index:

```
EXPLAIN SELECT x1, y1, z2
FROM t1,t2
WHERE y1 = x2;
```

```
*** Help information returned. 22 rows.
*** Total elapsed time was 1 second.
```

Explanation

-
- 1) First, we lock MyDB.t2 for read on a reserved RowHash to prevent global deadlock.
 - 2) Next, we lock MyDB.t1 for read on a reserved RowHash to prevent global deadlock.
 - 3) We lock MyDB.t2 for read, and we lock MyDB.t1 for read.

- 4) We do an all-AMPs RETRIEVE step from MyDB.t1 by way of an all-rows scan with a condition of ("NOT (MyDB.t1.y1 IS NULL)") into Spool 2 (all_amps), which is redistributed by the hash code of(MyDB.t1.y1) to all AMPs. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated with low confidence to be 4 rows (84 bytes). The estimated time for this step is 0.03 seconds.
 - 5) We do an all-AMPs JOIN step from MyDB.t2 by way of a RowHash match scan, which is joined to Spool 2 (Last Use) by way of a RowHash match scan. MyDB.t2 and Spool 2 are joined using a merge join, with a join condition of ("y1 = MyDB.t2.x2"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with index join confidence to be 6 rows (198 bytes). The estimated time for this step is 0.11 seconds.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.13 seconds.

Restriction on Partial Coverage of Queries Containing a TOP *n* or TOP *m* PERCENT Clause

A join index cannot be used to partially cover a query that specifies the TOP *n* or TOP *m* PERCENT option.

Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query

Whether the Optimizer decides to include a join index in its query plan is a more complicated choice than simply determining if the index contains all the table columns specified in the query. The columns on which the base tables in the index definition are joined and their respective referential constraints also play an important role. See [Rules for Whether Join Indexes With Extra Tables Cover Queries](#).

In many cases, the Optimizer does not consider using a join index in its access plan if that index is defined on more tables than the query references.

The join index is more likely to be used in the query access plan if the extra inner joins are defined on primary key-foreign key relationships in the underlying base tables that ensure proper row preservation. The referential integrity relationship between the base table primary and foreign keys can be specified using any of the three available methods for establishing referential constraints between tables.

In the case of join index outer joins, outer table rows are always preserved automatically, so there is no requirement for a referential integrity constraint to exist to enable their preservation.

In the case of foreign key-primary key inner joins, the same preservation of rows follows from a declarative referential integrity constraint. In this case, the Optimizer does consider a join index with extra inner joins in its definition to cover a query. The following paragraphs explain why a referential constraint preserves logical integrity.

Assume that the base tables in a join index definition can be divided into two distinct sets, s_1 and s_2 .

s_1 contains the base tables referenced in the query, while s_2 contains the extra base tables the query does not reference. The base tables in s_1 are joined to the base tables in s_2 on foreign key-primary key columns, with the tables in s_2 being the primary keys in the relationships. The foreign key values cannot be null because if the foreign key column set contains nulls, the losslessness of the foreign key table cannot be guaranteed.

The following assertions about these base tables are true:

- The extra joins do not eliminate valid rows from the join result among the base tables in s_1 because FOREIGN KEY and NOT NULL constraints ensure that every row in the foreign key table finds its match in the primary key table.
- The extra joins do not introduce duplicate rows in the join result among the base tables in s_1 because the primary key is, by definition, unique and not nullable.

These assertions are also true for extra joins made between base tables that are both in s_2 .

Therefore, the extra joins in a join index definition, if made on base tables that are defined in a way that observes these assertions, preserve all the rows resulting from the joins among the base tables in s_1 and do not add spurious rows.

This result permits the Optimizer to use the join index to cover a query that references fewer tables than the index definition inner joins together.

A covering join index whose definition includes one or more tables that is not specified in the query it covers is referred to as a broad join index. A wide range of queries can make use of a broad join index, especially when there are foreign key-primary key relationships defined between the fact table and the dimension tables that enable the index to be used to cover queries over a subset of dimension tables. For more information about broad join indexes, see *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

The Optimizer can select a join index for a query plan if the index contains either the same set, or a subset, of the tables referenced by the query. If more tables are referenced in the join index definition than are referenced by the query, the Optimizer generally does not consider that index as a candidate for coverage because the extra joins can either eliminate rows or produce duplicate rows or both.

Because a referential integrity relationship guarantees the losslessness of the foreign key table in the join with its primary key table, extra tables in a join index definition do not disqualify it from query plan consideration if the extra joins allow the join index to preserve all the rows for the join result of the subset of tables in the query.

For example, suppose you define a join index on a set of 5 tables, t_1 - t_5 , respectively, with foreign key-primary key joins in the directions indicated (foreign key table followed by its parent primary key table):

t_1 t_2 t_3 t_4 t_5

Queries that reference the following table subsets can be covered by this join index because the extra joins, either between two tables where one is in the query and the other is not, or between two tables that are both not in the query, do not cause any loss of rows for the join result of the subset of tables in the query:

- t1
- t1, t2
- t1, t2, t3
- t1, t2, t3, t4

As a result of this property, the following conditions that reference extra joins can be exploited by the Optimizer when the number of tables referenced by the join index definition exceeds the number of tables referenced by the query. In each case, x1 is a unique RI-related column in table t1 and x2 is a unique RI-related column in table t2:

| Join Index Extra Join Condition | Qualifications |
|---------------------------------|---|
| x1 = x2 | <ul style="list-style-type: none"> • x1 is the foreign key in the relationship. • t1 is referenced by the query. • t2 is not referenced by the query. |
| x1 = x2 | <ul style="list-style-type: none"> • x1 is the primary key in the relationship. • t2 is referenced by the query. • t1 is not referenced by the query. |
| x1 = x2 | <ul style="list-style-type: none"> • x1 is the foreign key in the relationship. • x2 is the primary key in the relationship. • Neither t1 nor t2 is referenced by the query. |
| x1 = x2 | <ul style="list-style-type: none"> • x1 is the primary key in the relationship. • x2 is the foreign key in the relationship. • Neither t1 nor t2 is referenced by the query. |

One restriction with two critical exceptions must be added to the above optimization to make the coverage safe: when one table referenced in the join index definition is the parent table of more than one FK table, the join index is generally disqualified from covering any query that references fewer tables than are referenced in the join index.

For example, suppose you define a join index on a set of 5 tables, t1 - t5, respectively, with foreign key-primary key joins in the directions indicated (foreign key table followed by its parent primary key table) by the following diagram.

t1 t2 t3 t4 t5

For these RI relationships, table t4 is the parent table of both tables t3 and t5. The losslessness of the foreign key table depends on the fact that the parent table has all the primary keys available. Joining the parent table with another child table can render this premise false. Therefore, the final join result cannot be viewed as lossless for any arbitrary subset of tables.

The following two points are exceptions to this restriction:

- When the foreign key tables are joined on the foreign key columns, there is no loss on any of the foreign key tables because they all reference the same primary key values in their common parent table.

- All foreign key tables reference the same primary key column in the primary key table. By transitive closure, all these foreign key tables are related by equijoins on the foreign key columns.

By specifying extra joins in the join index definition, you can greatly enhance its flexibility.

For example, suppose you have a star schema based on a sales fact table and the following dimension tables:

- customer
- product
- location
- time

You decide it is desirable to define a join index that joins the fact table sales to its various dimension tables to avoid the relatively expensive join processing between the fact table and its dimension tables whenever ad hoc join queries are made against them.

If there are foreign key-primary key relationships between the join columns, which is often the case, the join index can also be used to optimize queries that only reference a subset of the dimension tables.

Without taking advantage of this optimization, you must either create a different join index for each category of query, incurring the greater cost of maintaining multiple join indexes, or you lose the benefit of join indexes for optimizing the join queries on these tables altogether. By exploiting the foreign key-primary key join properties, the same join index can be selected by the Optimizer to generate access plans for a wide variety of queries.

Rules for Whether Join Indexes With Extra Tables Cover Queries

The following rules explain how to design a set of underlying base tables for a join index definition in such a way to ensure that the Optimizer selects the index for an access plan if it inner joins more tables than the query references. Such join indexes are referred to as broad join indexes. For more information on broad join indexes, see [Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#), *Teradata Vantage™ - Database Design*, B035-1094, and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

| IF there are more inner-joined tables in a join index definition than the number of tables referenced in a query and ... | THEN the Optimizer ... |
|--|--|
| the extra joins are <i>not</i> made on foreign key-primary key columns in the underlying base tables | does <i>not</i> consider the join index for the query plan. This is because the presence of extra joins in the definition can either eliminate existing rows from the query evaluation or produce duplicate rows during optimization. |
| the extra joins <i>are</i> made on foreign key-primary key columns in the underlying base tables | considers the join index for use in for the query plan. |
| both of the following conditions are true: | |

| IF there are more inner-joined tables in a join index definition than the number of tables referenced in a query and ... | THEN the Optimizer ... |
|---|------------------------|
| <ul style="list-style-type: none"> • The join column set of the inner table in the extra outer join is unique • Either the inner table or both the inner and outer tables involved in the extra outer join are extra tables | |

Restriction on Number of Join Indexes Selected by the Optimizer Per Table

The Optimizer can use several join indexes for a single query, selecting one or more multitable join indexes as well as additional single-table join indexes for its join plan. The join indexes selected depend on the structure of the query, and the Optimizer might not choose all applicable join indexes for the plan. Always examine your EXPLAIN reports to determine which join indexes *are* used for the join plans generated for your queries. If a join index you think should have been used by a query was not included in the join plan, try restructuring the query and EXPLAIN it once again.

The join planning process selects a multitable join index to replace any individual table in a query when the substitution further optimizes the query plan. For each such table replaced in the join plan by a multitable join index, as many as two additional single-table join indexes can also be considered as replacements if their inclusion reduces the size of the relation to be processed, provides a better distribution, or offers additional covering.

The limit on the number of join indexes substituted per individual table in a query is enforced to limit the number of possible combinations and permutations of table joins in the Optimizer search space during its join planning phase. The rule helps to ensure that the optimization is worth the effort. This means that the time spent generating the query plan should not exceed the accrued performance enhancement gained from the optimization.

Restrictions and Limitations for Error Tables

You cannot create a join index on an error table (see [CREATE ERROR TABLE](#)).

Restrictions and Limitations for Row-Level Security Columns

You can create a join index on a row-level security-protected table only if all of the following rules are followed.

- The index references a single row-level security-protected table and no other table.
- All of the constraint columns in the table are included in the index definition.

Restrictions and Limitations for SQL UDFs

You cannot invoke an SQL UDF anywhere in a join index definition.

Restrictions and Limitations for the EXPAND ON Clause

You cannot specify an EXPAND ON clause anywhere in a join index definition.

Restrictions and Limitations for Ordered Analytical Functions

When an ordered analytical function is specified on columns that are also defined for a row compressed join index, the Optimizer does not select the join index to process the request.

Restrictions and Limitations for Join Index Normalization

You cannot specify the NORMALIZE option for a join index definition.

Restrictions and Limitations for Load Utilities

You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have join indexes because those indexes are not maintained during the execution of these utilities. This is true both for partitioned and nonpartitioned join indexes. If you attempt to load data into base tables with join indexes using these utilities, an error message returns and the load does not continue.

To load data into a join-indexed base table, you must drop all defined join indexes before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Load utilities like Teradata Parallel Data Pump, BTEQ, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, *are* supported for base tables on which join indexes are defined.

Restrictions and Limitations for the Reconfig Utility

You must drop and rebuild all value-ordered join indexes after you run the Reconfig utility. For more information, see *Support Utilities*, B035-1180.

The same is *not* true for hash-ordered join indexes.

Restrictions on Creating a Join Index on a Table Concurrent With Dynamic AMP Sample Emulation on That Table

You cannot create a join index for a table while that table is subject to dynamic AMP sample emulation. To disable dynamic AMP sampling, contact the Teradata Support Center.

To use dynamic AMP sampling on the table with a new join index, use the following general procedure.

1. Create the new join index on the table on the target system.
2. Extract a fresh dynamic AMP sample from the target system.

3. Apply the fresh sample to the source system.

Permanent Journal Recovery of Join Indexes Is Not Supported

You can use ROLLBACK or ROLLFORWARD statements to recover base tables that have join indexes defined on them; however, the join indexes are *not* rebuilt during the recovery process. Instead, they are marked as not valid. You must drop and recreate any such join indexes before the Optimizer can use them again to generate a query plan.

When a join index has been marked not valid, the SHOW JOIN INDEX statement displays a special status message to inform you that the join index has been so marked.

Block-Level Compression and Join Indexes

Join indexes can be compressed at the data block level or not, depending on whether they have been compressed or decompressed using the Ferret utility. For information about the Ferret utility, see *Utilities*.

All limits on data block sizes apply to the noncompressed size of a join index. Block compression does not raise any of these limits, nor does it enable more data to be stored in a single data block than can be stored in an noncompressed data block of the same size.

CREATE MACRO and REPLACE MACRO

Performing a macro frees you from having to type the entire set of statements in the macro body each time the operation defined by these statements is needed. Macros simplify an operation that is complex or must be performed frequently.

Restriction on External Authorization From a Macro

You cannot specify CREATE AUTHORIZATION in a macro because the authorization password is not saved in either encrypted or unencrypted form. This would compromise the security of the OS logon user ID. See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#).

Otherwise, the system returns an error to the requestor.

Running a Macro

Running a macro constitutes an implicit transaction. Therefore, in Teradata session mode, a macro need not be enclosed between BEGIN TRANSACTION and END TRANSACTION statements.

When the session performing a macro is in ANSI mode, the actions of the macro are uncommitted until a commit or rollback occurs in subsequent statements unless the macro body ends with a COMMIT statement. If you define a macro using a COMMIT statement, then it can be performed only in sessions running in ANSI mode.

The following table explains how to define and use a macro so it can be committed in both ANSI and Teradata session modes:

| IF a macro is to be performed in this session mode ... | THEN ... |
|--|---|
| ANSI exclusively | add a COMMIT WORK statement to the end of its definition. |
| both ANSI and Teradata | omit the COMMIT WORK statement from the definition. Tell users to specify a COMMIT WORK statement explicitly whenever they perform the macro in ANSI session mode. |

Users do not receive responses to individual statements contained in the macro body until all its requests have been completed successfully. Any object that is referenced by a request is locked until the macro transaction is completed, or until the macro is terminated because of a statement error.

If a macro contains a data definition statement, it cannot contain other requests. A data definition statement in a macro is not fully resolved until the macro is performed. At that time, unqualified references to database objects are resolved using the default database for the user submitting the EXECUTE statement. It is therefore recommended that object references in a macro data definition statement be fully qualified in the macro body.

Limit On Request Size

The maximum size of the fully expanded text for a macro is 2 MB.

Because expansions of source text in macro definitions are made to fully qualify object names and to normalize expressions, it is possible for a macro to be defined but also to be unusable because of stack overflows in the Syntaxer at performance time. In this case, the system returns an error.

Resolution of Non-Qualified Names

When a CREATE MACRO request is performed, the system parses the text of the requests specified in the macro and resolves any names that are not fully qualified in those requests. Note that non-qualified names in DDL requests in macros are left unresolved.

Consider the following example, in which the default database for user *xyz* is database *abc*, and the database for user *lmn* is database *def*. User *xyz* defines *macro_1* as follows:

```
CREATE MACRO macro_1 AS (
  CREATE TABLE mambo (
    column_1 INTEGER,
    column_2 INTEGER);
);
```

The fully qualified table name is not specified in the CREATE TABLE request.

Because CREATE TABLE is a DDL statement, the name of its containing database is not resolved when the macro is created. Instead, the name is resolved later, when the macro is performed. The result is that the macro always observes the correct storage hierarchy when it is performed.

The following table shows how this works:

| IF this user performs the macro named <i>macro_1</i> ... | THEN a table named <i>mambo</i> is created in this database ... |
|--|---|
| xyz | abc |
| lmn | def |

Now suppose that user *xyz* creates the macro named *macro_2* as indicated here:

```
CREATE MACRO macro_2 AS (
  UPDATE tango
  SET counter = counter + 1);
```

Note that the fully-qualified table name is *not* specified in the UPDATE request.

Because UPDATE is a DML statement, it is fully resolved when *macro_2* is created. Subsequently, performing *macro_2* causes *abc.tango* to be updated, whether performed by *abc*, *lmn*, or any other user.

To summarize, the following rules apply to non-qualified names when they are specified in macros:

- Non-qualified names in a macro definition are *not* resolved in DDL statements when the macro is created. They are not resolved until the macro is performed.
- Non-qualified names in a macro definition are *fully* resolved in DML statements when the macro is created.

Semantic Errors Are Sometimes Not Reported At the Time a Macro Is Created

The system checks the macro text for syntax errors, but not for semantic errors for EXEC and DDL statements. Because of this, it is possible to create a macro that is syntactically valid, but not valid semantically. No message is returned when this occurs.

If there are semantic errors in the macro definition, then a message is returned to the requestor when you perform it.

For example, it is possible to create or replace a macro definition that contains an EXEC request for a macro that is not defined in the system. The following example creates the macro without returning an error even though the macro it performs, *no_such_macro*, is not defined in the database:

```
CREATE MACRO test_macro AS (
  EXEC no_such_macro);
```

The system creates the macro as requested. It is only when you attempt to perform *test_macro* that you discover the semantic error:

```
EXEC test_macro;

EXEC test_macro;

*** Failure 3824 Macro 'no_such_macro' does not exist.
      Statement# 1, Info =0
*** Total elapsed time was 1 second.
```

This failure to enforce semantic correctness is restricted to EXEC requests and DDL requests. If a macro definition contains any DML requests, then the objects those statements reference must exist or the system aborts the attempt to create the macro.

Because the following CREATE MACRO request contains a reference to the table named *table_1* that does not exist, the system aborts it and returns an error to the requestor because *table_1* does not exist.

```
CREATE MACRO test_macro_2 AS (
  SELECT *
  FROM table_1);

CREATE MACRO test_macro_2 AS (SELECT * FROM table_1);

*** Failure 3807 Object 't1' does not exist.
      Statement# 1, Info =0
*** Total elapsed time was 1 second.
```

Using the ASTERISK (*) Character in Macros

A macro defined using the ASTERISK (*) character is bound to the definitions of any tables it references as they were defined at the time the macro was created or replaced.

For example, consider the following macro:

```
CREATE MACRO get_emp AS (
  SELECT *
  FROM employee;)
```

If a column is later added to or removed from the employee table, the following statement still returns the number of columns that existed in employee when *get_emp* was defined:

```
EXEC get_emp;
```

If columns have been dropped or data types have been changed, performing the macro can result in an error message or unexpected behavior.

Rules for Using Parameters in Macros

Parameters are values that are entered by the user into the EXEC request for use by the macro during execution. The following rules apply to their use in CREATE MACRO and REPLACE MACRO requests:

- You cannot pass the names of database objects to a macro as parameters.
This refers to tables and views and their columns, databases, users, and similar database objects.
You can use dynamic SQL within a stored procedure. For information on how to use dynamic SQL within a stored procedure, see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.
- Use of parameters is optional in a macro and, when required, must be specified as part of the CREATE MACRO or REPLACE MACRO request.
- In defining parameters for a macro, follow the macro name in your CREATE/REPLACE MACRO request with the names and attributes of the appropriate parameters. Note that data type definitions are required for each parameter. Other attributes can include format specifications or default values. Define new formats via a FORMAT phrase and defaults by a default control phrase, as necessary.
- If you specify them where either an ordinal positional integer number or an expression is valid, such as an ordinary grouping set in a GROUP BY clause or the ordering specification for an ORDER BY clause, Vantage treats parameters referenced by requests that are contained within a macro as expressions, not as constant literals.

As a result, Vantage does *not* apply such parameter expressions as constant literals, if it applies them at all. Instead, you should specify grouping and ordering specifications explicitly within the body of the macro. See the information about the GROUP BY and ORDER BY clauses in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

- If you supply a parameter value in the EXEC request that does not conform to the specified format, data type, or default value, the request aborts and returns a message to the requestor.
- The maximum number of parameters you can specify per macro is 2,048.
- If you specify a query band by means of a macro parameter, it can only be a *transaction* query band.

If you submit a CREATE MACRO or RENAME MACRO request that uses a parameter to specify a *session* query band, the request aborts and the system returns a message to the requestor.

Macro Definitions and Server Character Sets

A macro can contain explicit [VAR]CHARACTER(*n*) [CHARACTER SET ...] clauses. If the server character set (for example, CHARACTER SET LATIN) is not specified, the macro definition expands the [VAR]CHARACTER(*n*) clause.

Expansion is done according to the following rules:

- If the clause is applied to the macro parameter, it takes the user default server character set of the creator or modifier of the macro (creator user default server character set).

For example, if the creator user default server character set is Unicode, then the macro is expanded as shown in the immediately following example:

Original macro definition:

```
CREATE MACRO mm AS (
  column_a CHARACTER(5))
(SELECT :column_a);;
```

Expanded macro definition:

```
CREATE MACRO mm AS (
  column_a CHARACTER(5) CHARACTER SET UNICODE)
(SELECT :column_a);;
```

- If the clause is applied to an expression within the body of the macro, it takes the server character set of the expression if the expression is typed as CHARACTER. Otherwise, the server character set is set to the default server character set for the creator user.

For example, if the creator user default server character set is UNICODE, then the macro is qualified as shown in the immediately following example when it is performed.

Original macro definition:

```
CREATE TABLE table_2 (
  ck CHARACTER(5) CHARACTER SET KANJI1);

CREATE MACRO mm AS (
  SELECT ck (CHARACTER(6))
  FROM table_2);;
```

Qualified macro definition:

```
SELECT ck (CHARACTER(6), CHARACTER SET KANJI1)
FROM table_2;
```

When the following macro is performed, it is qualified as seen in the SELECT request that immediately follows it.

Original macro definition:

```
CREATE MACRO mm AS (
  SELECT 123 (CHARACTER(12)));;
```

Qualified macro definition:

```
SELECT 123 (CHARACTER(6), CHARACTER SET UNICODE);
```

- If the macro contains a CREATE TABLE or CREATE VIEW with character definition and the CHARACTER SET clause is omitted, the macro expansion takes the user default server character set of the user who performs it.

For example, suppose the creator user default server character set is UNICODE. When the macro definition below is performed by a different user having a user default server character set of LATIN, then *table_1* is created as shown in the immediately following example.

Original macro definition:

```
CREATE MACRO tt AS (
  CREATE TABLE table_1
    (column_1 CHARACTER(5)););
```

Expanded macro definition:

```
CREATE TABLE table_1 (
  column_1 CHARACTER(5) CHARACTER SET LATIN NOT CASESPECIFIC);
```

The same applies to a CREATE VIEW statement within a CREATE MACRO statement.

You cannot specify a character data type that has a server character set of KANJI1.

Macro Support for Dates

Validation of date strings in a macro takes place when it is performed. The format of date strings validate against a DATE column format or the default DATE format, both of which can change after creating the macro.

To guarantee that dates are properly validated, create or replace your macros by doing either or both of the following:

- Specify dates as ANSI date literals instead of strings.
ANSI date literals are accepted as dates for any date operation.
- Specify the date format you want in a FORMAT phrase in the macro.

Macro Support for Global Temporary and Volatile Tables

You can reference both global temporary tables and volatile tables from a macro.

When you perform a macro that involves a global temporary table, then all references to the base temporary table are mapped to the corresponding materialized global temporary table within the current session. Note that this means the same macro, when performed from two different sessions, can have very different results depending on the content of the materialized global temporary tables.

When you perform a macro that contains an INSERT request that inserts rows into a global temporary table, and that table is not materialized in the current session, then an instance of the global temporary table is created when the INSERT request performs.

When you perform a macro that contains a CREATE VOLATILE TABLE request, then the referenced table is created in the current session. If the macro includes a DROP on the named volatile table, then that table is dropped from the current session.

Macro Support for Large Objects

You can use BLOB and CLOB columns in macros as long as they do not violate the restrictions on large object use with SQL.

These restrictions are named in the following bulleted list:

- You cannot create a base table with more than 32 LOB columns, nor can you alter a base table to have more than 32 LOB columns.
- You can increase the maximum size of a LOB column up to the system limit, but you cannot decrease the maximum size of an existing LOB column.
- LOB columns can only carry the following attributes:
 - NULL
 - NOT NULL
 - TITLE
 - FORMAT
- LOB columns cannot be a component of any index.
- A base table containing LOB columns must have at least one non-LOB column to act as its primary index.
- The first column specified in a base table definition cannot be a LOB unless a uniqueness constraint is defined on some non-LOB column set in that table.
- The uniqueness of a SET base table must be defined on the basis its non-LOB column set because LOB columns cannot be used to specify row uniqueness.
- You cannot define integrity constraints on LOB columns, nor can you define integrity constraints that reference LOB columns.
- LOB columns can be components of a view subject to the restrictions provided in this list and the semantics of view definitions.
- LOB columns can be parameters of a user-defined function.

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

Macro Support for UDFs

You can invoke both external and SQL UDFs from any SQL request encapsulated within a macro definition as long as you follow the rules for invoking UDFs from the statement.

Macro Support for UDTs

Macros support both distinct and structured UDTs as well as Period data types, which are a special form of UDT.

Specifically, macros support the following UDT features:

- UDT parameters
- UDT expressions

Macro Support for Query Bands

Macro support for the SET QUERY_BAND statement is different depending on whether the Query Band is set for a session or a transaction:

- SET QUERY_BAND ... FOR SESSION is supported in the same way that other DDL statements are. You cannot set a session query band using a macro parameter. Otherwise, the system returns an error to the requestor.
- SET QUERY_BAND ... FOR TRANSACTION is not supported when it is the only statement in a macro. It is supported when there are other statements in the macro.

You can set a transaction query band using a macro parameter, including the QUESTION MARK parameter.

Macro Support for Query Bands in Proxy Connections

The following rules apply to macro support for query bands in proxy connections.

- Once a macro has been created, its immediate owner is its containing database or user, 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.
- The CONNECT THROUGH privilege for a SET_QUERYBAND with a PROXYUSER in a macro is validated against the trusted user when the macro is executed.

Macro Support for Workload Analysis Statements

The following workload analysis statements are not supported for macros. If you attempt to execute a macro that contains any of these statements, the system returns an error to the requestor.

- COLLECT DEMOGRAPHICS
- INITIATE INDEX ANALYSIS
- INITIATE PARTITION ANALYSIS
- RESTART INDEX ANALYSIS

Macros and Tactical Queries

Macros can be a very effective method of performing tactical queries. The pros and cons of using macros to perform tactical queries in various applications and application workloads are described in detail elsewhere:

- See *Teradata Vantage™ - Database Design*, B035-1094 for further information about using macros to perform tactical queries.
- See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146, for a comparison of the relative efficiencies of macros and stored procedures for performing tactical queries.

Function of REPLACE MACRO Requests

REPLACE MACRO executes as a DROP MACRO request followed by a CREATE MACRO request except for the handling of privileges. Vantage retains all of the privileges that were granted directly on the original macro.

If the specified macro does not exist, a REPLACE MACRO request creates it. In this case, the REPLACE request has the same effect as a CREATE MACRO request.

If an error occurs during the replacement of the macro, the existing macro remains in place as it was prior to the performance of the REPLACE MACRO request (it is not dropped). This is analogous to a ROLLBACK on the operation.

CREATE METHOD

Method Definitions Are Split Between CREATE METHOD And CREATE TYPE Definitions

Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

Unlike other SQL database objects, the definition for a method is split between CREATE METHOD and its signature as defined in its associated CREATE TYPE statement.

Although the definition for a method is generally similar to that of a UDF, unlike a UDF definition, the following components of a method are defined only in its signature:

- SPECIFIC NAME clause
- LANGUAGE clause
- PARAMETER STYLE clause
- [NOT] DETERMINISTIC clause
- SQL data access (NO SQL) clause

The RETURNS clause is defined both in the CREATE METHOD definition and in the CREATE TYPE signature for the method.

Dropping A Method

There is no DROP METHOD statement that you can use to drop a method body.

Instead, what you must do to drop a method is use the DROP METHOD or DROP SPECIFIC METHOD option of the ALTER TYPE statement (see [ALTER TYPE](#)) to drop the method signature from its type definition. This action also drops the external routine for the specified method.

You can also use the REPLACE METHOD statement (see [REPLACE METHOD](#)) to replace a method body.

Character Set Issues for Naming Methods

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 external procedure object even if the new object name contains only single-byte characters. Instead, use a multibyte session character set.

Naming Conventions: Avoiding Name Clashes Among UDFs, UDMs, and UDTs

For UDF, UDM, and UDT names:

- The DatabaseID, TVMName column pair must be unique within the DBC.TVM table. Use the DBC.Tables2 view to view rows from DBC.TVM.
- The signature of the *database_name.routine_name(parameter_list)* routine must be unique.

UDFs, UDMs, and UDTs can have the same SQL names as long as their SPECIFIC names and associated routine signatures are different. In the case of UDTs, the SPECIFIC name reference is to the SPECIFIC names of any method signatures within a UDT definition, not to the UDT itself, which does not have a SPECIFIC name.

The value of *database_name* is always SYSUDTLIB for UDFs associated with UDTs, including UDFs used to implement the following functionality on behalf of a UDT:

- Casts
- Orderings
- Transforms

The Database ID column entry is always the same. The name uniqueness is dependent on the TVMName column value only.

TVMName Entry for UDTs and UDMs

The following describes the TVMName entry for UDTs and UDMs.

UDTs created by a CREATE TYPE request have a SPECIFIC name that is system-generated based on the specified UDT name. For information on object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The database generates the SPECIFIC name by truncating any UDT names that exceed the permitted number of characters.

When you submit a CREATE TYPE statement, the system automatically generates a corresponding UDF to obtain a UDT instance.

The SPECIFIC name of a UDT, UDM or UDF must be unique to avoid name clashes.

| Type of External Routine | SPECIFIC Name |
|--|--|
| UDT | always its UDT name. |
| <ul style="list-style-type: none"> • UDM • UDF | user-defined. There are two exceptions to this rule for structured types: <ul style="list-style-type: none"> • System-generated observer methods. • System-generated mutator methods. The SPECIFIC names for these are always the same as the names of the structured UDT attributes on which they operate. Note that for UDMs, the SPECIFIC name is defined in its signature within its associated UDT, not within the CREATE METHOD statement. |

The signatures of external routines must be unique. The following rules apply:

- For every UDT you create, the system generates a UDF with the following signature: `SYSUDTLIB.UDT_Name()`.
No other UDF can have this signature.
- When you create a UDM, the system treats it as a UDF for which the data type of its first argument is the same as the UDT on which that UDM is defined.
For example, a UDM named `area()`, defined on the UDT named `circle`, would have the signature `SYSUDTLIB.area(circle)`. It follows that there can be no other UDF with this same signature.

For a single database (SYSUDTLIB) Teradata UDT environment, the following rules apply:

- A UDT and a SYSUDTLIB UDF with no parameters cannot have the same name.
- A method for a structured UDT cannot have the same name as any of the attributes for that type if the signature of the method and the signature of either the observer or mutator methods for the attribute match.

You must define a transform group for each UDT you create. Because the system creates a transform group for you automatically when you create a distinct UDT, you cannot create an additional explicit transform group without first dropping the system-generated transform. The names of UDT transform groups need not be unique, so you can use the same name for all transform groups.

The names of transform groups are stored in `DBC.UDTTransform`.

The system adds SPECIFIC method names to the `TVMName` column in `DBC.TVM` for:

- Observer and mutator methods, which are auto-generated for structured type attributes.
- Instance methods and constructor methods created by `CREATE TYPE`, `ALTER TYPE`, or `CREATE METHOD` statements where the coder does not specify a SPECIFIC method name.

A SPECIFIC name of up to 28 characters is generated based on the UDT and attribute names.

The SPECIFIC name is generated as the concatenation of the following elements in the order indicated:

1. The first 8 characters of the UDT name.
2. A LOW LINE (underscore) character.
3. The first 10 characters of the for observer attribute name, mutator attribute name, instance method name, or constructor method name, as appropriate.
4. A LOW LINE (underscore) character.
5. The last 8 HEXADECIMAL digits of the routine identifier assigned to the observer, mutator, instance, or constructor method name, as appropriate.
6. A character sequence is appended, `_O` for observer, `_M` for mutator, `_R` for instance, or `_C` for constructor, appropriate.

The remaining characters, up to the 30th character, are filled with SPACE characters.

Definitions And Purposes Of The Various Method Names

The table on the following page is intended to clarify the definitions and purposes of the various method name types in the following list:

- The method name, which is specified in both the CREATE TYPE statement that associates the method with the UDT and in the CREATE METHOD statement that defines the method body.
- The SPECIFIC method name, which is specified in the method signature declaration of the CREATE TYPE statement that associates the method with the UDT.
- The method object name, or external method name, which is specified in the EXTERNAL NAME clause of the CREATE METHOD statement that defines the method body.

| Method Name | Scenario 1 | Scenario 2 | Scenario 3 | Scenario 4 |
|--|--|--|--|--|
| <i>method_name</i> | Always specified. Used as SQL-invoked routine name. | Always specified. Used as SQL-invoked routine name. | Always specified. Used as SQL-invoked routine name. | Always specified. Used as SQL-invoked routine name. |
| SPECIFIC <i>method_name</i> | Specified. Used as the unique identifying SPECIFIC name for the method. | Specified. Used as the unique identifying SPECIFIC name for the method. | Not specified. System logic generates a default SPECIFIC method name and uses it as the unique identifying name for the method. | Not specified. System logic generates a default SPECIFIC method name and uses it as the unique identifying name for the method. |
| EXTERNAL NAME <i>external_method_name</i> | Specified. Used as the routine entry point for the method. | Not specified. The system assumes that the SPECIFIC method name is also the routine entry point for the method. | Not specified. The system assumes that the SPECIFIC method name is also the routine entry point for the method. The system-generated SPECIFIC method name in this scenario is also used as the routine entry point for the method. | Specified. Used as the routine entry point for the method. |

Function of Methods

A user-defined method, also referred to as a method or UDM, is a special kind of UDF that has the following properties:

- It is uniquely associated with a particular UDT.
- It is always created in the *SYSUDTLIB* database.
- It is invoked using dot notation, which is in the form of *udt_expression.method(...)* (see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210 for details).
- It can be written in either C or C++ (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details).

Methods must be defined both as part of the UDT definition as specified by its CREATE TYPE statement (see [Method Definitions Are Split Between CREATE METHOD And CREATE TYPE Definitions](#), [CREATE TYPE \(Distinct Form\)](#) and [CREATE TYPE \(Structured Form\)](#)) and as a separate method body definition using CREATE METHOD. The definition made by a CREATE METHOD statement actually declares the method, including its EXTERNAL NAME and EXTERNAL SECURITY definitions, if required.

You can add the signatures for newly defined methods to a type definition or drop the signature for an existing method using the ALTER TYPE statement (see [ALTER TYPE](#)).

Each method has an associated signature, defined as its method name and its declared parameter list. The method signature must match one of the method signatures that was specified in the CREATE TYPE statement of the specified UDT (see [CREATE TYPE \(Distinct Form\)](#) and [CREATE TYPE \(Structured Form\)](#)), or in a type redefinition made by an ALTER TYPE statement.

In other words, there must be a method signature associated with the UDT such that the following statements are all true:

- Its method name is the same.
- Its parameter list is the same.
- If the method is defined as a CONSTRUCTOR type, then the method signature also includes the CONSTRUCTOR keyword and its associated UDT must be a structured type.

The declared parameter list is that specified in the method specification of its associated CREATE TYPE statement. The system generates an augmented parameter list for each method internally. The augmented parameter list has the associated UDT name, or subject parameter, as its first parameter and the declared parameter list as its second parameter. Together, these form the augmented parameter list.

For example, consider the following method specifications for the UDT named *udt_name*:

```
CREATE TYPE udt_name
...
CONSTRUCTOR METHOD udt_name(CP1, CP2, ...) ...,
INSTANCE METHOD M1(MP1, MP2, ...) ...,
... ;
```

The following table describes the declared and augmented parameter lists for these methods:

| Methods | Declared Parameter Lists | Augmented Parameter Lists |
|-------------|--------------------------|---------------------------|
| Constructor | (CP1, CP2, ...) | (udt_name, CP1, CP2, ...) |
| Instance | (MP1, MP2, ...) | (udt_name, MP1, MP2, ...) |

The augmented parameter list is used to check for conflicts between methods specified in CREATE TYPE or ALTER TYPE statements and existing methods and UDFs. The augmented parameter list is also used in the method resolution process.

You can declare any data type, including LOBs, UDTs, TD_ANYTYPE, and VARIANT_TYPE for a method parameter.

Note:

You can only declare an input parameter to have the `VARIANT_TYPE` data type, but you can declare any method parameter to have the `TD_ANYTYPE` data type. For more information on developing methods, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Methods support both name overloading and overloading of server character sets or precision. Multiple methods of a UDT can have the same method name as long as their parameter lists are different. Overloading of character sets and precision is supported by using the `TD_ANYTYPE` data type for parameters and return types.

By defining a method using `TD_ANYTYPE` as a parameter data type, you can overload the method based on its server character set or numeric precision rather than its name. When you define a method using `TD_ANYTYPE`, the database determines the parameter data types at execution time based on the parameters that are provided.

In addition to allowing `TD_ANYTYPE` to act as an alias for the type for parameter data types with undetermined attributes, you can also use `TD_ANYTYPE` to resolve all possible parameter data types. This allows you to develop fewer signatures for the same basic method as well as to provide flexibility in coding the logic for the required method behavior.

However, using `TD_ANYTYPE` as a parameter data type results in the loss of the implicit conversions that automatically convert input values to those that match the method signature. As a result, you have a greater responsibility to make sure that any undefined parameters are properly resolved and processed.

Method Types

The database supports four types of methods, but only two of those can be created by a user: instance and constructor. The system does not support static methods.

| THIS type of method ... | Applies to this type of UDT ... |
|-------------------------|--|
| Instance | <ul style="list-style-type: none"> • Distinct • Structured Instance methods operate on a specific instance of a UDT. |
| Constructor | Structured. Constructor UDTs initialize an instance of a structured UDT. |

Instance methods operate on an instance of a UDT, while constructor methods initialize an instance of a UDT. Instance methods are the more common type. See [Instance Methods](#) and [Constructor Methods](#) for details.

The other two method types, observer and mutator, are specific types of instance method that the system generates automatically for each attribute of a structured UDT at the time the UDT is created. You cannot create either of these using `CREATE METHOD`. See [CREATE TYPE \(Structured Form\)](#) for details.

Instance Methods

Instance methods define an allowable operation on a structured UDT. An instance method does not operate directly on a UDT, but rather on a particular *instance* of that UDT.

When an instance method is invoked, a structured type instance is passed as an implicit subject parameter. This subject parameter is associated with the parameter name, SELF, and is the reason that you cannot name any method parameters using the name SELF.

Instance methods typically manipulate the data attribute set of a structured UDT. Examples of instance methods include observer and mutator methods.

Constructor Methods

Constructor methods initialize a structured UDT instance with input arguments, creating an instance value for the type. The ANSI SQL standard defines something called a *site*, which is little more than an umbrella name assigned to a large set of different kinds of variables. Incorporating this definition, a structured UDT is seen to determine a type that is the data type of a site. Again, this does little more than state that a structured UDT is the data type of an instance of a variable.

A constructor method, then, assigns specific values to the elements of a site according to the attributes of the structured UDT that is the data type for that site.

The specification of a constructor method must comply with the following rules:

- SELF AS RESULT must be specified as part of the method specification in the definition of the structured UDT (see [CREATE TYPE \(Structured Form\)](#)).
- The method name must be identical to the name of the structured data type being defined.
- The return data type specified for CONSTRUCTOR METHOD in the definition of the structured UDT must be identical to the data type for the corresponding attribute in the member list of the structured type definition.

Method Signatures

The signature of an instance or constructor method is the partial definition that you must provide when you create a UDT (see [CREATE TYPE \(Distinct Form\)](#) and [CREATE TYPE \(Structured Form\)](#)) or add an instance or constructor method to the UDT definition (see [ALTER TYPE](#)).

The method signature specified for a UDT definition does not *completely* define an instance or constructor method; however it *does* canonically define the following aspects of an instance or constructor method:

- The association between the specified method and the UDT being created with a CREATE TYPE statement.
- Its specific name, parameter style, SQL data access, and whether it is deterministic or not (see [Method Definitions Are Split Between CREATE METHOD And CREATE TYPE Definitions](#)). These specifications are all mandatory.

The method signature also defines, possibly redundantly, the data type it returns. Note that the RETURNS clause is mandatory for the method signature, but optional within the clauses included within CREATE METHOD. The definitions must be identical if both are specified.

A method signature must redundantly and identically specify the following list of items from its associated method body.

- Its method name.
- Its parameter list.
- The CONSTRUCTOR keyword if it is so defined in the method body of its associated CREATE METHOD statement.

You do *not* need to specify the INSTANCE keyword if its associated CREATE METHOD statement explicitly defines it as such because if no keyword is specified for a method, it is always assumed to be an instance method by default.

You must always create the method body using the CREATE METHOD statement.

Relationship Among Methods, UDFs, and External Procedures

External routines, the generic label used for UDFs, table UDFs, methods, and external procedures, are specific variations of one another and share most properties in common, including being written in a third-generation programming language such as C or C++.

See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#), [CREATE FUNCTION \(Table Form\)](#), and [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#) for more information.

When an external routine is invoked, the system passes a handle to the UDT argument instead of the UDT value. Given the handle, an external routine can get, or set, the value of a UDT argument by means of a set of library functions provided by Teradata (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147).

The ANSI SQL standard contrasts external routines with what it refers to as SQL routines, which are routines written in the SQL language. The SQL form of procedures (see [CREATE PROCEDURE and REPLACE PROCEDURE \(SQL Form\)](#)) is an example of an SQL routine.

When To Run Methods in Unprotected Mode

You should develop your methods in protected mode, which is the default when you create a new method. Protected and secure modes are states in which each instance of a method runs in a separate process. This is done to protect the system from many common programming errors such as non-valid pointers, corrupted stacks, and illegal computations such as divide-by-zero errors that would otherwise crash the database, produce problems with memory leakage, or cause other potentially damaging results.

The difference between a protected mode server and a secure mode server is that a protected mode server process runs under the predefined OS user *tdatuser*, while a secure server process runs under the OS user specified by the UDF in its EXTERNAL SECURITY clause. The two processes are otherwise identical.

These problems all cause the database to crash if they occur in unprotected mode. Methods can also cause the database to crash in protected and secure modes if they corrupt the shared data areas between the database and the protected or secure mode method.

Protected and secure modes are designed to be used for the following purposes only.

- Testing all methods that are in development.
- Running any methods that cause the OS to consume system resources.

This includes anything that causes the OS to allocate system context, including open files, pipes, semaphores, tokens, threads (processes), and so on.

If a method does *not* perform operations that cause the OS to consume system resources, then you should change its protection mode to EXECUTE NOT PROTECTED (see [ALTER METHOD](#)) after it has met your qualification standards for production use.

If a method *does* perform operations that cause the OS to consume system resources, then you should always run it in protected mode, even after it has been thoroughly debugged and meets all your production-level quality standards.

The following table summarizes this information for production-ready methods:

| IF a method ... | THEN you should run it in this mode ... |
|---|---|
| does not cause the OS to consume system resources | unprotected. |
| performs operations that cause the OS to consume system resources | <ul style="list-style-type: none"> • <i>protected</i> under the predefined OS user <i>tdatuser</i>. or • <i>secure</i> under either of the following users: <i>tdatuser</i> the OS user specified by the EXTERNAL SECURITY clause. |

The best practice is to develop and test your methods on a non-production test system. You should run any newly created method several hundred times, both to ensure that it does not crash the system and to determine any performance issues that could be avoided by alternate method design and better coding techniques.

You can use the `cufconfig` utility (see *Teradata Vantage™ - Database Utilities*, B035-1102 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147) to expand the number of protected or secure mode servers from the default value of 2 per AMP or PE to a maximum of 20 per AMP or PE vproc. The minimum is 0.

Protected mode servers consume disk resources as follows:

$$\text{Total swap space required} = \frac{\text{Number of vprocs}}{\text{node}} \times \frac{\text{Number of protected mode servers}}{\text{vproc}} \times \frac{256 \text{ KB}}{\text{server}}$$

In unprotected mode, a method is called directly by Vantage rather than running as a separate process. You should only alter a new method that does not require the OS to allocate system context to run in

unprotected mode after you have thoroughly tested and evaluated its robustness and performance impact. Once the newly created CPU-operations-only method has passed your quality measures and is ready to be put into production use, you should alter it to run in unprotected mode.

External Security Clause

This clause is mandatory for all methods that perform operating system I/O. Failing to specify this clause for a method that performs I/O can produce unpredictable results and even cause the database, if not the entire system, to reset. See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#).

Note that *authorization_name* is an optional Teradata extension to the ANSI SQL:2011 standard.

When a method definition specifies EXTERNAL SECURITY DEFINER, then that method executes:

- Under the OS user associated with the specified external authorization using the context of that user.

| IF the method runs in this mode ... | THEN the OS user must be ... |
|-------------------------------------|---|
| protected | <i>tdatudf</i> , which must be a member of the <i>tdatudf</i> OS group. |
| secure | an OS user assigned to an authorization name using the CREATE AUTHORIZATION statement. The specified OS user must belong to the <i>tdatudf</i> OS group. |

- The external security authorization associated with the method must be contained within the same database as the method.

The following rules apply.

- If you do not specify an authorization name, then you must create a default DEFINER authorization name before a user attempts to execute the method.
- If you have specified an authorization name, then an authorization object with that name must be created before the you can execute the method.

The system returns a warning message to the requestor when no authorization name exists at the time the method is being created.

CREATE ORDERING and REPLACE ORDERING

How REPLACE ORDERING Differs From CREATE ORDERING

The following rules and behaviors differentiate the REPLACE ORDERING and CREATE ORDERING statements:

- If you specify REPLACE ORDERING, and the specified ordering exists, the system replaces it with the new definition.
- If the specified ordering does not exist and the associated UDT does not have a defined ordering, then the system creates the specified ordering.
- If the REPLACE ORDERING statement specifies an ordering for a UDT with which an existing ordering is associated, the system returns an error to the requestor because you can specify only one ordering per UDT.

System Default Ordering Functionality for ARRAY and VARRAY Types

Teradata provides only basic ordering functionality for a newly created one-dimensional or multidimensional ARRAY/VARRAY type. This ordering functionality is provided to avoid hashing issues and to permit ARRAY/VARRAY types to be valid for columns in SET tables. The basic ordering functional that Teradata provides does not permit relational comparison of ARRAY/VARRAY values in any form.

Function of UDT Ordering Maps

A UDT ordering specifies how two values of the same UDT are to be compared using relational operators (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146). The system also uses the ordering definition for other comparison-related operations such as groupings, orderings, sorts, and uniqueness determinations.

An ordering is associated with a UDT by means of the CREATE ORDERING statement. You must define an ordering for all UDTs that you develop, and you must complete the definition before you can use it to define the data type of any table. If you attempt to use a UDT in any way without first creating an ordering for it, the system returns an error to the requestor.

There are a number of SQL operations that depend on an ordering definition to complete. For instance, a UDT must have an ordering to be used in any of the following operations.

- WHERE clause with comparison operators
- [NOT] BETWEEN ... AND
- [NOT] IN
- GROUP BY
- ORDER BY
- SELECT DISTINCT
- EXCEPT and MINUS set operators

- INTERSECT set operator
- UNION set operator
- COUNT (DISTINCT UDT_column)
- CREATE TABLE when the table contains a column typed with the UDT

| FOR this type of UDT ... | THE ordering functionality ... |
|--------------------------|--|
| distinct | <p>is generated automatically by the system.</p> <p>This is not true if the source data type is a LOB. In this case, you must explicitly create an ordering for the LOB using the CREATE ORDERING statement in conjunction with an appropriate external routine.</p> <p>If the system-generated ordering semantics is adequate, you need not define explicit ordering functionality using the CREATE ORDERING statement.</p> <p>If your applications require different or richer ordering semantics, then you can specify explicit ordering functionality using CREATE ORDERING.</p> |
| structured | must be defined explicitly using the CREATE ORDERING statement. |

Why Vantage Does Not Support The ANSI SQL STATE and RELATIVE Comparison Options

The ANSI SQL:2011 standard supports 3 comparison options for ordering definitions:

- MAP
- RELATIVE
- STATE

Vantage does not support the STATE and RELATIVE comparisons because they do not mesh well with the Teradata parallel architecture. The issue that makes these options problematic is that there is a strong relationship between the concepts of equality and the hash values that Vantage generates for many of its join strategies. Two values that are equal in the relational use of the concept must generate hash values that are also equal.

The MAP ordering approach enables Vantage to generate internally an appropriate hash value for the UDT, enabling all Optimizer join strategies to function properly.

Rules for Order Mapping UDFs

If you specify a UDF as the MAP routine for an ordering, it must obey the following rules:

- It must have one and only one parameter.
- The data type of the single parameter you specify must be the same as that of the UDT specified as *UDT_name*.
- The result data type must be a predefined data type.

Its predefined data type is valid except the following:

- BLOB
- CLOB
- It must be declared to be DETERMINISTIC.
- The name you specify for it in the ordering definition must be the name of a UDF contained within the *SYSUDTLIB* database.

See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#) for details about UDF parameters, result data types, and the DETERMINISTIC option.

Rules for Order Mapping Methods

If you specify a method as the MAP routine for an ordering, it must obey the following rules:

- It must be associated with the UDT for which this ordering is defined.
- Its declared parameter list must be empty.
- Its result data type must be a predefined data type.

Its predefined data type is valid except the following:

- BLOB
- CLOB
- It must be declared to be DETERMINISTIC.
- The name you specify for it in the ordering definition must be the name of a method contained within the *SYSUDTLIB* database.
- If you create a structured UDT that might be used for a hash index definition, you must ensure that its ordering function *never* returns a null. If it does, the system returns an error to the requestor.

See [CREATE METHOD](#) for details about method associations with UDTs, parameters, result data types, and the DETERMINISTIC option.

Recommendation for Handling Nulls

If you intend to support either of the following things with respect to UDT nulls, then you should specify a NOT NULL attribute for any column typed with that UDT:

- Support null UDT attributes.
- Code MAP ordering routines that can return nulls.

The reason for this recommendation probably is not obvious. The problem that must be dealt with is that if nulls are permitted in the column and either the MAP ordering routine or the system-generated observer routine can return nulls, the semantics are somewhat similar to the situation where you query a column that is specified with the NOT CASESPECIFIC attribute, where, for identical queries against static data, the results the system returns to the requestor can vary from request to request.

The ordering routine for a UDT determines both how comparisons of column values are made *and* the sort order for values having that user-defined data type.

If you do not follow this recommendation, then it is possible for a column null and a structured type containing null attributes whose MAP or observer routine returns NULL to be treated equally. This means that sometimes the system might return a column null in the result set and other times it might return the non-null structured UDT that contains null attributes in the result set.

Assuming that the structured UDT named *myStructUdtWithNullAttributes* was created with mapping and observer routines that can return nulls, and that the type supports null attributes (which can be inferred from its name), then a table defined along the lines of the following example, with all UDT columns specified with the NOT NULL attribute, cannot behave indeterminately with respect to nulls returned by a query against it:

```
CREATE TABLE udtTable (
  id          INTEGER,
  udtColumn   myStructUdtWithNullAttributes NOT NULL);
```

If you create a structured UDT that might be used for a hash index definition, you must ensure that its ordering function does not return a null. Otherwise, the database returns an error to the requestor.

High-Level Description of How MAP Routines Work

When you compare two values of a given UDT, the system invokes the MAP routine for each of them, then compares the returned values from the MAP routine invocations. Returned values can be specified to have any predefined data type except BLOB or CLOB, and the result of the UDT data type comparison is the same as the result of comparing the returned predefined type values.

The following notation represents the various types of variables used to specify the values to be compared and the external routines used to compare them:

- Let *x* and *y* represent the respective values of the UDT to be compared.
- The following table indicates the notation used to represent UDFs and methods:

| IF the MAP routine is this type of external routine ... | THEN use the following notation to represent it ... |
|---|---|
| user-defined method | M() |
| user-defined function | F |

The following table provides examples of the relationships between different values of the same UDT and the user-defined methods used to operate on them:

| IF the following method mappings evaluate to TRUE ... | THEN the result of <i>x comparison_operator y</i> is ... |
|---|--|
| <i>x.M()</i> = <i>Y.M()</i> | <i>x</i> = <i>Y</i> |
| <i>x.M()</i> < <i>Y.M()</i> | <i>x</i> < <i>Y</i> |
| <i>x.M()</i> <> <i>Y.M()</i> | <i>x</i> <> <i>Y</i> |

| IF the following method mappings evaluate to TRUE ... | THEN the result of <i>x comparison_operator y</i> is ... |
|---|--|
| $x.M() > Y.M()$ | $x > Y$ |
| $x.M() \leq Y.M()$ | $x \leq Y$ |
| $x.M() \geq Y.M()$ | $x \geq Y$ |

The following table provides examples of the relationships between different values of the same UDT and the user-defined functions used to operate on them:

| IF the following UDF mappings evaluate to TRUE ... | THEN the result of <i>x comparison_operator y</i> is ... |
|--|--|
| $F(x) = F(y)$ | $x = y$ |
| $F(x) < F(y)$ | $x < y$ |
| $F(x) <> F(y)$ | $x <> y$ |
| $F(x) > F(y)$ | $x > y$ |
| $F(x) \leq F(y)$ | $x \leq y$ |
| $F(x) \geq F(y)$ | $x \geq y$ |

Relationship Between Ordering And Collation Sequence For Character Data

If a mapping routine returns a character data type, the collation that is in effect is always the effective collation sequence at the time the UDT is accessed, not at the time that the ordering or UDT was created.

CREATE ORDERING/REPLACE ORDERING and System-Generated Constructor UDFs

Note that creating or replacing an ordering also causes the system-generated UDF constructor function for the UDT to be recompiled invisibly; invisible in that the system does not return any compilation messages unless the compilation fails for some reason, in which case the system returns an appropriate error message to the requestor.

CREATE PROCEDURE and REPLACE PROCEDURE (External Form)

Relationship Among UDFs, Table UDFs, and External Procedures

Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

External routines, the generic label used for UDFs, table UDFs, methods, and external procedures, are specific variations of one another and share most properties in common. See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#) and [CREATE FUNCTION \(Table Form\)](#).

Usage Restrictions for External Procedures

The following restrictions apply to external procedures.

- 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 external procedure object even if the new object name contains only single-byte characters. Instead, use a multibyte session character set.
- You cannot perform a CREATE PROCEDURE (External Form) request from an embedded SQL application.
- You can only execute SQL function calls using the CLIV2 or JDBC APIs.
- You cannot include CLIV2 or JDBC SQL function calls that execute the administration of row-level security constraints in any way in an external procedure definition.
- You cannot write Java strings into columns of a global temporary trace table (see [CREATE GLOBAL TEMPORARY TRACE TABLE](#)) having the following data types.
 - GRAPHIC
 - VARGRAPHIC
 - LONG GRAPHIC

Though you can define global temporary trace table columns with these types, the Java language does not support them, so you cannot write Java strings into them from a Java procedure.

- There are also several restrictions for external procedures that are invoked by triggers. See [Triggers and External Stored Procedures That Make SQL Calls](#) for details.

The following additional restrictions apply to Java external procedures only.

- You cannot access LOBs using inline processing with dynamic result sets.

- You cannot process result sets from other sessions or databases.
- Java procedures do not support pass back from call to call for dynamic result sets.

The results cannot be passed up a level at a time.

- You cannot mix non-result set and dynamic result set requests in a multistatement request.
- You cannot copy result sets.

Making a copy of a result set object does not return it twice.

- An SQL procedure cannot consume the dynamic results from a call to a Java procedure.

Memory Considerations for INOUT Parameters

If the size of an output value returned to an INOUT parameter is larger than the memory the system had allocated for the input value for that parameter when the procedure was called, the CALL request fails and returns an overflow error to the requestor.

The following example illustrates this. Suppose you have created a procedure named *myintz* with a single INOUT parameter.

```
CALL myintz(32767);
```

The smallest data type the system can use to store 32,767 is SMALLINT, so it allocates the 2 bytes required to store a SMALLINT number for the parameter. If this CALL request returns a value greater than or equal to 32,768 to the INOUT parameter, the system treats it as an overflow for a SMALLINT, irrespective of the data type assigned to the parameter when it was created, returns an error because the largest positive value that a SMALLINT variable can contain is 32,767.

See CALL in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Recompiling an External Procedure

Use the COMPILE option of the ALTER PROCEDURE (External Form) statement to recompile an existing external SQL procedure. See [ALTER PROCEDURE \(External Form\)](#).

When To Run External Procedures in Unprotected Mode

You should develop your external procedures in protected mode, which is the default when you create a new external procedure. Protected and secure modes are states in which each instance of a procedure runs in a separate process. The difference between a protected mode server and a secure mode server is that a protected mode server process runs under the predefined OS user *tdatuser*, while a secure server process runs under the OS user specified by the UDF in its EXTERNAL SECURITY clause. The 2 processes are otherwise identical.

This is done to protect the system from many common programming errors such as non-valid pointers, corrupted stacks, and illegal computations such as divide-by-zero errors that would otherwise crash the database, produce problems with memory leakage, or cause other potentially damaging results.

These problems all cause the database to crash if they occur in unprotected mode. External procedures can also cause the database to crash in protected and secure modes if they corrupt the shared data areas between the database and the protected or secure mode procedure.

Protected and secure modes are designed to be used for the following purposes only.

- Testing all external procedures that are in development.
- Running any external procedures that cause the OS to consume system resources.

This includes anything that causes the OS to allocate system context, including open files, pipes, semaphores, tokens, threads (processes), and so on.

- Running Java procedures.

If a procedure does *not* perform operations that cause the OS to consume system resources, then you should change its protection mode to EXECUTE NOT PROTECTED (see [ALTER PROCEDURE \(External Form\)](#)) after it has met your qualification standards for production use.

If a procedure *does* perform operations that cause the OS to consume system resources, then you should always run it in protected mode, even after it has been thoroughly debugged and meets all your production-level quality standards.

The following table summarizes this information for production-ready external procedures.

| IF an external procedure ... | THEN you should run it in this mode ... |
|---|---|
| does not cause the OS to consume system resources | unprotected. |
| performs operations that cause the OS to consume system resources | <ul style="list-style-type: none"> • <i>protected</i> under the predefined OS user <i>tdatuser</i>. or • <i>secure</i> under either of the following users. <i>tdatuser</i> the OS user specified by the EXTERNAL SECURITY clause. |
| is written in Java | protected. This is not optional. Procedures written in Java must <i>always</i> be run in protected mode. |

The best practice is to develop and test your external procedures on a non-production test system. You should run any newly created external procedure several hundred times, both to ensure that it does not crash the system and to determine any performance issues that could be avoided by alternate procedure design and better coding techniques.

You can use the `cufconfig` utility (see *Teradata Vantage™ - Database Utilities*, B035-1102 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147) to expand the number of protected or secure mode servers from the default value of 2 per AMP or PE to a maximum of 20 per AMP or PE vproc. The minimum is 0.

Protected mode servers consume disk resources as follows.

$$\text{Total swap space required} = \frac{\text{Number of vprocs}}{\text{node}} \times \frac{\text{Number of protected mode servers}}{\text{vproc}} \times \frac{256 \text{ KB}}{\text{server}}$$

In unprotected mode, an external procedure is called directly by the database rather than running as a separate process. You should only alter a new procedure that does not require the OS to allocate system context to run in unprotected mode after you have thoroughly tested and evaluated its robustness and performance impact. Once the newly created CPU-operations-only UDF has passed your quality measures and is ready to be put into production use, you should alter it to run in unprotected mode.

Each Java server for UDFs requires roughly 30 MB of memory for swap space, and there can be 2 such Java servers per node. A Java multithreaded server for non-secure mode Java procedures uses a minimum of an additional 30 MB (the amount required can be larger, depending on the size of the JARs for a user.), so each node requires approximately 100 MB of swap space if all server flavors are used.

Differences Between CREATE PROCEDURE and REPLACE PROCEDURE

You can create or replace an external procedure using the same syntax except for the keywords CREATE and REPLACE.

- If you specify CREATE, the procedure must not exist.
- If you specify REPLACE, you can either create a new procedure or replace an existing procedure.

When you replace an existing external procedure, the replaced procedure does not retain the EXECUTE NOT PROTECTED attribute if one had previously been set using the ALTER PROCEDURE statement (see [ALTER PROCEDURE \(External Form\)](#)).

The advantage to using REPLACE is that you can avoid having to grant the EXECUTE privilege again to all users who already had that privilege on the procedure.

Dynamic Result Sets

Dynamic result sets are supported for external procedures written in C or C++ using the CLv2 API and for Java external procedures using the JDBC API.

A procedure can return from 0 to 15, inclusive, result sets.

To create a result set to return to the caller or client, the external procedure must do these things:

- Submit the request to the database using the same session, which is the default connection, as the session for the external procedure.
- Submit a single statement containing a SELECT request.
- Specify a SPReturnResult of 1, 2, 3, or 4 in the Options parcel.
- Specify keep response or positioning.

| TO keep the result set until EndRequest is called so that parcels can then be fetched ... | Set keep_resp to this value ... |
|---|---------------------------------|
| either in random order by row or sequentially within a row using the DBCAREA settings Positioning-action, Positioning-statement-number, and Positioning-value | P |
| in sequential order, the Rewind function called, and then the parcels can again be fetched | Y |

When an external procedure executes, result sets created during its execution that match the conditions in the preceding bulleted list are returned to a calling procedure or client application of an external procedure if all of the following conditions are met:

- The external procedure did not send a Cancel parcel to the Dispatcher.
- The caller or client has indicated its willingness to receive the result sets by setting DynamicResultSetsAllowed to Y in the Options parcel for the request that called the procedure.
- The number of result sets to return is less than or equal to the *number_of_sets* value in the DYNAMIC RESULT SETS clause of the CREATE PROCEDURE or REPLACE PROCEDURE request.

If the external procedure *does* send a Cancel parcel to the Dispatcher, then the Dispatcher does not return the result set to the caller or client.

To match the functionality of the SCROLL and NO SCROLL cursors for SQL procedures, the set of rows returned to the caller or client depends on whether the SetPosition parcel, flavor 157, is sent with the SELECT request.

| IF the SetPosition parcel is ... | THEN the set of rows returned follows the rules for ... |
|----------------------------------|---|
| sent | SCROLL cursors. |
| not sent | NO SCROLL cursors. |

For CLv2 applications, the SetPosition parcel is sent when keep_resp = P.

The external procedure sends the ResultSetCurrentPosition parcel to the Dispatcher Continue mailbox to indicate the last row read prior to the external procedure completing. If the procedure never sends the ResultSetCurrentPosition parcel, then the platform assumes that while the entire response might have been sent, none of the rows have been read. This assumption is made even if the external procedure sends a Continue request to the database.

An external procedure can read a dynamic result set by submitting the CALL with the DynamicResultSetsAllowed field in the Options parcel set to the same session that invoked the external procedure.

External Procedures and Large Objects

Vantage does not support returning inline LOBs in an external procedure. If you need an external procedure to process inline LOBs, then place an SQL procedure wrapper around the external procedure.

To do this, use the following general procedure.

1. Create an SQL procedure that calls the external procedure and also returns an inline LOB.
2. Create an external procedure that returns the LOB AS LOCATOR to the SQL procedure created in the first step.

The SQL procedure then returns the inline LOB to the requestor.

Teradata Unity Support for External Procedures

Teradata Unity sends request-specific context information as part of a request that calls an external procedure to enable Vantage to change the result of the executed procedure indirectly by substituting a value predefined by Teradata Unity for a non-deterministic result. Vantage makes this context information available to an external procedure when it is called from the default connection for the session.

However, external procedures can generate and use their own arbitrary non-deterministic values that Vantage has no knowledge of. Therefore, Vantage cannot guarantee that an external procedure call produces a consistent result.

An external procedure can create a separate connection to another database, or to the same database by means of logging onto another session. In this case, the Teradata Unity-provided context information is not available for any SQL submitted using this type of connection.

Parameter Names and Data Types

The parameter list contains a list of variables to be passed to the external procedure. The list is bounded by open and closed parentheses even if there are no parameters to be passed.

Each parameter type is associated with a mandatory data type to define the type of the parameter passed to or returned by the external procedure. The specified data type can be any valid data type including TD_ANYTYPE and VARIANT_TYPE (see *Teradata Vantage™ - Data Types and Literals*, B035-1143 for a complete list of data types).

Note that you can only specify the VARIANT_TYPE data type for input parameters, but you can specify TD_ANYTYPE for all parameters. Character data types can also specify an associated CHARACTER SET clause.

By defining a procedure using TD_ANYTYPE as a parameter data type, you can overload the procedure based on its server character set or numeric precision rather than its name. When you define a procedure using TD_ANYTYPE, the database determines the parameter data type at execution time based on the parameters that are provided.

In addition to allowing TD_ANYTYPE to act as an alias for the type for parameter data types with undetermined attributes, you can also use TD_ANYTYPE to resolve all possible parameter data types. This allows you to develop fewer procedure signatures for the same basic procedure as well as to provide flexibility in coding the logic for the required procedure behavior.

However, using TD_ANYTYPE as a parameter data type results in the loss of the implicit conversions that automatically convert input values to those that match the procedure signature. As a result, you have a greater responsibility to make sure that any undefined parameters are properly resolved and processed.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about TD_ANYTYPE and see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for more information about how to code external procedures to take advantage of the TD_ANYTYPE data type.

You cannot specify a character parameter data type with a server character set of KANJI1. Otherwise, the database aborts the request and returns an error to the requestor.

| IF the external routine for the procedure is written in this language ... | THEN the maximum number of parameters you can specify in its parameter list is ... |
|---|--|
| <ul style="list-style-type: none"> • C • C++ | 256 |
| Java | 255 |

Parameter names are used by the COMMENT statement (see [COMMENT \(Comment Placing Form\)](#)) and are reported by the HELP PROCEDURE statement (see [HELP PROCEDURE](#)). Parameter names, with their associated database and procedure names, are also returned in the text of error messages when truncation or overflow errors occur with an external procedure call.

The following table summarizes the standard Teradata session mode semantics with respect to character string truncation.

| IF the session mode is ... | THEN ... |
|----------------------------|---|
| ANSI | any pad characters in the string are truncated silently and no truncation notification is returned to the requestor. A truncation exception is returned whenever non-pad characters are truncated. If there is a truncation exception, then the system does not call the procedure. |
| Teradata | the string is truncated silently and no truncation notification message is returned to the requestor. |

See [PARAMETER STYLE Clause](#) and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details on parameter passing conventions for the TD_GENERAL and SQL parameter styles.

See the documentation on the SQL Descriptor Area (SQLDA) in *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for a list of the data type encodings that procedure IN, INOUT, and OUT parameters can return to a client application.

LANGUAGE Clause

You can specify the LANGUAGE and SQL DATA ACCESS clauses in any order.

If you only have the object code for your external procedure body, you must ensure that it is completely compatible with C, C++, or Java object code, even if it was written using a different programming language.

SQL DATA ACCESS Options

| Option | Description |
|-------------------|---|
| CONTAINS SQL | <p>The procedure can execute SQL calls using CLIV2 or JDBC.</p> <p>The procedure neither reads nor modifies SQL data in the database.</p> <p>An example is a procedure whose body consists of just control statements local to the procedure.</p> <p>If such a procedure attempts to read or modify SQL data, or calls a procedure that attempts to read or modify SQL data, the system raises the following SQLSTATE exception code.</p> <pre>'2F004' - reading SQL-data not permitted</pre> |
| MODIFIES SQL DATA | <p>The procedure can execute all SQL calls that can validly be called from an SQL procedure using CLIV2 or JDBC.</p> <p>This is the default option for SQL procedures that do not specify an SQL Data Access clause when the procedure is defined.</p> <p>An example of such a statement is an UPDATE, INSERT or DELETE.</p> |
| <u>NO SQL</u> | <p>The procedure cannot execute SQL calls.</p> <p>This is the default option for external SQL procedures.</p> |
| READS SQL DATA | <p>The procedure cannot execute SQL calls using CLIV2 or JDBC that modify SQL data, but can make SQL calls that read SQL data.</p> <p>If such a procedure attempts to modify SQL data, or calls a procedure that modifies database data, the system raises the following SQLSTATE exception code.</p> <pre>'2F002' -modifying SQL-data not permitted.</pre> |

External Data Access Clause

This optional clause defines the relationship between a UDF or external procedure and data that is external to the database.

The option you specify determines:

- Whether or not the external routine can read or modify external data
- Whether or not the database will redrive the request involving the function or procedure after a database restart

If Redrive protection is enabled, the system preserves responses for completed SQL requests and resubmits uncompleted requests when there is a database restart. However, if the External Data Access

clause of an external routine is defined with the MODIFIES EXTERNAL DATA option, then the database will not redrive the request involving that function or procedure. For details about Redrive functionality, see:

- *Teradata Vantage™ - Database Administration*, B035-1093
- The RedriveProtection and RedriveDefaultParticipation DBS Control fields in *Teradata Vantage™ - Database Utilities*, B035-1102.

If you do not specify an External Data Access clause, the default is NO EXTERNAL DATA.

The following table explains the options for the External Data Access clause and how the database uses them for external routines.

| Option | Description |
|----------------------------|--|
| MODIFIES EXTERNAL DATA | The routine modifies data that is external to the database. In this case, the word <i>modify</i> includes delete, insert, and update operations. Note: Following a database failure, the database does not redrive requests involving a function or external stored procedure defined with this option. |
| <u>NO</u> EXTERNAL DATA | The routine does not access data that is external to the database. This is the default. |
| READS EXTERNAL DATA | The routine reads, but does not modify data that is external to the database. |

SQL SECURITY Privilege Options

The following table defines the meanings of the SQL SECURITY options.

| Privilege Option | Description |
|------------------|---|
| CREATOR | Assign the privileges of the creator of the procedure regardless of its containing database or user. |
| <u>DEFINER</u> | Assign the privileges of the definer of the procedure. This is the default. |
| INVOKER | Assign the privileges possessed by the user at the top of the current execution stack. |
| OWNER | Assign the privileges owned by the owner of the procedure, which are the privileges possessed by its containing database or user. |

External SQL Procedures and SQL

You can execute SQL function calls from external procedures using either of the following languages and APIs.

| FOR an external routine written in this language ... | You can use this API to execute SQL calls ... |
|--|---|
| <ul style="list-style-type: none"> • C • C++ | CLIV2 |
| Java | JDBC |

This does *not* mean that you can embed SQL statements within an external procedure, only that you can use standard CLIV2 or JDBC function calls to the appropriate driver, which then translates those calls into SQL requests and submits them to the Parser for processing.

Note:

The database does not support SQL function calls for mainframe-attached systems.

See *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418 or *Teradata JDBC Driver Reference*, available at <https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html>, for information about the available SQL function calls.

Restrictions on Declaring an C++ External Procedure

If you specify CPP in the Language Clause, then you must declare the called C++ procedure as extern “C” to ensure that the procedure name is not converted to an overloaded C++ name, for example.

```
extern “C”
void my_cpp(long int *input_int, long int *result, char sqlstate[6])
{
```

For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Rules for Creating a Java External Procedure

Java external procedures have 2 special requirements for creation or replacement that other types of external procedures do not.

- The user who creates a Java external procedure must be the same user who registered its associated JAR file by calling SQLJ.Install_Jar (see [JAR Files](#) and [SQLJ Database](#)). This means that the *jar_name* you specify when you create the procedure must refer to a JAR file that *you* registered. If any other user registered the JAR file you specify, the CREATE/REPLACE PROCEDURE request aborts and returns an error to the requestor.
- The containing database for any Java external procedure you create must match the default database at the time you register its JAR file using SQLJ.Install_Jar.

Details About the Function of Java External Procedures

When you write the code for an external procedure definition, you indicate that its language is Java by specifying LANGUAGE JAVA in the mandatory LANGUAGE clause (see [LANGUAGE Clause](#)).

Java external procedures execute SQL code using the standard JDBC driver interface. Because the Java external procedure is running on the database and is invoked from within a logged on session, its connection to the database is by means of a default connection named `jdbc:default:connection`.

Note that a Java program can create a separate connection to another database, or to the same database (by means of another session to be logged onto). If an external Java procedure does this, its actions can create an undetectable deadlock.

The ANSI SQL:2011 standard specifies the following set of rules for a Java external procedure.

- You must compile the Java source code outside the database.
The external Java language routine must be written using the JDBC standard for coding embedded SQL statements in a Java program.
The resulting class or classes (the byte code) must be placed in a JAR file.
- You must then register the JAR file with the database.
- To do this, you call an external procedure named `SQLJ.Install_Jar`.
- The system creates the Java external procedure using its EXTERNAL NAME clause, which specifies the registered JAR file and its associated Java class.

Once it has been created, you can access the Java routine in the same manner as any external procedure.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information on writing Java external procedures.

JAR Files

A JAR (Java ARchive) file is a collection of Java classes in a zip file format. The classes (byte compiled Java) are referenced when an external Java routine is created using the EXTERNAL NAME clause of CREATE PROCEDURE (External Form).

JAR files are always created outside of the database. Before you can reference a class, you must register it and copy it into the SQL environment. Once a JAR has been installed in the database, you cannot change its content in any way. You are restricted to either deleting a JAR file or replacing it in its entirety.

JAR files are available only to the user who installs them by means of a call to the `SQLJ.Install_Jar` external procedure (see [SQLJ Database](#) and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about this procedure). Analogous to the infrastructure for C and C++ external routines, the system creates a directory on the platform for each database that contains a JAR. Analogous to the fact that C and C++ DLLs created for 1 or more external routines in a given database cannot be accessed by other users or databases, so JAR files created for a Java external routine cannot be accessed by other users or databases.

There are no specific privileges for JAR files. Therefore, user or database *A* cannot create an external procedure that references a JAR installed in user or database *B*. However, user or database *A* can be granted access to a Java external procedure that has been created in user or database *B* by using the same privileges as are used for C and C++ external routines. See GRANT (SQL Form) in *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for information on how to do this.

One way to ensure such ready access to Java external procedures is to install all JAR files and create all Java external procedures in the same database and then grant access to them for all users who must execute them.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details.

Data Type Mapping Between SQL and Java

When you create or replace an external Java procedure, the system converts all SQL data types to or from its corresponding Java data type based on the type of parameter mapping.

External procedures default to Simple Mapping, where the SQL data types map to Java primitives or, when an appropriate primitive does not exist, to Java classes. In other words, if there is a Java type in the Simple Mapping column, the default mapping maps to that type. If the Simple Mapping column has a *not applicable* entry for an SQL data type, then the default mapping maps to the Java class in the Object Mapping column.

If you create or replace a Java external procedure that permits nulls to be passed to or from it, you cannot use simple mapping. To override the default mapping, you must specify the appropriate Java class from the Object Mapping column in the parameter list of the EXTERNAL NAME clause.

The following table defines the data type mapping between SQL and Java.

| SQL Data Type | Java Data Type | | |
|------------------|----------------|--------------------------------------|----------------------|
| | Simple Mapping | Simple Mapping Java Primitive Symbol | Object Mapping |
| BYTEINT | byte | B | java.lang.Byte |
| SMALLINT | short | S | java.lang.Short |
| INTEGER | int | I | java.lang.Integer |
| BIGINT | long | J | java.lang.Long |
| NUMERIC | not available | not available | java.math.BigDecimal |
| DECIMAL | not available | not available | java.math.BigDecimal |
| FLOAT | double | D | java.lang.Double |
| REAL | double | D | java.lang.Double |
| DOUBLE PRECISION | double | D | java.lang.Double |
| BYTE | not available | B | byte[] |
| VARBYTE | not available | B | byte[] |

| SQL Data Type | Java Data Type | | |
|--|----------------|--------------------------------------|--------------------|
| | Simple Mapping | Simple Mapping Java Primitive Symbol | Object Mapping |
| BLOB | not available | not available | java.sql.Blob |
| DATE | not available | not available | java.sql.Date |
| TIME TIME does not support the full nanosecond time granularity provided by the database. | not available | not available | java.sql.Time |
| TIMESTAMP | not available | not available | java.sql.Timestamp |
| INTERVAL | not available | not available | java.lang.String |
| CHARACTER | not available | not available | java.lang.String |
| VARCHAR | not available | not available | java.lang.String |
| CLOB | not available | not available | java.sql.Clob |
| GRAPHIC | not available | not available | not available |
| VARGRAPHIC | not available | not available | not available |
| UDT | not supported | not available | not supported |

SQLSTATE Values for Java External Procedures

A Java external procedure can raise a `java.lang.SQLException` code with an exception number and message that are returned to the database and used to set the SQLSTATE and error message. Java exceptions thrown during the execution of a Java external procedure using SQL are often handled within Java. When this occurs, the exceptions do not affect SQL processing. Only those Java exceptions that are not handled within Java are seen in the SQL environment as SQL exception conditions.

When this occurs, the procedure returns an exception number and message to the database which the system then uses to set the SQLSTATE value and error message that it returns to the requestor.

The range for valid SQLSTATE `SQLException` codes is 38U00 - 38U99, inclusive.

Some predefined error cases, such as non-valid JAR name, that have specific codes defined by the ANSI SQL standard, are also supported. The Teradata implementation of Java external procedures follows those definitions. For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 and *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

Java External Routine-Specific Dictionary Tables

The system dictionary has 3 tables that are used exclusively to keep track of the JAR files and Java external procedure that are defined on your system. Those tables, and their respective purposes, are summarized in the following table.

| Table Name | Purpose |
|-----------------------|--|
| DBC.Jar_Jar_Usage | <p>Contains 1 row for each JAR included in the SQL-Java path of another JAR. This is analogous to a <i>#include</i> directive in C and C++, where 1 JAR needs to include code from a different JAR. Therefore, if JAR A includes Jar B and Jar C, 1 row is inserted into <i>DBC.Jar_Jar_Usage</i> for the dependency of Jar A on Jar B, and 1 row is inserted for the dependency of Jar A on Jar C.</p> <p>You can only alter the SQL-Java path for a JAR by calling the external procedure <i>SQLJ.Alter_Java_Path()</i>, which means that a call to <i>SQLJ.Alter_Java_Path()</i> is the only action that can add rows to this table.</p> <p>For more information, see <i>Teradata Vantage™ - SQL External Routine Programming</i>, B035-1147.</p> |
| DBC.Jars | <p>Contains 1 row for each installed JAR.</p> <p>JARs can only be installed by means of a call to the <i>SQLJ.Install_Jar()</i> external procedure. Similarly, they can only be replaced by means of a call to the <i>SQLJ.Replace_Jar()</i> external procedure.</p> <p>For more information, see <i>Teradata Vantage™ - SQL External Routine Programming</i>, B035-1147.</p> |
| DBC.Routine_Jar_Usage | <p>Contains 1 row for each Java external procedure that names a JAR in its external Java reference string.</p> <p>The table indicates the dependency of a Java external procedure on a given JAR.</p> <p>For more information, see <i>Teradata Vantage™ - SQL External Routine Programming</i>, B035-1147.</p> |

Columns for handling Jars also exist in the *DBC.DBase*, *DBC.Dependency*, and *DBC.TVM* tables. See *Teradata Vantage™ - Data Dictionary*, B035-1092 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for the structure of these tables and more information about their usage.

SQLJ Database

The *SQLJ* system database contains 3 views defined by the ANSI SQL:2011 standard as well as five external procedures for handling JAR files. *SQLJ*, which is created by a DIP script, should be treated as if it were an extension to the system dictionary beneath the *DBC* database. You should never insert anything into, or delete anything from, the *SQLJ* database.

The DIP script that creates *SQLJ* and its components follows the pattern set by the Data Dictionary initialization process, which includes revoking all privileges that could result in modification of the views and external procedures contained within *SQLJ*.

The *SQLJ* database requires sufficient space for all of its required components, and its initial space allocation is determined based on that necessity.

The following 3 views are system-defined for reporting information about JAR files.

| View Name | Purpose |
|------------------------|--|
| SQLJ.Jar_Jar_Usage | Identifies each JAR owned by a given user or database on which other JARs defined on the system are dependent. |
| SQLJ.Jars | Identifies the installed JARs on the system that can be accessed by the current user or database. |
| SQLJ.Routine_Jar_Usage | Identifies the JARs owned by a given user or database on which external Java routines defined on the system are dependent. |

See *Teradata Vantage™ - Data Dictionary*, B035-1092 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details about the definitions and usage of these views.

The following five external procedures are system-defined for maintaining JAR files. You must have the EXECUTE privilege on any of the procedures to be able to execute it.

| Procedure Name | Purpose |
|-----------------------|---|
| SQLJ.Alter_Java_Path | Changes the search path for Java classes across installed JAR files. |
| SQLJ.Install_Jar | Registers a JAR file and its classes with the database. You must have either the CREATE EXTERNAL PROCEDURE or the CREATE FUNCTION privilege to be able to install JAR files. |
| SQLJ.Replace_Jar | Replaces an installed JAR file. You must have either the DROP PROCEDURE or the DROP FUNCTION privilege to be able to replace JAR files. |
| SQLJ.Remove_Jar | Removes a JAR file and its classes from the database. You must have either the DROP PROCEDURE or the DROP FUNCTION privilege to be able to remove JAR files. |
| SQLJ.Redistribute_Jar | Redistributes an installed JAR file to all nodes on the system. You must have either the DROP PROCEDURE or the DROP FUNCTION privilege on the current database to be able to redistribute JAR files. |

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details on the signatures and usage of these external procedures.

CREATE PROCEDURE Dictionary Table Actions for Java Procedures

The system performs the following actions on the Java external procedure-specific dictionary tables referenced by the *SQLJ* database during a CREATE PROCEDURE request.

1. The uniqueness of the Java procedure to be created is verified based on its name.

- The EXTERNAL NAME clause for the Java procedure is then examined to determine the JAR being referenced as its source.

When the referenced JAR reference has been found, the system searches *DBC.Jars* for the JAR name specified in the EXTERNAL NAME clause.

- The JAR name specified in the EXTERNAL NAME clause must be defined in the database for the CREATE PROCEDURE request to complete successfully.

If the system does not find the JAR name as an entry in *DBC.Jars*, the request aborts and returns an error to the requestor.

- Once the new Java procedure has been verified, the system adds a row to *DBC.Routine_Jar_Usage* to indicate that the new Java routine uses the JAR specified in its EXTERNAL NAME clause.

REPLACE PROCEDURE Dictionary Table Actions for Java Procedures

The system performs the following actions on the Java external procedure-specific dictionary tables referenced by the *SQLJ* database during a REPLACE PROCEDURE request.

- The uniqueness of the Java procedure to be replaced is verified based on its name.
- The EXTERNAL NAME clause for the Java procedure is then examined to determine the JAR being referenced as its source.

When the referenced JAR has been found, the system searches *DBC.Jars* for the JAR name specified in the EXTERNAL NAME clause.

- The JAR name specified in the EXTERNAL NAME clause must be defined in the database for the REPLACE PROCEDURE request to complete successfully.

If the system does not find the JAR name as an entry in *DBC.Jars*, the request aborts and returns an error to the requestor.

- Once the new Java procedure has been verified, the system takes 1 of 2 possible paths, depending on whether the named procedure is new or already exists.

| IF the procedure ... | THEN the system ... |
|----------------------|---|
| is new | adds a row to <i>DBC.Routine_Jar_Usage</i> to indicate that the new Java routine uses the JAR specified in its EXTERNAL NAME clause. |
| already exists | updates the row in <i>DBC.Routine_Jar_Usage</i> to indicate that the replacement Java routine uses the JAR specified in its EXTERNAL NAME clause. |

SQL DATA ACCESS Clause

The SQL Data Access clause indicates whether a procedure can issue SQL statements and, if so, what type. An external procedure that contains “SQL statements” actually contains CLIV2, ODBC, or JDBC calls for those statements, not the standard SQL statements themselves.

You can specify the SQL DATA ACCESS and LANGUAGE clauses in any order.

By supporting the execution of SQL calls from external procedures, the following things can all be done:

- You can write external procedures to perform SQL requests using CLIV2 or JDBC in the same way a client application can make CLIV2 calls to perform SQL requests.
- You can use the CREATE PROCEDURE (External Form) and REPLACE PROCEDURE (External Form) statements to specify a data access clause that defines the specific capabilities of SQL calls made from the external procedure.

When a CLIV2 external procedure specifies the MODIFIES SQL DATA option for its data access clause, for example, the system knows that the procedure must be linked with an API, but it does not know *which* API the procedure must be linked with. To handle this, the EXTERNAL NAME clause must explicitly specify that the procedure is to link with the CLIV2 libraries.

Explicitly specifying a package to link with in the EXTERNAL NAME clause is not valid for external procedures that use JDBC. For those procedures, the determination is made based on the combination of the LANGUAGE and SQL Data Access clauses you specify for the CREATE PROCEDURE or REPLACE PROCEDURE request.

Once it knows this information, the system takes care of finding the proper libraries, linking with them, and setting up the proper environmental variables during execution.

To avoid link and execution conflicts with existing UDF libraries, the database maintains a dynamic library that is separate from the standard UDF library for each database that contains CLIV2- or Java-based external procedures.

PARAMETER STYLE Clause

This optional clause specifies the parameter passing style to be used by the external procedure.

The SQL parameter passing style allows the code body to indicate NULL data. This cannot be done with the TD_GENERAL parameter passing style. If you do not specify a parameter style clause, then the system defaults to PARAMETER STYLE SQL. You must specify a parameter passing style of Java for all external routines written in Java. 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 (see [External Java Reference Strings](#)).

You can specify the parameter style clause in 1 of 2 places.

- SQL data access clause
- External procedure name

You cannot specify the parameter style both places in the same external procedure definition statement. Only 1 parameter style clause is permitted per external procedure definition.

Specific details of both options are described in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

External Body Reference Clause

The mandatory external body reference clause both declares that the external procedure is external to the database and identifies the location of all the file components it needs to be able to execute.

If an external procedure name alone is not sufficient to specify the location of all the procedure components, then you must also specify a string literal that explicitly specifies the path to each of those elements.

In all cases, *procedure_name* is the identifier you specify when you invoke the procedure from SQL requests.

You can optionally specify either or both of the following options for this clause.

- External procedure name
- Parameter style

You can specify an external procedure name in several different ways. See the following for details.

- [External String Literal](#)
- [Include Name Clause](#), [Library Name Clause](#), [Object File Name Clause](#), [Package Name Clause](#), and [Source File Name Clause](#)

For Java procedures, the external string literal specifies the JAR file, the Java class within that JAR, and the Java method within the class to be invoked when the system executes the external routine.

- The first part of the string literal specification is the JAR name or ID. This is the registered name of the associated JAR file. The name is created or specified with the external procedure *SQLJ.Install_Jar*.
- The second part of the string literal specification, which you specify after the colon, is the name of the Java class contained within the JAR file that contains the Java method to execute.
- The third part of the string literal specification is the name of the method that is executed.

For example, suppose you have a JAR file that has been registered in the database with the name *salesreports*, the class within the JAR is named *formal*, and the method within the class to be called is named *monthend*.

This location specification for the JAR, Java class, and the Java class method is completely specified by the following external string literal.

```
'salesreports:formal.monthend'
```

The Java method name must be unique within the database in which you are creating this Java external procedure.

You can also specify a Java parameter declaration list in the external body reference to specify the signature of a particular implementation of a Java method. This specification is mandatory when the system must choose among several possible implementations of the Java method. The Java parameters list specifies a

comma-separated list of Java data types (see [Data Type Mapping Between SQL and Java](#) and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147).

Note the following about Java data type names.

- They are case sensitive
- They can be fully qualified by a package name

The following table encapsulates the various specification possibilities of this clause for an external routine written in the C or C++ languages.

| IF CREATE PROCEDURE specifies this clause ... | THEN ... |
|---|---|
| EXTERNAL | <p>the C or C++ procedure name must match the name that follows the CREATE PROCEDURE keywords. Consider the following CREATE PROCEDURE statement.</p> <pre>CREATE PROCEDURE GetRegionXSP (INOUT region VARCHAR(64)) LANGUAGE C NO SQL EXTERNAL PARAMETER STYLE TD_GENERAL;</pre> <p>The C procedure name must be <i>GetRegionXSP</i>. If the client is mainframe-attached, then the C or C++ function name must be the DDNAME for the source.</p> |
| EXTERNAL NAME <i>external_procedure_name</i> | <p>the C or C++ procedure name must match <i>procedure_name</i>. Consider the following CREATE PROCEDURE statement.</p> <pre>CREATE PROCEDURE GetRegionXSP (INOUT region VARCHAR(64)) LANGUAGE C NO SQL EXTERNAL NAME xsp_getregion PARAMETER STYLE TD_GENERAL;</pre> <p>The C procedure name must be <i>xsp_getregion</i>. If the client is mainframe-attached, then <i>procedure_name</i> must be the DDNAME for the source.</p> |
| EXTERNAL NAME 'string' | <p>'string' can specify the C or C++ procedure name by stipulating the F option. For a Java external procedure, you must specify an external Java reference string rather than an EXTERNAL NAME 'string' value. You cannot specify the F option in <i>string</i> without also specifying an include, library, object, package, or source file name. The database needs 1 or more of these file names to link to. Consider the following CREATE PROCEDURE statement.</p> |

| IF CREATE PROCEDURE specifies this clause ... | THEN ... |
|---|---|
| | <pre>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;</pre> <p>The C function name must be <i>xsp_getregion</i>. If 'string' does not include the F option, then the C/C++/Java procedure name must match the procedure name specified with the CREATE PROCEDURE statement.</p> |

To adapt these examples for Java external procedures, you must specify a JAR reference string in place of the specification forms used for C and C++ external procedures.

For example, consider the following procedure definition for a Java external procedure.

```
CREATE PROCEDURE myclobdiagret2 (IN  b CLOB)
LANGUAGE JAVA
MODIFIES SQL DATA
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.myclobdiagret2';
```

The EXTERNAL NAME clause for this example specifies a JAR name, package name, and a Java class name, respectively, rather than an delimited function entry name, C or C++ library name, and so on. See [External Java Reference Strings](#) and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

You can specify the parameter style for the external procedure either in this clause or in the Optional Function Characteristics clause, but you can only specify the parameter style for an external procedure 1 time in its definition. For more information, see [Parameter Style Clause](#).

EXTERNAL NAME Clause

You use the External Name clause to specify the locations of all components needed to create the external routine.

All files included by, or linked to, an external routine must be accessible to the database. These files can reside on a client system or within the database; however, for security reasons, the best practice is to keep the source on the client system to ensure that the programmer writing the external routine can control who can access the code. To do this, specify the C option in the external string literal clause to notify the system that the source is stored on the client.

If security is not an issue, you can store the source code files on the server. There are specific default server directories for that purpose.

The system automatically copies any client-resident source, header, or object files specified in the External Name clause to the server. Note that the system does *not* transport external libraries and shared object files, which must be manually installed in the database.

You can specify the external function name in one of three ways:

- As an external routine name identifier with optional Parameter Style clause.
- As an external routine object name.
- As a coded string that specifies the explicit path to the specified code entity.

| IF the external routine name is . .. | THEN ... |
|---|--|
| an identifier | it is the name of the entry point for the external routine object. The identifier is case sensitive and must match the C or C++ external routine name. |
| a string | it is composed of one of the following: <ul style="list-style-type: none"> ◦ a C or C++ external routine entry point name specification ◦ an encoded path to the specified code entity For a Java external routine, you must specify an external Java reference string. The maximum length of the external name string is 1,000 characters. |

You can specify more than one encoded string per external routine definition, though some can only be specified once within a list of strings.

External String Literal

The external string literal specifies an encoded list of the components required to create an external procedure. When decoded, the string specifies what each component is and where it is located.

Depending on the requirements of the specific external procedure, you can specify the following components.

- Include file (C or C++ header file) paths (see [Include Name Clause](#))
- System library paths (see [Library Name Clause](#))
- Object code paths (see [Object File Name Clause](#))
- Package (user-created library) paths (see [Package Name Clause](#))
- Source code paths (see [Source File Name Clause](#))
- External Java reference paths (see [External Java Reference Strings](#))

Vantage uses the code path to access the file it specifies, then adds an appropriate extension as it transfers that file to the external procedure compile directory on the Teradata platform.

It is good programming practice not to use any of the following characters as the delimiter because they might be interpreted as a component of a path string rather than as a delimiter.

- / (SOLIDUS)
- \ (REVERSE SOLIDUS)
- : (COLON)

The only restriction enforced on the arbitrary delimiter character is that you must use the same delimiter character throughout the specified path string.

Vantage retrieves the source component from the specified client or server when it requests the source or object code.

You should name any include files (C or C++ header files) with the same name as specified in the include directive of the source file (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147).

For example for a z/OS client,

```
'CI|udfdefs|UDFDEFS|CS|myudf|UDFSRC|F|theudf'
```

where *UDFDEFS* and *UDFSRC* are the DDNAMEs for the files on the z/OS client system. Whenever you refer to an IBM client file in a code path, you must identify it by its client DDNAME.

This clause causes the system to do the following things.

- Retrieve the file *UDFDEFS* and rename it to *udfdefs.h* in the UDF compile directory for the server.
- Retrieve *UDFSRC* and rename it to *myudf.c* in the UDF compile directory for the server.

The C source *myudf.c* Should contain the following code.

```
#include <udfdefs.h>
```

or

```
#include "udfdefs.h"
```

Suppose you specify the following skeletal CREATE PROCEDURE (External Form) statement, where the omitted details are denoted by an ELLIPSIS (...) character.

```
CREATE PROCEDURE abc(...)
...
EXTERNAL NAME 'CS!matrix!matrix.c';
```

The name of the C source file in this example is *matrix.c* and the C procedure name, based on the procedure name *abc*, must have the following skeletal code.

```
void abc(...)
{
```

```
...
}
```

Suppose you specify the following skeletal CREATE PROCEDURE (External Form) statement, where, again, the omitted details are denoted by an ellipsis (...).

```
CREATE PROCEDURE abc(...)
...
EXTERNAL NAME 'CO!matrix!matrix.o';
```

There is no source file for this procedure; instead, the code is an object file named `matrix.o`.

For this CREATE PROCEDURE (External Form) statement to compile successfully, there must be a procedure in the specified object file named `abc`.

See the table at the end of the [External String Literal Examples](#) topic for more information about procedure name usage.

You can specify the various options as many times as needed except for the package option, which cannot be used in combination with any of the other options.

The maximum length for the external name string is 1,999 characters.

External Java Reference Strings

When you create an external procedure written in Java, the EXTERNAL NAME clause specification is slightly different than what you would specify for an external procedure written in C or C++. For Java external procedures, you must specify an external Java reference string that specifies the JAR file name, Java class name within the JAR, and the Java method name within the class to be invoked when the system executes the procedure in the following format.

```
'jar_name:[package_name.]java_class_name.java_method_name [java_data_type [,...]]'
```

jar_name

The registered name of the JAR file associated with the procedure that the system creates when you call the built-in external procedure `SQLJ.Install_Jar`.

package_name

[Required if *java_class_name* and *java_method_name* are in external package, optional otherwise.] The name and path of the external method package that contains *java_class_name* and *java_method_name*.

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.

java_method_name

The name of the Java method that is performed when this procedure executes.

java_data_type

[Required if *java_method_name* is overloaded, optional otherwise.] Java data type of *java_method_name*. Specifies the signature of a particular overloaded implementation of a Java method.

A Java data type can be either simple or object-mapped according to the parameter mapping rules described in [Data Type Mapping Between SQL and Java](#).

java_data_type names are case sensitive.

Client-Server External Procedure Code Specification

You must specify whether the external procedure code for include files, object files, and source files is located on the client system or on the Teradata platform system. To do this, you specify C for client or S for server.

External procedure code for library and package files is always located on the Teradata platform, but you still must specify the S location code for any library or package file paths you specify.

The character used to separate entities in the path specification is platform-specific when the file is stored on a client system, but not when the file is on the Teradata platform.

The following table provides more information about writing the client and server location specifications for include, object, or source files.

| IF you specify this location code ... | THEN you ... |
|---------------------------------------|--|
| C | must format the specification in the form required by the client application, for example, BTEQ. Refer to the appropriate client documentation for information about the required form of presentation. |
| S | can use either the SOLIDUS character (/) or the REVERSE SOLIDUS character (\) as the separator in the path specification for all platform operating systems. |

Include Name Clause

The Include Name clause specifies an explicit path to an include file that is to be used for this external procedure definition.

The syntax for this clause is as follows.

```
CI;name_on_server;include_name
```

or

```
SI;name_on_server;include_name
```

The character `i` represents a user-defined delimiter.

Perform the following procedure to specify an include file name string.

1. Begin the clause with the appropriate client or server location code.

| IF you specify this code ... | THEN the source or object code for the external procedure is stored on the ... |
|------------------------------|--|
| C | client. |
| S | server. |

2. Type the character `I` to indicate this is an include file specification.
3. Specify an arbitrary delimiter character to separate the `I` code and the `name_on_server` variable specified in the string.
4. Specify the name assigned to the include file, without the `.h` extension, on the server. The server adds the `.h` extension.

All *names on server* must be unique among the UDFs, table UDFs, and external procedures created within the same database. If the CREATE/REPLACE FUNCTION definition includes a nonunique *name_on_server* specification, the system does not create it.

The C source must have an include statement in the following form.

```
#include <name_on_server.h>
```

5. Specify your delimiter character to separate the *name_on_server* from the *include_name*.
6. Specify the path and name of the include file.

| IF the include file is on the ... | THEN you ... |
|-----------------------------------|--|
| client | must format the specification in the form required by the client application, for example, BTEQ. Refer to the appropriate client documentation for information about the required form of presentation. |
| server | can use either the SOLIDUS character (/) or the REVERSE SOLIDUS character (\) as the separator in the path specification for all platform operating systems. |

Library Name Clause

The Library Name clause specifies the name of a non-standard library file on the server that would not normally be linked with the external procedure being defined.

The syntax for this clause is as follows.

```
SL|library_name
```

The character | represents a user-defined delimiter.

Perform the following procedure for specifying a library file name string.

1. Begin the clause with the character S to indicate this is a server specification.
2. Type the character L to indicate this is a library file specification.
3. Specify an arbitrary delimiter character to separate the L code and the library name specified in the string.
4. Specify the name assigned to the non-standard library file on the server. The server automatically adds prefix or suffix values as needed.

The path must name a server library that is already installed on the system.

You can use either \ or / characters to specify the path for all operating systems.

Object File Name Clause

The Object File Name clause specifies an explicit path to an object file that is to be used for this external procedure definition.

The syntax for this clause is either of the following.

- CO|name_on_server|object_name
- SO|name_on_server|object_name

The character | represents a user-defined delimiter.

Perform the following procedure for specifying an object file name string.

1. Begin the clause with the appropriate client or server location code.

| IF you specify this code ... | THEN the source or object code for the external procedure is stored on the ... |
|------------------------------|--|
| C | client. |
| S | server. |

2. Type the character O to indicate this is an object file specification.
3. Specify an arbitrary delimiter character to separate the O code and the *name_on_server* variable specified in the string.
4. Specify the name assigned to the object file on the server. Do not specify an extension for this file name.

All *names on server* must be unique among the UDFs, table UDFs, and external procedures created within the same database. If the CREATE/REPLACE PROCEDURE definition includes a nonunique *name_on_server* specification, the system does not create it.

5. Specify your delimiter character to separate the *name_on_server* from the *object_name*.
6. Specify the path and name of the object file.

| IF the object file is on the ... | THEN you ... |
|-------------------------------------|--|
| client | must format the specification in the form required by the client application, for example, BTEQ. Refer to the appropriate client documentation for information about the required form of presentation. |
| server | can use either the SOLIDUS character (/) or the REVERSE SOLIDUS character (\) as the separator in the path specification for all platform operating systems. |

Package Name Clause

The Package Name clause specifies an explicit path to a package file that is to be used for this external procedure definition. Packages are libraries that can contain external procedure C or C++ functions and other functions to be called by an external procedure.

A typical package includes a function library as well as a script that contains all the necessary SQL DDL and DCL statements to create the procedures and make them available to users.

A package is a shared object file with a .so extension for Linux systems.

Package files must be distributed to all server nodes.

You cannot specify the package option with any other encoded path string clause, but you can specify it with a procedure entry point name string (see [EXTERNAL NAME Clause](#)).

To distribute a third-party package to a database node, use their documented installation procedure. If you are installing a package developed by your programming staff, you can use either of the following general procedures.

- Install the package in a specific directory and then use PCL to distribute it to all nodes.
- FTP the package from a client system.

The syntax for this clause is as follows.

```
SP ; package_name
```

The character ; represents a user-defined delimiter.

If the package supports SQL calls, then you must specify the name of the API that is to handle those calls in addition to specifying an appropriate SQL Data Access option.

See [Specifying the CLI Option for the EXTERNAL NAME Package Clause](#) and [SQL DATA ACCESS Clause](#) for details.

Perform the following procedure for specifying a package file name string.

1. Begin the clause with the character S to indicate this is a server specification.
2. Type the character P to indicate this is a package file specification.
3. Specify an arbitrary delimiter character to separate the P code and the package name specified in the string.
4. Specify the path and name assigned to the package file on the server.

The package file extension must be .so for Linux systems.

The path must name a package or API that is already installed on the system and that has been distributed to all its nodes. Its maximum length is 256 characters.

All *names on server* must be unique among the UDFs, table UDFs, and external procedures created within the same database. If the CREATE/REPLACE PROCEDURE definition includes a nonunique *name_on_server* specification, the system does not create it.

You can use either \ or / characters to specify the path for all platforms.

Specifying the CLI Option for the EXTERNAL NAME Package Clause

If an external procedure supports CLIV2 SQL calls, you must indicate that it does so when you specify the package name for the procedure.

| IF the external procedure supports the use of SQL calls using this API ... | THEN you must specify <i>package_name</i> as ... |
|--|--|
| CLIV2 | CLI |

The following rules apply to specifying the CLI package option.

- You can only specify the CLI package name for external procedures.
You *cannot* specify this option when you create UDFs or methods.
- You can specify the CLI package name only once in an EXTERNAL NAME clause.
You can also specify other user-defined packages, but not another CLI or other Teradata package option.
- You must specify this option for each external procedure that uses CLIV2 calls to perform SQL.
- The system links all CLIV2-based external procedures created in 1 database to the same dynamic library in that database, but they are strictly segregated from the standard external procedure and UDF dynamic libraries in the same database.
- You must run all CLIV2-based external procedures in EXECUTE PROTECTED mode only.

CLlV2-based external procedures *cannot* run in EXECUTE NOT PROTECTED mode, and you cannot use the ALTER PROCEDURE (External Form) statement (see [ALTER PROCEDURE \(External Form\)](#)) to change the protection mode of such a procedure to EXECUTE NOT PROTECTED.

Source File Name Clause

The Source File Name clause specifies the location of a source file that is to be used for this external procedure definition.

The syntax for this clause is either of the following.

- `CSiname_on_serverisource_name`
- `SSiname_on_serverisource_name`

The character *i* represents a user-defined delimiter.

Perform the following procedure for specifying a source file name string.

1. Begin the clause with the appropriate client or server location code.

| IF you specify this code ... | THEN the source or object code for the external procedure is stored on the ... |
|------------------------------|--|
| C | client. |
| S | server. |

2. Type the character S to indicate this is a source file specification.
3. Specify an arbitrary delimiter character to separate the S code and the *name_on_server* variable specified in the string.
4. Specify the name assigned to the source file on the server.

All *names on server* must be unique among the UDFs, table UDFs, and external procedures created within the same database. If the CREATE/REPLACE PROCEDURE definition includes a nonunique *name_on_server* specification, the system does not create it.

5. Specify your delimiter character to separate the *name_on_server* from the *source_name*.
6. Specify the path and name of the source file.

You can use either \ or / characters to specify the path for all platforms.

External String Literal Examples

The following examples demonstrate various external procedure external string literals.

Example 1: External String Literal Examples

The following example indicates that the source is to be obtained from the client, from absolute directory procedure_source. The file name on the server and the source file name are both sales.c. The C procedure name is sales1.

```
'CS|sales|C:\procedure_source\sales.c|F|sales1'
```

Example 2: External String Literal Examples

In the following example, the object is to be obtained from the lowest node on the server in directory /home /james/spdev/imagef.o. The file name on the server is img.o. The name of the object it retrieves is imagef.o.

```
'S0|img|/home/james/spdev/imagef.o|F|img_match'
```

Example 3: External String Literal Examples

The following example indicates that the header file sp_types.h and the C source file stdxml.c to be used for the procedure are to be found on the server.

/home/jcx/headers/sp_types.h is the relative path from the home or current client directory for the logged on user to the header file and home/jcx/src/stdxml.c is the relative path to the C source file. The procedure name in the C source code is called stdxml. Both files have the same name on the server.

```
'SI|sp_types|/home/jcx/headers/sp_types.h|SS:stdxml|home/jcx/src/  
stdxml.c|F|stdxml'
```

The following table summarizes the naming issues for the EXTERNAL NAME clause and its various components.

| Procedure Name | Source File Name | Procedure Name in DBC. TVM | C/C++ Procedure Name | Comments |
|---|---|----------------------------|---|---|
| <i>procedure_name only</i> | <i>procedure_name</i> | <i>procedure_name</i> | <i>procedure_name</i> | <i>procedure_name</i> must be unique within its database. |
| <i>procedure_name and external_procedure_name but not as 'string'</i> | <i>external_procedure_name</i> | <i>procedure_name</i> | <i>procedure_name</i> if not in 'string' or <i>external_procedure_name</i> if in 'string' | |
| <i>procedure_name and external_procedure_name as 'string'</i> | <i>source_name</i> as specified in 'string' | <i>procedure_name</i> | <i>procedure_name</i> if not in 'string' or <i>external_procedure_name</i> if in 'string' | |

External Procedure Default Location Paths

External procedures expect information to be in certain default locations when they are created or performed.

9: CREATE PROCEDURE and REPLACE PROCEDURE (External Form)

The following list catalogs the default path locations used by external procedures for the following purposes:

- Store procedure source files
- Compile procedure source files
- Store .so or JAR files
- Store shared memory files

The following table identifies the default directory paths for these resources and activities.

| File/Directories | Linux | Description |
|-------------------------|---|---|
| Header file | /etc/opt/teradata/tdconfig/ Teradata/tdbs_udf/usr/ | The header file <code>sqltypes_td.h</code> must be specified with an include directive in the procedure source. You can copy this file if you code or compile the procedure outside of the database. |
| Source directory path | /etc/opt/teradata/tdconfig/ Teradata/tdbs_udf/usr/ | The default directory to search for source files. If the source or object file is on the Teradata platform in this directory, then you can specify the relative path from this directory for any server components specified in the external name string. This applies to all the following file types. <ul style="list-style-type: none">• Include• Object This category includes JAR files.• Package• Source |
| Compiler path | /usr/bin/gcc | |
| Linker path | /usr/bin/ld | |
| Compiler temporary path | /var/opt/teradata/tdtemp/ UDFTemp/ | Temporary directory where external procedures are compiled Any files needed for the compilation process are moved here. This includes source files from the server or client as well as object and header files, if needed. Temporary compilation directories only exist during the duration of a compilation. |
| UDF library path | /etc/opt/teradata/tdconfig/ udflib/ | Read-only directory where dynamically linked libraries are stored. |
| UDF server memory path | /var/opt/teradata/tdtemp/ udfsrv/ | Directory where shared memory files used for the execution of protected mode procedures are stored. |
| JAR library path | /etc/opt/teradata/tdconfig/ jarlib/ | Directory where JAR file libraries are stored. |

External Procedure and UDF .so Linkage Information

There is only 1 .so file per database per node per application category, which is shared by all external procedures, UDFs, and table UDFs of a given application category contained by a database.

The application categories are as follows.

- Standard
 - Applies to all UDFs and table UDFs.
 - Also applies to external procedures that do not use the CLIV2 or Java APIs to make SQL calls.
- CLIV2
 - Applies to all external procedures that use the CLIV2 API to make SQL calls.
- Java
 - Applies to all external procedures that use the JDBC API to make SQL calls.

The system generates a separate .so file for each non-Java application category. Java UDFs and external procedures do not generate .so files because their source code is contained within their associated JAR files.

Whenever you create, replace, or drop an external procedure, the .so file for that database has to be relinked with all the other external procedure object files and then redistributed to all the nodes.

EXTERNAL SECURITY Clause

This clause is mandatory for all external procedures that perform operating system I/O. Failing to specify this clause for a procedure that performs I/O can produce unpredictable results and even cause the database, if not the entire system, to reset.

Note that *authorization_name* is an optional Teradata extension to the ANSI SQL:2011 standard.

When a procedure definition specifies EXTERNAL SECURITY DEFINER, then that procedure executes in one of the following environments.

- Specified OS platform user context of the associated security statement created for that purpose.
- Same database or user in which the procedure is defined.

The following rules apply.

- If you do not specify an authorization name, then you must create a default DEFINER authorization name (see [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#)) before a user attempts to execute the procedure.
- If you have specified an authorization name, then an object with that name must be created before the you can execute the procedure.

The system returns a warning message to the requestor when no authorization name exists at the time the procedure is being created.

Related Information

| For more information ... | See ... |
|--|---|
| on how to specify more than 1 encoded string per external routine definition | the following topics. <ul style="list-style-type: none"> • Include Name Clause • Library Name Clause • Object File Name Clause • Package Name Clause • Source File Name Clause |
| on how to specify the C option in the external string literal clause | External String Literal |
| on default platform directories | UDF Default Location Paths |
| on how to specify external function names | the following topics. <ul style="list-style-type: none"> • Parameter Style Clause • Function Entry Name Clause • External String Literal |

CREATE PROCEDURE and REPLACE PROCEDURE (SQL Form)

SQL Data Access Options

| Option | Description |
|-------------------|---|
| CONTAINS SQL | <p>The procedure can execute SQL control statements.</p> <p>The procedure neither reads nor modifies SQL data in the database.</p> <p>An example is a procedure whose body consists of just control statements local to the procedure.</p> <p>If such a procedure attempts to read or modify SQL data, or calls a procedure that attempts to read or modify SQL data, the system raises the following SQLSTATE exception code.</p> <pre>'2F004' - reading SQL-data not permitted.</pre> |
| MODIFIES SQL DATA | <p>The procedure can execute all SQL statements that can validly be called from an SQL procedure.</p> <p>This is the default option for SQL procedures that do not specify an SQL Data Access clause when the procedure is defined.</p> <p>An example of such a statement is an UPDATE, INSERT or DELETE.</p> |
| READS SQL DATA | <p>The procedure cannot execute SQL statements that modify SQL data, but can execute statements that read SQL data.</p> <p>An example is the FETCH statement.</p> <p>If such a procedure attempts to modify SQL data, or calls a procedure that modifies database data, the system raises the following SQLSTATE exception code.</p> <pre>'2F002' -modifying SQL-data not permitted.</pre> |

SQL SECURITY Privilege Options

| Privilege Option | Description |
|------------------|---|
| CREATOR | Assign the privileges of the creator of the procedure regardless of its containing database or user. |
| <u>DEFINER</u> | Assign the privileges of the definer of the procedure. This is the default privilege option. |
| 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. |

Invocation Restrictions

You cannot specify either CREATE PROCEDURE or REPLACE PROCEDURE requests as part of a procedure definition.

Default Session Collation For Procedures

The default server character set for a procedure is set by Teradata support. If you want to change this default, contact Teradata support.

The following bullets list the possible default session collations for a procedure.

- The default session collation is that of the user who *executes* the procedure.
This is the standard default.
- The default session collation is that of the user who *created* the procedure.

This means that result sets returned to either a client application or a calling procedure use the collation established for the called procedure even though they return their results using the character set of the client application or procedure.

Memory Considerations for INOUT Parameters

If the size of an output value returned to an INOUT parameter is larger than the memory the system had allocated for the input value for that parameter when the procedure was called, the CALL request fails and returns an overflow error to the requestor.

The following example illustrates this. Suppose you have created a procedure named *myintz* with a single INOUT parameter.

```
CALL myintz(32767);
```

The smallest data type the system can use to store 32,767 is SMALLINT, so it allocates the 2 bytes required to store a SMALLINT number for the parameter. If this CALL request returns a value greater than or equal to 32,768, the system treats it as an overflow for a SMALLINT *irrespective of the data type assigned to the parameter when the procedure was created*, aborts the request, and returns an error because the largest positive value that a SMALLINT variable can contain is 32,767.

See CALL in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details and further examples.

Recompiling an SQL Procedure

Use the ALTER PROCEDURE (SQL Form) statement to recompile an existing SQL procedure (see [ALTER PROCEDURE \(SQL Form\)](#)).

Special Condition Handling for SIGNAL and RESIGNAL Statements

The condition names defined in a condition declaration can be specified in a condition handler declaration for two purposes:

- To document the program by allowing you to associate a descriptive symbolic name for a specific SQLSTATE value.
- To enable you to specify user-defined conditions and to enable the condition handlers to act on them when the procedure uses the condition name in a SIGNAL or RESIGNAL statement.

The following usage rules apply to specifying a condition name in a handler declaration.

- A condition name specified in a handler declaration must be defined either within the containing compound statement or within a containing outer compound statement.

The following example illustrates how to specify a condition name and its associated SQLSTATE value in a handler. In this example, the condition declaration at line 3 defines condition name *divide_by_zero*, associating it with SQLSTATE '22012'. The EXIT handler at line 4 is then defined to handle *divide_by_zero*. While the procedure executes, the divide by zero exception with SQLSTATE '22012' is raised at line 8, and is handled by the EXIT handler for *divide_by_zero*. After the successful completion of the EXIT handler statements, compound statement *cs1* yields its control of the procedure, which then completes successfully.

```

1.  CREATE PROCEDURE condsp1 (
      INOUT IOParam2 INTEGER,
      OUT  OParam3  INTEGER)
2.  cs1: BEGIN
3.      DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
4.      DECLARE EXIT HANDLER
5.          FOR divide_by_zero, SQLSTATE '42000'
6.          SET OParam3 = 0;
7.      SET IOParam2=0;
8.      SET OParam3 = 20/IOParam2; /* raises exception 22012 */
9.  END cs1;
```

- If a condition value in a handler declaration specifies SQLEXCEPTION, SQLWARNING, or NOT FOUND, you cannot also specify a condition name in the same handler declaration. Otherwise, the procedure aborts and returns an error to the requestor during its compilation.

The following example illustrates how to specify a condition name and a generic condition in different handlers. In this example, a handler is defined at line 4 for SQLEXCEPTION, and a separate handler is defined for condition name *divide_by_zero* at line 7.

```

1.  CREATE PROCEDURE condsp1 (
      OUT OParam3 INTEGER)
2.  cs1: BEGIN
```

```

3.     DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
4.     DECLARE EXIT HANDLER
5.         FOR SQLEXCEPTION
6.             SET OParam3 = 0;
7.     DECLARE EXIT HANDLER
8.         FOR divide_by_zero
9.             SET OParam3 = 1;
10.    ...
11. END cs1;
    DROP TABLE notab;
    CALL condsp1(IOParam2, OParam3);

```

The following example illustrates the *non*-valid specification of a condition name and a generic condition in the same handler. In this example, the handler at line 5 specifies both SQLEXCEPTION and the condition name *divide_by_zero*, which is not valid. A procedure defined in this way aborts and returns an error to the requestor during its compilation.

```

1.  CREATE PROCEDURE condsp2 (
      OUT OParam3 INTEGER)
2.  cs1: BEGIN
3.      DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
4.      DECLARE EXIT HANDLER
5.          FOR SQLEXCEPTION, divide_by_zero
6.              SET OParam3 = 0;
7.      ...
8.  END cs1;

```

- A condition name cannot be specified more than once in a handler declaration. A procedure defined in this way aborts and returns an error to the requestor during its compilation.

The following example illustrates the non-valid specification of the same condition name twice in the same handler. In this example, condition name *divide_by_zero* is specified more than once in the handler at line 5.

```

1.  CREATE PROCEDURE condsp2 (
      OUT OParam3 INTEGER)
2.  cs1: BEGIN
3.      DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
4.      DECLARE EXIT HANDLER
5.          FOR divide_by_zero, divide_by_zero
6.              SET OParam3 = 0;
7.      ...
8.  END cs1;

```

- A handler declaration cannot specify both a condition name and the SQLSTATE value associated with that condition name. A procedure defined in this way aborts and returns an error to the requestor during its compilation.

The following example illustrates the non-valid specification of a condition name and its associated SQLSTATE value in the same handler. In this example, the handler at line 4 is defined to handle both the condition name *divide_by_zero* and the SQLSTATE value '22012' associated with that condition name.

```

1.  CREATE PROCEDURE condsp2 (
      OUT OParam3 INTEGER)
2.  cs1: BEGIN
3.      DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
4.      DECLARE EXIT HANDLER
5.          FOR divide_by_zero, SQLSTATE '22012'
6.              SET OParam3 = 0;
7.      ...
8.  END cs1;
```

- If a handler declaration is defined to handle a condition name, no other handler declaration in the same compound statement can define the SQLSTATE associated with that condition name. A procedure defined in this way aborts and returns an error to the requestor during its compilation.

The following example illustrates the non-valid specification of a condition name and its associated SQLSTATE value in different handlers. In this example, the handler at line 4 is defined to handle *divide_by_zero* and the handler at line 7 is defined for SQLSTATE '22012', which is the SQLSTATE value associated with *divide_by_zero*.

```

1.  CREATE PROCEDURE condsp2 (
      OUT OParam3 INTEGER)
2.  cs1: BEGIN
3.      DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
4.      DECLARE EXIT HANDLER
5.          FOR divide_by_zero
6.              SET OParam3 = 0;
7.      DECLARE EXIT HANDLER
8.          FOR SQLSTATE '22012'
9.              SET OParam3 = 1;
10.  ...
11. END cs1;
```

- You cannot specify multiple handler declarations that have the same condition name in the same compound statement. A procedure defined in this way aborts and returns an error to the requestor during its compilation.

The following example illustrates non-valid specification of the same condition name in multiple handlers. In this example, the handlers at lines 4 and 7 are defined to handle the same condition name, *divide_by_zero*.

```

1.  CREATE PROCEDURE condsp1 (
      OUT OParam3 INTEGER)
2.  cs1: BEGIN
3.      DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
4.      DECLARE EXIT HANDLER
5.          FOR divide_by_zero
6.              SET OParam3 = 0;
7.      DECLARE EXIT HANDLER
8.          FOR divide_by_zero
9.              SET OParam3 = 1;
10.  ...
11.  END cs1;

```

- If you declare a handler to handle a condition name that has an SQLSTATE value associated with it, that handler is also associated with that SQLSTATE value.

The following example illustrates how to specify a condition name and its associated SQLSTATE value in a handler. In this example, the condition declaration at line 3 defines condition name *divide_by_zero* and associates it with SQLSTATE '22012'. The EXIT handler at line 4 is defined to handle *divide_by_zero*. While the procedure executes, the divide by zero exception with SQLSTATE '22012' is raised at line 8 and is then handled by the EXIT handler defined to handle *divide_by_zero*. After the successful completion of the EXIT handler statements, compound statement *cs1* yields its control of the procedure, which then completes successfully.

```

1.  CREATE PROCEDURE condsp1 (
      INOUT IOParm2 INTEGER,
      OUT OParam3 INTEGER)
2.  cs1: BEGIN
3.      DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
4.      DECLARE EXIT HANDLER
5.          FOR divide_by_zero, SQLSTATE '42000'
6.              SET OParam3 = 0;
7.      SET IOParm2=0;
8.      SET OParam3 = 20/IOParm2;      /* raises exception 22012 */
9.  END cs1;

```

- The handler action is associated with every condition name defined in the condition multivalue of a handler declaration.

The following example illustrates the association of the same handler action with multiple condition names. In this example, a CONTINUE handler is defined at line 5 for condition names *divide_by_zero*

and *table_does_not_exist*. While the procedure executes, the CONTINUE handler at line 5 can handle exceptions SQLSTATE '22012' (SQLCODE 2802) and SQLSTATE '42000' (SQLCODE 3807) raised by lines 9 and 10, respectively.

```

1.  CREATE PROCEDURE condsp1 (
      INOUT IOParm2 INTEGER,
      OUT  OParam3 CHARACTER(30))
2.  cs1: BEGIN
3.      DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
4.      DECLARE table_does_not_exist CONDITION FOR SQLSTATE '42000';
5.      DECLARE CONTINUE HANDLER
6.          FOR divide_by_zero, table_does_not_exist
7.          SET OParam3 = 0;
8.      SET IOParm2=0;
9.      SET OParam3 = 20/IOParm2;          /*raises exception 22012*/
10.     INSERT notab VALUES (IOParm2+20);/*raises exception 42000*/
11. END Cs1;
BTEQ> DROP TABLE notab;
BTEQ> CALL condsp1(IOParm2, OParam3);

```

Details About Dynamic Result Sets

You are not limited to the result of an OUT or INOUT value for a CALL request. Dynamic result sets permit an SQL procedure to return a single or as many as 15 result sets. Each result set is returned as it would be for a single multistatement request, not unlike a macro that contains several SELECT requests. The response is the same as the output from a macro: if there are multiple SELECT statements in a macro, then it produces multiple statement responses to the SELECT requests.

The process by which procedures do this is as follows.

1. Using its DYNAMIC RESULT SETS *number_of_sets* specification, the CREATE PROCEDURE statement establishes the maximum number of result sets the procedure can return. This value for *number_of_sets* can range from 0 to 15, inclusive. The specified range applies only to dynamic result sets and does *not* include results returned in INOUT or OUT parameters.

The system returns the result sets in the form a multistatement response spool to the client application.

2. A DECLARE CURSOR statement enables the procedure to create a result set. The SELECT clause of the DECLARE CURSOR statement determines the result set. The DECLARE CURSOR statement must also contain a WITH RETURN clause to be a result set cursor. See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.
3. The system creates the result set when it opens the cursor with an OPEN statement.

The OPEN statement causes the SELECT clause in the DECLARE CURSOR statement to be executed, producing the result set. The result sets are returned in the order they were opened.

The cursor must remain open when the procedure exits in order for the result set to be returned to the caller. This contrasts the case where a procedure returns a single result set, where all cursors are closed when the procedure exits. Cursors are also closed when exiting a compound statement if the cursor was opened within the compound statement.

However, a cursor that specifies a `WITH RETURN` clause is not closed when the procedure exits. If the procedure does close the result cursor, then the result set is deleted and not returned to the caller.

Result sets are returned in the order they were opened.

4. If a procedure is called from a client or an external procedure, the result set is returned in the form of a multistatement result. This is exactly the same form that would be created if a macro that contained multiple `SELECT` statements within it was executed.

The first result is for the first statement, and are the results for any `OUT` or `INOUT` arguments to the procedure. The second and subsequent results spool are the output for the result sets in the order they were opened.

| Procedure Called By | Result Set Returned To |
|---------------------|------------------------|
| Procedure. | Procedure. |
| Client application. | Client application. |

5. A procedure can use the dynamic form of the `DECLARE CURSOR` statement.

In this case, the `DECLARE CURSOR` statement specifies a statement name in place of the `SELECT` statement that it would specify for the static case.

The statement name is referenced by the `PREPARE` statement to prepare the dynamic `SELECT` statement contained in a string. The prepared `SELECT` statement is executed when the cursor is opened with the `OPEN` statement.

The system supports this functionality by means of the following SQL cursor and control language statements.

- `CREATE PROCEDURE` and `REPLACE PROCEDURE` statement `DYNAMIC RESULT SETS` clause.
The `DYNAMIC RESULT SETS` clause specifies the maximum number of result sets to be returned.
- `DECLARE CURSOR` statement `WITH RETURN` clause.

The `WITH RETURN` clause specifies that the cursor to be opened is a result set cursor to be returned to the client or to the caller.

See [Rules and Limitations for Dynamic Result Sets](#) and *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

- `PREPARE` statement

This statement enable a procedure to create a dynamic `DECLARE CURSOR` statement to allow the creation of different result sets.

`PREPARE` allows dynamic parameter markers.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

- OPEN statement USING clause

The USING clause supports the use of dynamic parameter markers used in a dynamic DECLARE CURSOR statement.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

A DYNAMIC RESULT SETS clause must be specified to return a result set to the caller of the procedure, whether the caller is a client application or another procedure. This clause is optional and is not used if the procedure does not return a result set. If you do not specify a DYNAMIC RESULT SETS clause, then the system assumes zero result sets. Note that this total does not include results returned in an INOUT or OUT parameter.

You can return a result set to either a calling procedure or to a client application, but not to both.

The result set output for a procedure appears in the response spool after the results for the procedure OUT or INOUT parameters result row. The result set is a response spool similar to how a response spool would appear if a macro had been written to perform several SELECT statements. The only difference is an extra parcel indicating that the result set came from a procedure.

For example, the response spool for the following macro and the procedure following it are identical except for the title for the procedure output arguments versus the macro and the success response.

```
CREATE MACRO sample_m (c INTEGER) AS (
  SELECT :c+1;

  SELECT *
  FROM m1;

  SELECT *
  FROM m2
  WHERE m2.a > :c;) ;
```

The BTEQ output from this macro is as follows.

```
EXEC sample_m(1);
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.

      (c+1)
-----
        2

*** Query completed. One row found. 2 columns returned.

      a              b
```

```

-----
      1      2.000000000000000E 000

*** Query completed. One row found. 2 columns returned.

      a      b
-----
      2      4.000000000000000E 000

```

The following procedure that uses result sets produces the same output.

```

CREATE PROCEDURE sample_p (INOUT c INTEGER)
  DYNAMIC RESULT SETS 2
  BEGIN
    DECLARE cursor_1 CURSOR WITH RETURN FOR
      SELECT * FROM m1;
    DECLARE cursor_2 CURSOR WITH RETURN FOR
      SELECT * FROM m2 WHERE m2.a > c;
    SET c = c +1;
    OPEN cursor_1;
    OPEN cursor_2;
  END;

```

The BTEQ output from the procedure is as follows.

```

CALL sample_p(1);
*** Procedure has been executed.
*** Total elapsed time was 1 second.
      1
-----
      2

*** Procedure has been executed. One row found. 2 columns returned.
*** Starting Row Number: 1
*** Database Name: FSK
*** Procedure Name: SAMPLE_P

      a      b
-----
      1      2.000000000000000E 000

*** Procedure has been executed. One row found. 2 columns returned.
*** Starting Row Number: 1
*** Database Name: FSK
*** Procedure Name: SAMPLE_P

      a      b
-----
      2      4.000000000000000E 000

```

The called procedure that contains the DYNAMIC result sets clause returns the following possible SQLSTATE warnings.

| SQLSTATE Code | Meaning |
|---------------|---|
| '0100C' | The procedure returned additional result sets. The system returns this SQLSTATE warning code when the procedure produces result sets. |
| '0100E' | The procedure attempted to return too many result sets. The system returns this SQLSTATE warning code when the procedure leaves more result set cursors open than are specified in the DYNAMIC RESULT SETS clause. |

Rules and Limitations for Dynamic Result Sets

The following rules and limitations apply to SQL procedure dynamic result sets.

- No parameter can have a TD_ANYTYPE data type.
- You can return dynamic result sets for a single SELECT request only.
Multistatement SELECT requests are not valid for dynamic result sets.
- You cannot return dynamic result sets to both a called procedure and its calling procedure or calling client application.
- You cannot close an open cursor if you want to return dynamic result sets.
- The receiver of a dynamic result set must use its driver (such as CLv2 or ODBC) to indicate that it will accept the results. See *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, and *ODBC Driver for Teradata® User Guide* for details.
- Use the WITH RETURN ONLY clause to specify that the procedure is to use the cursor as a result set cursor.
- Cursor rules are identical to those for procedures that do not support dynamic result sets (see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details) except for the changes and restrictions indicated in the following list.
 - You cannot specify PREPARE for a cursor unless it handles result sets.
 - You cannot specify OPEN ... USING for a cursor unless it handles result sets.
 - You cannot specify FOR UPDATE with a WITH RETURN clause.
 - If you specify WITH RETURN ONLY ..., then the cursor cannot be fetched within the procedure that opened it.
 - You cannot close a dynamic result sets cursor once it has been opened because closing the cursor results in the result set not being returned.
 - The returned result set inherits the following CLv2 response attributes from the caller, not from the procedure that created it.

- Response mode
- Keep response
- LOB response mode

For example, if you submit a CALL from BTEQ that generates a result set, the result set sent to the procedure is in Indicator mode, while the result set sent to BTEQ is in Field mode.

See *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 for details.

- The collation sequence for the returned result set is that of the called procedure, not the collation sequence of the caller or the session.
- The starting position of the returned result set and the set of rows in the returned result set is determined by the scrollability of the cursor.

The system always returns the entire result set.

- The various options have performance implications based on how much data the system generates for them as the following table indicates.

| Option | IF the caller is a procedure THEN the ... | IF the caller is a client application THEN the ... |
|----------------------------|--|--|
| WITH RETURN ONLY | <ul style="list-style-type: none"> ◦ system generates 1 result spool table for calling procedures. | <ul style="list-style-type: none"> ◦ system generates 1 result spool table for the client application. <p>The system returns a value of '02000' for SQLSTATE if you attempt to fetch rows from a procedure.</p> |
| WITH RETURN ONLY TO CLIENT | <ul style="list-style-type: none"> ◦ system generates 1 result spool table for the client application. | <ul style="list-style-type: none"> ◦ system generates 1 result spool table for the client application. |
| WITHOUT RETURN | <ul style="list-style-type: none"> ◦ system generates 1 result spool table. ◦ spool is closed when the procedure exists or exits the compound block cursor in which it is defined. | <ul style="list-style-type: none"> ◦ system generates 1 result spool table. ◦ spool is closed when the procedure exists or exits the compound block cursor in which it is defined. |

- You can refer to recursive views and recursive queries from a procedure definition.
- You cannot refer to a WITH clause from a procedure definition.

Teradata Unity Support for SQL Procedures

Teradata Unity sends request-specific context information as part of a request that calls an SQL procedure to enable Vantage to change the result of the procedure indirectly by substituting a value predefined by Teradata Unity for a non-deterministic result. Vantage makes this context information available to the SQL statements embedded within a procedure when it is called from the default connection for the session.

However, SQL procedures can generate and use their own arbitrary non-deterministic values that Vantage has no knowledge of. Therefore, Vantage cannot guarantee that an SQL procedure call produces a consistent result in a Teradata Unity environment.

SQL Procedure Support for Transaction Query Bands

You can set the transaction query band (but *not* the session query band) using a parameter value, including a QUESTION MARK parameter, that is passed to an SQL procedure.

SQL Procedures and Proxy Connections

The following rules apply to procedures created in a proxy connection.

- The immediate owner of the procedure is the creator of all permanent objects created through the procedure.
- During procedure execution, Vantage checks the privileges of the immediate owner of the procedure for all statements specified and all objects referenced in the procedure body.
- The CONNECT THROUGH privilege for SET QUERY_BAND with a PROXYUSER in a procedure is validated against the trusted user when the procedure is executed.

SQL Procedure Support For UDTs

For correct operation of a UDT within a procedure, the UDT must have its mandatory ordering and transform functionality defined. Additionally, the tosql and fromsql transform routines must be duplicated by an equivalent set of predefined data type-to-UDT and UDT-to-predefined data type implicit cast definitions. You can do this by referencing the same routines in both the CREATE TRANSFORM and CREATE CAST statements.

For distinct UDTs, if you plan to use the system-generated functionality, no additional work is required because the transform and implicit casting functionality have already been defined. For structured UDTs, you must explicitly define the transform and cast functionality, as shown by the following examples.

```
/* Transform Functionality */
CREATE TRANSFORM FOR address ClientIO (
  TO SQL WITH SPECIFIC FUNCTION SYSUDTLIB.stringToAddress,
  FROM SQL WITH SPECIFIC METHOD toString);

/* Implicit Cast To Back Up The ToSql Functionality */
CREATE CAST ( Varchar(100) AS address )
  WITH SPECIFIC FUNCTION SYSUDTLIB.stringToAddress
  AS ASSIGNMENT;

/* Implicit Cast To Back Up The FromSql Functionality */
CREATE CAST (address AS Varchar(100))
```

```
WITH SPECIFIC METHOD ToString
AS ASSIGNMENT;
```

You can declare an input parameter to have the VARIANT_TYPE UDT data type *only* within the body of the procedure definition, which means that the system passes the dynamic UDT data type to this parameter position during runtime using the NEW VARIANT_TYPE expression (see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210). You *cannot* declare an IN parameter data type for the procedure itself to have the VARIANT_TYPE data type.

You cannot declare the data type for any parameter to be TD_ANYTYPE.

Details about SQL procedure support for the following DML statements are documented in *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

- SELECT INTO a UDT local variable.

You can SELECT INTO a UDT local variable. No expressions are allowed in the INTO list.

- FETCH INTO a UDT local variable.

You can FETCH INTO a UDT local variable. No expressions are allowed in the INTO list.

You cannot perform a FETCH INTO operation in which you attempt to fetch into a predefined type nor can you perform a fetch into operation that involves UDT types that are not 100 percent identical. In other words, no implicit casts are applied for these operations.

- INSERT INTO a UDT local variable.

You can INSERT INTO a UDT local variable in a table, but you cannot invoke methods on such a local variable if those methods are invoked within the INTO clause.

Supported DDL Statements in SQL Procedures

Teradata supports the following SQL DDL statements for SQL procedures.

| | | |
|--|---|---|
| <ul style="list-style-type: none"> • ALTER FUNCTION • ALTER TABLE • ALTER TABLE TO CURRENT • ALTER TRIGGER • BEGIN LOGGING • COLLECT STATISTICS (Optimizer Form) • COMMENT • CREATE CAST • CREATE DATABASE • CREATE ERROR TABLE • CREATE FUNCTION (External Form) | <ul style="list-style-type: none"> • CREATE TABLE • CREATE TRANSFORM • CREATE TRIGGER • CREATE USER • CREATE VIEW • DELETE DATABASE • DELETE USER • DROP CAST • DROP DATABASE • DROP HASH INDEX • DROP INDEX • DROP JOIN INDEX • DROP MACRO • DROP ORDERING • DROP PROCEDURE | <ul style="list-style-type: none"> • END LOGGING • MODIFY DATABASE • MODIFY PROFILE • MODIFY USER • RENAME FUNCTION (External Form) • RENAME FUNCTION (SQL Form) • RENAME MACRO • RENAME PROCEDURE • RENAME TABLE • RENAME TRIGGER • RENAME VIEW • REPLACE CAST • REPLACE FUNCTION |
|--|---|---|

| | | |
|---|---|--|
| <ul style="list-style-type: none"> • CREATE FUNCTION (SQL Form) • CREATE HASH INDEX • CREATE INDEX • CREATE JOIN INDEX • CREATE MACRO • CREATE ORDERING • CREATE PROFILE • CREATE RECURSIVE VIEW • CREATE ROLE | <ul style="list-style-type: none"> • DROP PROFILE • DROP ROLE • DROP STATISTICS (Optimizer Form) • DROP TABLE • DROP TRANSFORM • DROP TRIGGER • DROP USER • DROP VIEW | <ul style="list-style-type: none"> • REPLACE FUNCTION (SQL Form) • REPLACE MACRO • REPLACE ORDERING • REPLACE TRANSFORM • REPLACE TRIGGER • REPLACE VIEW • SET QUERY_BAND ... FOR TRANSACTION |
|---|---|--|

Usage Considerations for DDL Statements in Procedures

- You can use DDL COMMENT statements in a procedure. You *cannot* specify DML COMMENT statements, which are restricted to embedded SQL applications, to fetch the comments for database objects, columns of a table, and parameters.
- The queue table form of CREATE TABLE (see [CREATE TABLE \(Queue Table Form\)](#)) cannot be executed in a procedure. All other forms of the CREATE TABLE statement are valid.
- If a CREATE VOLATILE TABLE statement is included in a procedure, the volatile table is created in the database of the user. If an object with the same name already exists in that database, the result is a runtime exception.

DML statements within a procedure referencing the volatile table must either have the login database of the user as the qualifier, or not have any qualifying database name.

- A CREATE DATABASE or CREATE USER statement in a procedure must contain the FROM clause. The statement result depends on the FROM clause as indicated by the following table.

| WHEN a procedure ... | THEN ... |
|--------------------------------|--|
| contains a FROM clause | the specified database is the immediate owner of the user or database created. |
| does not contain a FROM clause | the system reports an SPL compilation error during procedure creation. If you specify either a CREATE USER statement or a CREATE DATABASE statement without a FROM clause as a dynamic SQL statement within a procedure, the same error is reported as a runtime exception during procedure execution. |

Unsupported DDL Statements in SQL Procedures

You cannot specify the following DDL statements in an SQL procedure.

| | | |
|--|--|---|
| <ul style="list-style-type: none"> • ALTER CONSTRAINT • ALTER METHOD | <ul style="list-style-type: none"> • CREATE TABLE (queue and trace table forms) | <ul style="list-style-type: none"> • REPLACE METHOD • REPLACE PROCEDURE |
|--|--|---|

| | | |
|---|--|---|
| <ul style="list-style-type: none"> • ALTER PROCEDURE • ALTER TYPE • BEGIN QUERY LOGGING • CREATE CONSTRAINT • CREATE GLOP SET • CREATE METHOD • CREATE PROCEDURE | <ul style="list-style-type: none"> • CREATE TYPE (all forms) • DATABASE • DROP CONSTRAINT • DROP GLOP SET • DROP TYPE • END QUERY LOGGING • FLUSH QUERY LOGGING • HELP (all forms) | <ul style="list-style-type: none"> • REPLACE QUERY LOGGING • REPLACE TYPE • SET QUERY_BAND ... FOR SESSION • SET ROLE • SET SESSION (all forms) • SET TIME ZONE • SHOW (all forms) |
|---|--|---|

You also cannot specify any DDL statements in an SQL procedure that administer any aspect of any row-level security constraints.

Note that these statements are restricted from being specified in CREATE PROCEDURE or REPLACE PROCEDURE DDL SQL text, *not* from being invoked as dynamic SQL from within a procedure definition. Dynamic SQL requests are parsed and bound at run time, not when the procedure that contains them is compiled, so it does not follow the same rules as must be observed when SQL requests are executed from within a compiled procedure. Any SQL statement that can be invoked as standalone dynamic SQL can also be invoked as dynamic SQL from within a procedure definition.

Session Mode Impact on DDL Requests in Procedures

The behavior of DDL requests specified within the body of SQL procedures at runtime depends on the session mode in place when the procedure is created.

- A DDL request specified within an explicit transaction in a procedure in Teradata session mode must be the last request specified in that transaction. Otherwise, the system raises a runtime exception.
- When you perform a procedure in ANSI session mode, each DDL request specified in the procedure body must be explicitly terminated by a COMMIT WORK request. Otherwise, the system raises a runtime exception.

Supported DML Statements in SQL Procedures

You can specify the following SQL DML statements in an SQL procedure.

| | |
|---|---|
| <ul style="list-style-type: none"> • ABORT • BEGIN TRANSACTION • CALL • COLLECT STATISTICS (QCD Form) • COMMENT • COMMIT [WORK] • DECLARE CURSOR (selection form) • DELETE (all forms) • DROP STATISTICS (QCD Form) • END TRANSACTION | <ul style="list-style-type: none"> • ROLLBACK • SELECT (only in cursors) You cannot specify row-returning SELECT requests in dynamic SQL that is written using the <i>DBC</i>. <i>SysExecSQL</i> procedure. • SELECT AND CONSUME TOP 1 (only in non-updatable cursors) • SELECT INTO You cannot specify row-returning SELECT INTO requests in dynamic SQL that is written using the <i>DBC</i>. <i>SysExecSQL</i> procedure. • SELECT AND CONSUME TOP 1 INTO |
|---|---|

- | | |
|---|--|
| <ul style="list-style-type: none"> • INSERT • MERGE | <ul style="list-style-type: none"> • UPDATE (all forms) |
|---|--|

Unsupported DML Statements in SQL Procedures

You *cannot* specify the following SQL DML statements in an SQL procedure.

- | | |
|--|---|
| <ul style="list-style-type: none"> • CHECKPOINT • COLLECT DEMOGRAPHICS • DUMP EXPLAIN • ECHO • INITIATE INDEX ANALYSIS • INITIATE PARTITION ANALYSIS | <ul style="list-style-type: none"> • INSERT EXPLAIN • RESTART INDEX ANALYSIS • SELECT (outside of a cursor) • SELECT AND CONSUME • SELECT AND CONSUME (in updatable cursors) |
|--|---|

Supported DCL Statements in SQL Procedures

You can specify the following SQL DCL statements in an SQL procedure when the creator is also the immediate owner of the procedure. That is, an SQL procedure can contain DCL statements only if it is created in the creating in the database of the user.

- | | |
|---|--|
| <ul style="list-style-type: none"> • GIVE • GRANT (all forms) • GRANT CONNECT THROUGH • GRANT LOGON | <ul style="list-style-type: none"> • REVOKE (all forms) • REVOKE CONNECT THROUGH • REVOKE LOGON |
|---|--|

Unsupported DCL Statements in SQL Procedures

You cannot create an SQL procedure that contains any DCL request that administers row-level security privileges.

Session Mode Impact on DCL Statements in SQL Procedures

The runtime behavior of DCL statements specified in SQL procedures depends on the session mode of the Teradata session in which the procedure is created.

- A DCL statement specified within an explicit (user-defined) transaction in an SQL procedure in Teradata session mode must be the last statement in that transaction. Otherwise, the system raises a runtime exception.

- When performing a procedure in ANSI session mode, each DCL statement specified in the procedure body must be followed by a COMMIT WORK request. Otherwise, the database aborts the request and raises a run time exception.

Rules for Using the PREPARE Statement In an SQL Procedure

The following rules apply to using the PREPARE statement in an SQL procedure definition.

- The PREPARE statement prepares the dynamic cursor SQL statement for execution.
- You can use the PREPARE statement only for procedures that return result sets.
- The SQL statement name specified by a PREPARE statement must be a standard SQL identifier.
- The system passes PREPARE statements to the Parser for syntax checking.

If there is a syntax error, the statement returns a syntax exception error to the requestor.

- The statement must be a dynamic cursor SELECT statement if a PREPARE statement references a statement name in a dynamic DECLARE CURSOR statement.

Otherwise, the system returns an SQLSTATE '07005' error, meaning that the prepared statement is not a cursor specification.

- Whether specified as a string expression or as a variable, the dynamic SQL statement text can be as long as 64 kbytes.

This maximum includes SQL text, USING data, and CLlv2 parcel overhead.

- You can specify an OPEN ... USING statement only for a cursor that returns result sets.
- You cannot specify multistatement requests as part of a PREPARE statement.
- A dynamic SQL statement can include either parameter markers or placeholder tokens (the QUESTION MARK character) where any literal, particularly a SQL variable, reference is allowed.

The only exception to this is that neither parameter markers nor placeholder tokens can be specified in the select list.

- The system supplies values to the statement by means of the USING clause of the OPEN statement.
- You cannot perform a PREPARE statement as a standalone dynamic SQL statement.
- A maximum of 15 dynamic SQL statements is permitted in any procedure definition.

For more information, see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

Guidelines for Using the CURRENT_TIME and CURRENT_TIMESTAMP Functions in an SQL Procedure

When you create a procedure that contains the CURRENT_TIME or CURRENT_TIMESTAMP functions, Vantage incorporates the value for the function at the time the procedure is compiled using the time or time zone value for the creator of the function based on the local session time zone settings of the DBS Control parameters SystemTimeZoneHour and SystemTimeZoneMinute.

If you adjust the settings of these parameters to make a time zone adjustment, any procedure created with the intent of returning a current value for the `CURRENT_TIME` or `CURRENT_TIMESTAMP` functions that is adjusted for the local time no matter where the procedure is invoked will not return the intended result.

You could recompile the procedure periodically, but that is not a universally applicable solution to the problem.

The best practice is to insert an appropriate time zone string into the `tdlocaledef` file for your site. That way, Vantage automatically makes the necessary time zone changes, ensuring that the `CURRENT_TIME` and `CURRENT_TIMESTAMP` functions within the procedure always return an appropriate value.

See [Time Zone Strings](#) for a complete list of valid time zone strings.

SQL Multistatement Request Support in SQL Procedures

You can specify standard Teradata multistatement requests within the procedure body for procedures created in either ANSI or Teradata session modes. The feature has the same limitations as the standard multistatement requests submitted by means of BTEQ, embedded SQL, or other standard SQL statement entry paths (see *Teradata Vantage™ - SQL Fundamentals*, B035-1141). Like other SQL multistatement requests, the system sends procedure multistatement requests to the AMPs in parallel.

The statements contained within the multistatement request are restricted to DML statements such as `INSERT`, `UPDATE` and `DELETE`. You cannot specify any other type of SQL statements within the block.

You cannot include a `SELECT AND CONSUME` statement within the same multistatement request or explicit transaction as a `DELETE`, `MERGE`, or `UPDATE` statement that operates on the same queue table.

If there are no errors in any of the statements within the multistatement request block, the system does not post activity results.

If multiple statements specify a completion condition, the system returns only the condition code for the first, which is also the only statement in the block that is handled by an exception handler.

If the request aborts, the system rolls back all SQL statements within it and, if one has been specified, passes control to an exception handler. If no exception handler has been specified, the procedure exits with the error.

The multistatement block can be composed of multiple individual semicolon-separated DML statements or 1 dynamic DML statement. No other type of statement is valid.

The items in the following list are the only SQL statements that can be sent as part of a procedure multistatement request.

| | |
|---|--|
| <ul style="list-style-type: none">• <code>ABORT</code>• <code>BEGIN TRANSACTION</code>• <code>COMMIT</code>• <code>DELETE</code>• <code>END TRANSACTION</code>• <code>INSERT</code>• <code>MERGE</code> | <ul style="list-style-type: none">• <code>ROLLBACK</code>• <code>SELECT INTO</code> You cannot specify row-returning <code>SELECT</code> or <code>SELECT INTO</code> requests in dynamic SQL that is written using the <code>DBC.SysExecSQL</code> procedure.• <code>UPDATE</code> |
|---|--|

| IF the session is in this mode ... | THEN multiple INSERT ... SELECT statements ... |
|------------------------------------|---|
| ANSI | <p>use the fast path if the last statement in the request is a COMMIT statement.</p> <p>A DELETE ALL statement uses the fast path if the next and last sequential statement in the multistatement request is a COMMIT statement.</p> <p>The same applies when deleting entire PPI partitions.</p> |
| Teradata | <p>use the fast path if the multistatement request is an implicit transaction or an explicit transaction that terminates with an END TRANSACTION statement</p> <p>A DELETE ALL statement uses the fast path if it is either of the following.</p> <ul style="list-style-type: none"> • The last statement in the implicit request. • An explicit transaction terminated by an END TRANSACTION statement. <p>The same applies when deleting entire PPI partitions.</p> |

The dynamic SQL feature allows you to build ad hoc SQL statements within a procedure application. The system can build a request in one of the following ways:

- Using the SQL PREPARE statement

See [Rules for Using the PREPARE Statement In an SQL Procedure](#) and *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

Note that dynamic SQL written in this form *can* specify SELECT and SELECT INTO statements that return rows. See [Details About Dynamic Result Sets](#) and [Rules and Limitations for Dynamic Result Sets](#).

- Using the DBC.SysExecSQL procedure call for several semicolon-separated SQL DML statements.

When multiple statements are created in this way, the system executes the entire set as 1 request.

The set of SQL statements that is valid within a dynamic SQL request is the same as the valid set for static SQL. See [Supported DDL Statements in SQL Procedures](#), [Unsupported DML Statements in SQL Procedures](#), and [Supported DCL Statements in SQL Procedures](#).

Note that dynamic SQL written in this form cannot specify SELECT and SELECT INTO statements that return rows.

You can specify only one dynamic request per multistatement request.

Utilities and APIs Supporting CREATE/REPLACE PROCEDURE

You can create a procedure using the COMPILE command in BTEQ and BTEQWIN. See *Basic Teradata® Query Reference*, B035-2414.

You can also create a procedure using the CREATE PROCEDURE statement from CLv2, ODBC, and JDBC applications.

The following client software packages support procedure execution and DDL operations:

- BTEQ
- CLIV2
- JDBC
- ODBC
- PreProcessor2

For details on the use of local variables and condition handlers in SQL procedures, see the description of DECLARE and DECLARE HANDLER statements in *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

SQL Procedure Options

SQL procedures have 1 user-specified attribute, the source text storage option, as detailed in the following table.

| SPL Option | Description |
|------------|--|
| SPL | Stores the DDL definition source text of the procedure in the dictionary. SPL is the default option. |
| NO SPL | Does not store the source text of the procedure in the dictionary. |

You can specify this option in different ways from the different client utilities and interfaces from which the procedure is created, as described by the following table.

| FOR this API ... | SPECIFY the source text storage options in this way ... |
|------------------|---|
| BTEQ | as a parameter of the COMPILE command. |
| CLIV2 | with the SPOptions parcel using the DBCAREA extension Initiate Request. |
| ODBC | as a parameter specified in different ways depending on the platform. |

See the appropriate Teradata Tools and Utilities documents.

The ALTER PROCEDURE NO WARNING option enables suppression of procedure compilation warnings. See [ALTER PROCEDURE \(SQL Form\)](#).

Guidelines for Manipulating LOBs in an SQL Procedure

You must always exercise care when writing SQL procedures that access LOBs for the following reasons.

- While procedures execute in the PE, LOBs are stored on the AMPs.
- Procedures store a reference to the LOB that resides on the AMPs.
- All procedure operations that need access to a LOB must transfer that entire LOB to the PE on which the procedure is running.

The following example converts a CLOB containing XML sales data into a set of rows that are inserted into a sales table. The procedure also does the following:

- Stores the CLOB in a log table.
- Stores half of the CLOB into a local variable.
- Stores half of the CLOB into a different log table.

```
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);

CREATE TABLE saleshalflog (
    storecode  INTEGER,
    salesdate  DATE,
    sales      CLOB,
    logdate    DATE,
    logtime    TIME)
PRIMARY INDEX (storecode, salesdate, logdate, logtime);

CREATE PROCEDURE storessalesprocedure (
    storecode INTEGER,
    salesdate DATE,
    salesclob CLOB)
BEGIN
    DECLARE localclob CLOB;
    SET localclob = SUBSTR(:salesclob,1,CHARACTERS(:salesclob)/2 );
    INSERT saleslog (:storecode, :salesdate, :salesclob,
                    CURRENT_DATE, CURRENT_TIME );
    INSERT saleshalflog (:storecode, :salesdate, :localclob,
                        CURRENT_DATE, CURRENT_TIME );
    INSERT sales SELECT * FROM TABLE (xmlparser(:salesclob));
END;
```

Suppose the procedure were to be invoked by passing a 1 MB CLOB, as follows.


```

USING (
    storecode INTEGER,
    salesdate DATE,
    salesclob CLOB AS DEFERRED)
CALL storessalesprocedure (:storecode, :salesdate, :salesclob);

```

For this case, the following process occurs:

1. The CLOB is transferred from the client application to an AMP.
2. The procedure is invoked with a reference to the 1 MB CLOB and the other data.
3. The procedure builds a new CLOB, called *localclob*, by taking the substring of the 1 MB CLOB.
4. The entire 1 MB CLOB is transferred to the PE.
5. A half-MB CLOB is transferred to the AMP to be written for *LocalClob*.
6. A reference to *LocalClob* is stored in the PE.
7. The INSERT into the *SalesLog* table is sent to the Parser with a reference to the 1 MB CLOB.
8. The Parser sends an INSERT step, containing a reference to the 1 MB CLOB, to the AMP.
9. The AMP retrieves the 1 MB CLOB from the AMP where it is stored and inserts it into the *SalesLog* table.
10. The INSERT into the *SalesHalfLog* table is sent to the Parser with a reference to the half-MB CLOB.
11. The Parser sends an INSERT step, containing a reference to the half-MB CLOB, to the AMP.
12. The AMP retrieves the half-MB CLOB from the AMP where it is stored and inserts into the *SalesHalfLog* table.
13. The INSERT into the *Sales* table is sent to the Parser with a reference to the 1 MB CLOB.
14. The Parser sends INSERT ... SELECT steps, containing a reference to the 1 MB CLOB, to all AMPs.
15. The AMP containing the 1 MB CLOB converts it into rows that are hash-redistributed to other AMPs.
16. Those rows are inserted into the *Sales* table.

Note:

The use of *LocalClob* was unnecessary. It would have been more efficient not to use the local variable and instead place the substring expression in the INSERT statement like this.

```

INSERT INTO saleshalflog VALUES (:storecode, :salesdate, SUBSTR(
                                :salesclob, 1, CHARACTERS(:salesclob)/2),
                                CURRENT_DATE, CURRENT_TIME);

```

If it is not possible to place the LOB functions in the SQL statements, then consider converting the data types of the LOBs to VARBYTE or VARCHAR. Each conversion still brings the entire LOB into the PE, but for small LOBs it might be an option.

For SELECT operations inside a procedure, use of the FOR syntax can prevent unnecessary transfer of LOBs. Consider the following example that shows a procedure selecting a LOB and inserting it into another table.

```

CREATE TABLE sales (
  partnum    INTEGER,
  qtysold    INTEGER,
  storecode  INTEGER,
  salesdate  DATE)
PRIMARY INDEX (partnum, qtysold, storecode, salesdate);

CREATE TABLE saleslog (
  storecode  INTEGER,
  salesdate  DATE,
  sales      CLOB,
  logdate    DATE,
  logtime    TIME)
PRIMARY INDEX (storecode, salesdate, logdate, logtime);

CREATE PROCEDURE stores_sales_log_procedure
  (storecode INTEGER, salesdate DATE )
BEGIN
  FOR cur AS curname CURSOR FOR
    SELECT *
    FROM saleslog
    WHERE storecode = :storecode
    AND   salesdate = :salesdate;
  DO
    IF (:cur.lobdate = DATE-1) THEN
      INSERT sales
      SELECT *
      FROM TABLE (xml_sales_parser(:cur.sales));
    END IF;
  END FOR;
END;

```

The procedure named *stores_sales_log_procedure* performs the following process.

1. Stores a reference to the sales CLOB in *:cur.sales*.
2. Sends this reference with the INSERT request into the *sales* table request to the Parser and the INSERT ... SELECT steps sent to the AMPs.
3. If *:cur.sales* were then copied to a local variable, the CLOB is transferred from the AMP to the PE and then back again to the AMP.

The SELECT INTO- and FETCH INTO-styles of procedure SELECT statements make copies to local variables that cause extra LOB transfers from the AMP to the PE and back to the AMP.

The following case is an example of how *not* to create an efficient procedure for handling LOB data.

```

CREATE TABLE sales (
    partnum    INTEGER,
    qtysold    INTEGER,
    storecode  INTEGER,
    salesdate  DATE)
PRIMARY INDEX (partnum, qtysold, storecode, salesdate);

CREATE TABLE saleslog (
    storecode  INTEGER,
    salesdate  DATE,
    sales      CLOB,
    logdate    DATE,
    logtime    TIME)
PRIMARY INDEX (storecode, salesdate, logdate, logtime);

CREATE PROCEDURE stores_sales_stream_proc
(storecode INTEGER, salesdate DATE, salesclob CLOB)
BEGIN
    DECLARE resultrownum INTEGER;
    DECLARE partnum      INTEGER;
    DECLARE qtysold      INTEGER;
    INSERT INTO SalesLog VALUES (:storecode, :salesdate, :sales,
                                CURRENT_DATE, CURRENT_TIME );
    CALL xmlesalesparser(:resultrownum, :partnum, :qtysold, :salesclob);
    WHILE (resultrownum > 0 )
    DO
        INSERT INTO Sales VALUES (:partnum, :qtysold, :storecode,
                                :salesdate);
        CALL xmlesalesparser(:resultrownum, :partnum, :qtysold, :salesclob);
    END WHILE;
END;

```

Procedure for Creating an SQL Procedure

Use the following procedure to create an SQL procedure. Steps 4, 5 and 6 apply only when a compound statement is specified.

1. Identify and specify the procedure name after the keywords CREATE PROCEDURE. The *procedure_name* can optionally be qualified with *database_name*.
2. Identify the parameters, if desired, and specify them immediately after the procedure name.

The entire set of parameters is in a comma-separated list, and must be enclosed by LEFT PARENTHESIS and RIGHT PARENTHESIS characters.

- Each parameter consists of 3 elements, in the following left-to-right order.

- Parameter type (optional)
- Parameter name (mandatory)
- Data type of the parameter (mandatory)

You cannot specify a character parameter data type with a server character set of KANJI1. Otherwise, the database aborts the request and returns an error to the requestor.

3. Specify either a single statement to perform the main task, or the BEGIN keyword for a compound statement.

If you have specified a single statement, terminate that with a final SEMICOLON character to complete the CREATE PROCEDURE statement.

If you have started to specify a compound statement, go to step 4.

4. Following the BEGIN keyword, specify any variable declarations, cursor declarations, and condition handler declarations, in that order.

Within each handler declaration, specify a single statement or a BEGIN-END compound statement as the handler action.

5. Specify the statements to perform the main tasks.

These statements must be specified *after* the variable, cursor, and condition handler declarations.

6. Terminate the CREATE PROCEDURE statement with a final SEMICOLON character after the END keyword.

You can specify an optional ending label for the BEGIN-END block after the END keyword. If you specify the ending label, then you must specify an equivalent beginning label suffixed with a COLON (:) character before the BEGIN keyword.

Completing the Creation of an SQL Procedure

On successful completion of the procedure object, a success response is returned to the client application.

The following statements need to be specified to complete the procedure creation, depending on the session mode.

| Session Mode | Statements Required |
|--------------|--|
| ANSI | An explicit COMMIT statement. |
| Teradata | An explicit END TRANSACTION statement if the transaction is explicit. No such statement is required if the transaction is implicit. |

Aborting the Creation of an SQL Procedure

You can terminate the creation of an SQL procedure within an open transaction by running an ABORT or ROLLBACK request.

Processing an SQL CREATE PROCEDURE Request

A compiler in the database performs syntax and semantic checks on the source text of the SQL procedure.

The processing by the Parser involves the following stages in addition to the routine for creating any database object.

1. The database validates the SQL statements, including any expressions, inside the procedure body.
2. The errors or warnings encountered during validation are treated as procedure compilation errors or warnings. These are reported to the client utility as part of the success response.
3. On successful creation of a procedure, a success response is returned to the client application with warning code = 0.
4. If a failure is encountered, a failure parcel is returned. An error is treated as a failure. The current transaction is rolled back.

Retrieving an SQL Procedure Body

Use the SHOW PROCEDURE statement to display procedure source text. The text is returned in the same format as defined by the creator.

Use new line (CR/LF) characters wherever possible in the source text to increase its readability.

Function of REPLACE PROCEDURE Requests

REPLACE PROCEDURE executes as a DROP PROCEDURE request followed by a CREATE PROCEDURE request except for the handling of the privileges granted to the original procedure. The database retains all of the privileges that were granted directly on the original procedure.

If the specified procedure does not exist, REPLACE PROCEDURE has the same effect as performing a CREATE PROCEDURE statement.

If an error occurs during the replacement and the operation fails, the original procedure remains in place as it was prior to specifying the REPLACE PROCEDURE: the procedure is not dropped. This is analogous to rolling back the operation.

Significance of Platform and Session Mode For REPLACE PROCEDURE

If a procedure created in Teradata session mode is being replaced in ANSI session mode, it acquires the new session mode attributes and thereafter can only be performed in ANSI session mode.

Similarly, if a procedure created in ANSI session mode is replaced in Teradata session mode, it acquires the new session mode attributes and thereafter can only be performed in Teradata session mode.

You can replace an SQL procedure created with one default character set with a different default character set.

CREATE RECURSIVE VIEW and REPLACE RECURSIVE VIEW

Updatability of Recursive Views

Recursive views are not updatable.

The Concept of Recursion

A recursive view is a named query expression whose definition is self-referencing. See [Building a Recursive View](#). For information about the WITH RECURSIVE clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

The self-referencing property provides a method for searching a table set by means of iterative self-join and set operations. You can think of a recursive view as a persistent view definition that includes a retrieval from that same view.

Recursive views allow SQL queries to define a hierarchical search to find all possible paths in the data.

Suppose you need to determine all part numbers for all parts that are components at any level of a multicomponent part. This can also apply to a bill of materials, organization charts, family trees, the representation of discussion forum threads by a web or electronic mail client, transportation and communication networks, authorization graphs, document hierarchies, and software invocation structures.

For example, suppose you have the following table definition.

```
CREATE TABLE flights (
  source      VARCHAR(40),
  destination VARCHAR(40),
  carrier     VARCHAR(40),
  cost        DECIMAL(5,0));
```

After populating the table with data, its contents look like this.

| Flights | | | |
|----------|-------------|-------------------|------|
| Source | Destination | Carrier | Cost |
| Paris | Detroit | KLM | 7 |
| Paris | New York | KLM | 6 |
| Paris | Boston | American Airlines | 8 |
| New York | Chicago | American Airlines | 2 |
| Boston | Chicago | American Airlines | 6 |

| Flights | | | |
|---------|-------------|-------------------|------|
| Source | Destination | Carrier | Cost |
| Detroit | San Jose | American Airlines | 4 |
| Chicago | San Jose | American Airlines | 2 |

You must now solve the following problem: Find all the destinations that can be reached from Paris.

The most straightforward method to solve this class of problems is to use recursion, a class of algorithms characterized by self-reference and a terminal condition that can be either implicit or specified explicitly.

In SQL, recursive queries are typically built using these components.

- A non-recursive seed statement.
- A recursive statement.
- A connection operator. The only valid set connection operator in a recursive view definition is UNION ALL.
- A terminal condition to prevent infinite recursion.

Because this is a type of query that will be posed frequently, you decide to create a recursive view to handle it rather than performing a simple recursive query. For information about recursive queries, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146. The view definition is as follows.

```
CREATE RECURSIVE VIEW reachable_from (source,destination,depth) AS (
  SELECT root.source, root.destination, 0 AS depth
  FROM flights AS root
  WHERE root.source = 'Paris'
UNION ALL
  SELECT in1.source, out1.destination, in1.depth + 1
  FROM reachable_from AS in1, flights AS out1
  WHERE in1.destination = out1.source
  AND in1.depth <= 100);
```

Because the recursion is inherent in the definition of the view `reachable_from`, you can perform a simple SELECT statement on the view. In this recursive view definition, the following statement is the non-recursive or seed statement.

```
SELECT root.source, root.destination
FROM flights AS root
WHERE root.source = 'Paris'
```

The following query is the recursive component of the recursive view definition.

```
SELECT in1.source, out1.destination, in1.depth + 1
FROM reachable_from AS in1, flights AS out1
WHERE in1.destination = out1.source;
```

Notice that the recursive component refers to itself (*reachable_from* being the name of the recursive view of which it is the recursive component) in the FROM clause.

The UNION ALL set operator in the definition unions the results of the non-recursive and recursive query components to produce the view being defined.

Having defined the recursive view named *reachable_from*, you can run the following simple SELECT statement to find all the destinations that can be reached from Paris.

```
SELECT DISTINCT source, destination
FROM reachable_from;
```

The answer set is as follows.

| Source | Destination |
|--------|-------------|
| Paris | Detroit |
| Paris | New York |
| Paris | Boston |
| Paris | Chicago |
| Paris | San Jose |

In mathematics, a monotonic progression is one that either never increases (always decreases or remains the same) or never decreases (always increases or remains the same). For SQL, the definition of monotonicity is limited to a progression that never decreases. Explicitly, the number of rows in the working answer set never decreases: either the number stays the same or it grows larger as the recursion continues. Similarly, the field values of rows in the working answer set must never change.

One such example of violating monotonicity is the use of aggregates in a recursive query. For example, suppose you want to calculate the maximum price of a flight from Los Angeles to Boston using an iterative query. The first iteration computes a direct flight that costs 100 USD. The row in the result set has columns for origin, destination, and MAX(price). If a subsequent iteration produces a row with a new maximum value (such as LAX, BOS, \$200), then the row in the answer set must change because 100 USD is no longer the maximum cost for that particular trip. Changing a field value in this situation is a clear violation of monotonicity.

Various restrictions on SQL recursion ensure that *Q* can only increase monotonically.

One way to compute the fixpoint of query *Q* is the following procedure.

1. Start with an empty table, represented symbolically as $T \leftarrow \emptyset$.
2. Evaluate query *Q* over the current contents of table *T*.

| IF the query result is ... | THEN ... |
|----------------------------|---|
| identical to T | T is the fixpoint. |
| not identical to T | you have the $T \leftarrow$ query result, which is not the fixpoint. Go to Step 2. |

In other words, the fixpoint of a recursive query is reached when further efforts to identify more rows to insert into the result can find no such rows.

Based on the preceding explanation, a recursive query can be described as a query that is divided into the following processing phases.

1. Create an initial result set.
2. Perform a recursion that reaches a fixpoint on the initial result set.
3. Perform a final query on the result set derived from the recursion to return the final result set.

This algorithm shows one particular recursive query and its execution. The algorithm presented is designed to solve recursive queries, of which recursive views can be regarded as a special case (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for more information about recursive queries), of the following type.

```
WITH RECURSIVE reachable_from (destin, cost, depth) AS
  (SELECT root.destin, root.cost, 0 AS depth
   FROM flights AS root
   WHERE root.source = 'Paris'
  UNION ALL
   SELECT result.destin, seed.cost + result.cost, seed.depth + 1
     AS depth
   FROM reachable_from AS seed, flights AS result);
 WHERE Seed.Destin = Result.Source
 AND   Depth <= 20
 )
SELECT * FROM Reachable_From;
```

Note that depth checking code is not necessary for this particular query, but is presented as an example of a safe coding technique that avoids the possibility of infinite recursion, which is an issue you must always be aware of. See [Preventing Infinite Recursion Due To Cyclic Data](#) and [Preventing Infinite Recursion With Acyclic Data](#). Note that Vantage uses the smallest possible data type for any variable that is not cast to a specific type. This could present a problem if you were to set a depth counter that was initialized to a value of 0, but also specified a condition that permitted the recursion to loop to a depth of 128. The largest value the default data type for this counter can hold is 127 because when you initialize the variable to 0, the system assigns it a data type of BYTEINT by default.

To avoid this problem, you can explicitly cast the initial value of the counter to a data type of INTEGER, which supports far larger numbers than you could possibly require to control runaway recursion. See

Teradata Vantage™ - Data Types and Literals, B035-1143 for more information about default data types in the database.

The following pseudocode represents the previously mentioned algorithm for solving the recursion problem provided here, where the following assertions apply.

- T specifies the base relation named flights.
- The relations named *spool_1*, *spool_2*, *spool_3*, and *spool_4* specify the spools the system generates in the evaluation process.
- The first statement constitutes the initialization phase of the algorithm.
- The statements within the L1 block make up the recursion phase of the algorithm.
- The statement on L2 is third phase of the algorithm, where the last query returns the final result set.
- The contents of *spool_4* are the final result set that is returned to the user.

```

/* Insert all seed rows from the base relation into Spool_1. The function
/* function S retrieves all seed rows from the base relation Flights
/* (Root) */

Spool_1 = S (T);

/* The contents of Spool_1 result is appended to Spool_2 */

L1: Spool_2 = Spool_2 + Spool_1;

    If (NoRowsAppendedToSpool_2)

/* If no more rows are added in Spool_2, jump to L2 */

    GoTo L2;

/* Spool_1 and T are joined based on the join condition specified in */
/* the WHERE clause, e.g., */
/* WHERE Seed.Destin = Result.Source and Depth <= 20 */
/* and the result is appended to Spool_3. */

    Spool_3 = Spool_1 x T;
    Spool_1 = 0; /* Delete all rows from Spool_1 */
    Spool_1 = Spool_3; /* Append the contents of Spool_3 into Spool_1 */
    Spool_3 = 0; /* Delete all rows from Spool_2 */
    GoTo L1;

L2: Spool_4 = Spool_2 /* Append the contents of Spool_2 into Spool_4 */
    Spool_2 = 0; /* Delete the contents of Spool_2 */

```

Controlling the Cardinality Estimate For the Final Recursive Result Spool

The cardinality estimate of the final recursive result spool is controlled by factors established by Teradata support. If you think your site might benefit from a change to its value, consult with your Teradata support representative.

All Recursive View Definitions Must Be Linear

SQL does not support non-linear recursion.

When recursion is linear, invoking a recursive view produces at most 1 direct invocation of that recursive view. To enforce linearity, SQL restricts table references to a single occurrence in the recursive view definition, which can be specified either in the FROM clause of the recursive statement or in a subquery in the recursive statement, but not in both.

A recursion is said to be linear if it obeys the following rules.

- For every joined table in the recursive view definition, the recursively-defined table is referenced only once.
- The recursively-defined table is not referenced in both the FROM clause and in a subquery of the same query specification.
- The recursively-defined table is not referenced more than once in the FROM clause of the query specification.

If any of these conditions is not met, then the recursion is non-linear.

The following create text is an example of a non-valid attempt to create a recursive view definition. The definition is not valid because the recursive view Fibonacci is referenced more than once (using the column alias names *p* and *pp*, respectively) in the FROM clause of the recursively-defined relation in the view definition. As a result, it violates the rule stated in the third bullet.

```
CREATE RECURSIVE VIEW fibonacci (n, f, mycount) AS
  SELECT a, b, 0 AS mycount
  FROM t
  WHERE (a=0
  AND    b=0)
  OR    (a=1
  AND    b=1)
  UNION ALL
  SELECT n+1, p.f + pp.f, p.mycount + 1
  FROM fibonacci AS p, fibonacci AS pp
  WHERE (p.n - 1) = pp.n
  AND    p.mycount <= 100;
```

The following similarly conceived query that uses derived tables has the same problem.

```

CREATE RECURSIVE VIEW fibonacci (n, f, mycount) AS
  SELECT  n, f, mycount
  FROM (SELECT 0,0,0) AS a (n,f,mycount)
UNION ALL
  SELECT n, f, mycount
  FROM (SELECT 1,1,0) AS b (n, f, mycount)
UNION ALL
  SELECT n+1, p.f + pp.f, p.mycount + 1
  FROM fibonacci AS p, fibonacci AS pp
  WHERE (p.n - 1) = pp.n
  AND   p.mycount <= 100;

```

Building a Recursive View

The basic definition of a recursive view consists of these component classes.

- A non-recursive, or seed query class.
- A recursive query class.
- A UNION ALL set operator to connect the query components.
- A terminal condition to prevent infinite recursion.

A recursive view definition can contain multiple seed and recursive statements.

Normally, you should specify the terminal condition explicitly in the WHERE clause of the recursive statement, but such an explicit condition is not mandatory because recursive queries are implicitly limited by limits on user spool space and system disk space.

Syntax

```

CREATE RECURSIVE VIEW view_name [(column_name [, ...])] AS
  (query_expression_1 UNION ALL query_expression_2)

```

Syntax Elements

view_name

The name by which the view is referenced.

column_name

The name of a column in the base table.

query_expression_1

A seed statement.

query_expression_2

A recursive statement.

Example

The following example is based on the Flights table defined in [The Concept of Recursion](#).

```
CREATE RECURSIVE VIEW reachable_from (source,destination,depth) AS (
  SELECT root.source, root.destination, 0 AS depth
  FROM flights AS root
  WHERE root.source = 'Paris'
UNION ALL
  SELECT in1.source, out1.destination, in1.depth + 1
  FROM reachable_from AS in1, flights AS out1
  WHERE in1.destination = out1.source
  AND   in1.depth <= 100);
```

The *view_name* is `reachable_from`.

The base table is `flights`. The view has only base table columns `source`, `destination`, and `depth`.

query_expression_1 is:

```
SELECT root.source, root.destination, 0 AS depth
FROM flights AS root
WHERE root.source = 'Paris'
```

This is the seed statement (see [Seed Statement Component of the View Definition](#)) for the recursive view definition. The statement does not reference any recursive relations at any point in its definition.

query_expression_2 is:

```
SELECT in1.source,out1.destination,in1.depth + 1
FROM reachable_from AS in1, flights AS out1
WHERE in1.destination = out1.source
AND   in1.depth <=100
```

This is the recursive statement (see [Recursive Statement Component of the View Definition](#)) for the recursive view definition. The recursion occurs because the view name `reachable_from` is referenced in the FROM clause of this statement.

Stratification of Recursive View Evaluation

Recursive view evaluation is said to be stratified because the system evaluates the seed and recursive statement components (see [Seed Statement Component of the View Definition](#) and [Recursive Statement Component of the View Definition](#)) of a recursive view definition separately.

Specifying RECURSIVE For a Non-Recursive View

The system does not require a view defined explicitly with the keyword RECURSIVE to have a recursive reference. As long as such a view definition is otherwise correct syntactically, the system creates the view as specified without returning an error to the requestor. However, the resulting database object is not a recursive view.

This is analogous to the situation in which you explicitly define an outer join in your query, but, because the Optimizer evaluates the query text as if you had defined an inner join, and rewrites it accordingly. In that situation, the system makes an inner join on the specified tables, but does not return an error or a warning to the requestor. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for an extended analysis of the outer join specification issue.

Seed Statement Component of the View Definition

A seed statement is the non-recursive statement portion of a recursive view definition. Its purpose is to provide the initial row set to be used to build a recursive relation later in the process. To produce the desired recursion, the seed statement must produce 1 or more rows; otherwise, running the recursive query produces an empty result set.

By definition, there are no references to recursive relations anywhere in the seed statement. Taking the example developed in [The Concept of Recursion](#), the code highlighted in **bold** typeface is the seed statement for the definition.

```
CREATE RECURSIVE VIEW reachable_from (source,destination,depth) AS (
  SELECT root.source, root.destination, 0 AS depth
  FROM flights AS root
  WHERE root.source = 'Paris'
UNION ALL
  SELECT in1.source, out1.destination, in1.depth + 1
  FROM reachable_from in1, flights AS out1
  WHERE in1.destination = out1.source
  AND   in1.depth <= 100);
```

As you can see, there are no references to recursive relations within the seed query.

Recursive Statement Component of the View Definition

A recursive statement is the recursive statement portion of a recursive view definition. Its purpose is to cycle through recursively, building the eventual answer set. Unlike the seed statement component, the recursive statement component always references a recursive relation at least once, and possibly many times.

Taking the example developed in [The Concept of Recursion](#), the code highlighted in **bold** typeface is the recursive statement for the definition.

```

CREATE RECURSIVE VIEW reachable_from (source,destination,depth)
AS (
    SELECT root.source, root.destination, 0 AS depth
    FROM flights AS root
    WHERE root.source = 'Paris'
    UNION ALL
    SELECT in1.source, out1.destination, in1.depth + 1
    FROM reachable_from in1, flights AS out1
    WHERE in1.destination = out1.source
    AND in1.depth <= 100);

```

As you can see, there are references to the recursive view being defined, *reachable_from*, in the FROM clause of this statement as well as in its WHERE clause, in which the column names source, destination, and depth are referenced.

All forms of negation and all forms of aggregation are banned from recursive statements to ensure that monotonicity is never violated (see the description of fixpoint semantics in [The Concept of Recursion](#)).

You Can Specify Multiple Seed and Recursive Statements Within a Single Recursive View Definition

You can specify multiple seed and recursive statements within a single recursive view definition. The first statement specified must be a seed statement.

Although you can specify any number of seed or recursive statements within a single recursive view definition, the maximum SQL text size for the system is 1 megabyte.

The system evaluates the entire seed statement set at the beginning of the operation and then evaluates the entire recursive statement set on each iteration cycle, using the rows from the previous iteration as input. See [Building a Recursive View](#).

As an example, consider the following example view definition, which is used to find all cities that can be reached from the German city of Kaiserslautern either by train or by plane.

```

CREATE RECURSIVE VIEW tc (source, destination, carrier, depth) AS (
    SELECT f.source, f.destination, f.carrier, 0 AS depth
    FROM flights AS f                                -- Query_1
    WHERE f.source = 'Kaiserslautern'
    UNION ALL
    (SELECT r.source, r.destination, 'EuroRail', 0 AS depth
     FROM trains AS r                                -- Query_2
     WHERE r.source = 'Kaiserslautern')
    UNION ALL
    SELECT tcq.source, f.destination, f.carrier, tcq.depth + 1
    FROM tc AS tcq, flights AS f                    -- Query_3

```

```

WHERE tcq.destination=f.source
AND   tcq.depth <= 100
UNION ALL
SELECT tcq.source, r.destination, 'EuroRail', teq.depth + 1
FROM tc AS tcq, trains AS r                                -- Query_4
WHERE tcq.destination = r.source
AND   tcq.depth <= 100 ) );

```

In this example, *Query_1* and *Query_2* are both seed statements, while *Query_3* and *Query_4* are both recursive statements. The parentheses used to segregate *Query_1* from the other 3 queries are used to clarify the relationships, not to enforce execution order. Specifying parentheses within the recursive query do not affect execution order because the UNION ALL set operator is both associative and commutative.

Having defined the recursive view named *tc*, all you need to do to answer the question “find all cities that can be reached from the Kaiserslautern either by train or by plane” is to run the following simple SELECT statement.

```

SELECT *
FROM tc;

```

Recursive View Definition Restrictions

Three categorical sets of operators, clauses, and other miscellaneous features either cannot be used in the definition of a recursive view, or are restricted in their use in the definition.

The categories of restriction are as follows.

- Restrictions defined by the ANSI SQL:2011 SQL standard.
- Additional restrictions defined by Teradata.
- Miscellaneous restrictions.

The first set of feature restrictions is defined by the ANSI SQL:2011 standard. The restrictions defined by ANSI are fundamental to preventing recursive views from violating monotonicity (see [The Concept of Recursion](#)).

The ANSI restrictions are provided in the following list.

- The following conditional negation operators are valid within a seed statement.
 - NOT IN
 - NOT EXISTS

They are *not* valid when used within a recursive statement.

Negation is forbidden within recursive statements to avoid violations of monotonicity, which is a fundamental requirement of recursive views in SQL.

- Aggregation is valid within a seed statement.

It is *not* valid when used within a recursive statement.

A LEFT OUTER JOIN is valid when used within a seed statement.

Left outer joins are valid when used within a recursive statement only if the recursive view is the outer, or left, table specified in the outer join.

A RIGHT OUTER JOIN is valid when used within a seed statement.

Right outer joins are valid when used within a recursive query only if the recursive view is the outer, or right, table specified in the outer join.

A FULL OUTER JOIN is valid when used within a seed statement.

Full outer joins are valid when used within a recursive statement only if neither the inner nor the outer table in the outer join definition is the recursive view being defined or 1 of its component relations.

In other words, a full outer join used within a recursive query cannot cross recursion.

The ON clause of any join condition specified within a recursive view definition cannot specify a SAMPLE clause within a predicate subquery.

The following set operators are not valid when used within any component of a recursive view definition.

- EXCEPT ALL
- INTERSECT ALL
- MINUS ALL

The only valid set operator within a recursive view definition is UNION ALL.

The next set of feature restrictions is defined by Teradata. They are additional restrictions beyond those defined by the ANSI SQL:2011 standard. The Teradata restrictions are provided in the following list.

- Non-linear recursion is not supported (see [All Recursive View Definitions Must Be Linear](#)).
- Mutual recursion, the condition in which one recursive view references another recursive view, is not supported.
- The following rules and restrictions apply to the coding of derived tables within a recursive view definition.
 - Derived tables are valid within the seed statement of a recursive view definition.
 - Derived tables are *not* valid within the recursive statement of a recursive view definition.
 - You *cannot* code a WITH RECURSIVE or non-recursive WITH clause within the definition of a derived table.
 - You can reference a recursive view within the definition of a derived table.
- The following set operators are not valid when used within any component of a recursive view definition.
 - EXCEPT
 - EXCEPT ALL
 - INTERSECT
 - INTERSECT ALL
 - UNION
- Subqueries are valid within a seed statement.

They are *not* valid when used within a recursive statement of a recursive view definition.

- Ordered analytical and window functions are valid within a seed statement.

They are *not* valid when used within a recursive statement.

The following clauses are valid within a seed statement.

- CUBE
- GROUP BY
- GROUPING SETS
- HAVING
- ROLLUP

The HAVING clause of any condition specified within a recursive view definition cannot specify a SAMPLE clause within a predicate subquery.

They are *not* valid when used within a recursive request.

The DISTINCT operator is valid when used within a seed statement.

It is *not* valid when used within a recursive request.

- The TOP *n* and TOP *m* PERCENT clauses are not valid within a recursive view definition (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 and the CREATE VIEW/REPLACE VIEW statement in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information about the TOP *n* and TOP *m* PERCENT clauses).
- If a recursive view references multiple row-level security-protected tables, but the row-level security definitions are not the same for all of those tables. The system aborts the request and returns an error message to the requestor.

The last set of restrictions is a group of miscellaneous limitations, some of which are fundamental to the principles of recursion.

- To create a recursive view, the CREATE RECURSIVE VIEW statement must refer to its view name within the definition.

If the view definition does not refer to its own name, then the system creates a valid view, but it is not recursive (see [Specifying RECURSIVE For a Non-Recursive View](#) for details).

- Recursive views are not updatable.
- Because recursive views are not updatable, the WITH CHECK OPTION clause is not valid for recursive view definitions.
- The WITH RECURSIVE clause is not valid when used within any component of a recursive view definition (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for more information about the WITH RECURSIVE clause).
- You cannot reference a recursive query or recursive view definition more than once within a FROM clause.
- You cannot reference a non-recursive WITH clause within a CREATE RECURSIVE VIEW definition.
- References to another recursive view are not valid within a recursive view definition.

This is true at all levels of a view hierarchy, so you must be careful not to create a recursive view that references another recursive view at some deeper level of the view hierarchy.

Referential Restrictions For Recursive Views

You cannot refer to recursive views in the definitions of any of the following.

- Triggers
- Non-recursive views

Step Building Logic for Recursive Queries

The database parses recursive queries to identify the seed and recursive parts. The system defines the seed component to be all SELECT statements in the body of the recursive query that do not make any recursive references. By default, all that remains is defined to be the recursive component.

Consider the following recursive view definition as an example.

```
CREATE RECURSIVE VIEW tc (source, destin, carrier, depth) AS (
  SELECT f.source, f.destin, f.carrier, 0 AS depth
  FROM flights AS f
  WHERE f.source = 'Paris'
UNION ALL
  SELECT tcq.source, f.destin, f.carrier, tc.depth+1 AS depth
  FROM tc tcq, flights AS f
  WHERE tcq.destin=f.source
  AND   depth <= 100
UNION ALL
  SELECT tcq.source, r.destin, 'EuroRail', tc.depth+1 AS depth
  FROM tc AS tcq, trains AS r
  WHERE tcq.destin=r.source
  AND   depth <= 100
UNION ALL
  SELECT r.source, r.destin, 'EuroRail', 0 AS depth
  FROM trains AS r
  WHERE r.source = 'Kaiserslautern');
```

The seed query component of the view definition is highlighted in **boldface** text. The system treats all the rest of the SQL text as the recursive component of the definition.

Consider the following view definition as the foundation for examining how the Optimizer builds the steps to process a query made against it.

```
CREATE RECURSIVE VIEW reachable_from (destin, cost, depth) AS (
  SELECT root.destin, root.cost, 0 AS depth
```

```

FROM flights AS root
WHERE root.source = 'paris'
UNION ALL
SELECT result.destin, seed.cost + result.cost, seed.depth + 1
      AS newdepth
FROM reachable_from AS seed, flights AS result
WHERE seed.destin = result.source
AND   newdepth <= 20);

```

An EXPLAIN of the following query includes a RETRIEVE step from Root by way of the primary index "Root.Source = 'Paris'" JOIN step from Result with a join condition of ("Destin = Result.Source") AND "((Depth+1)<= 20)".

```

EXPLAIN SELECT *
FROM reachable_from;

```

In the hypothetical EXPLAIN report:

- Spool 1 in the plan is used to compute the result of the steps built for the seed query in a RETRIEVE step from Root by way of the primary index "Root.Source = 'Paris'" .
- Next, the seed query steps result are fed into Spool 2, which then contains the final result.
- The recursive steps of the query are performed in a JOIN step from Result with a join condition of ("Destin = Result.Source") AND "((Depth+1)<= 20)".

It uses Spool 1 as the seed result and joins that to flights.

- The result of that join, Spool 3, is then treated as the next seed.
Spool 3 feeds into Spool 1 which is used again by the seed query step building process.

Note that Spool 1 is cleared by marking it as "last use".

- The recursive steps are executed repeatedly until the result is empty.
- The retrieval of the rows from Spool 2 into Spool 4 produces the final result set that is returned to the requestor.

Preventing Infinite Recursion Due To Cyclic Data

Recursive views can recurse infinitely when there are cycles in the underlying data they are processing, as they can with acyclic data. See [Preventing Infinite Recursion With Acyclic Data](#). For example, there are likely to be many different routes one could take by means of a transportation system. Such cycles are very unlikely to appear in a bill of material problem, but the example below is presented as means of illustrating the issues.

For example, suppose you have the following parts table.

| Parts | | |
|-------|-------|----------|
| Major | Minor | Quantity |
| p1 | p2 | 2 |
| p1 | p3 | 3 |
| p1 | p4 | 2 |
| p2 | p3 | 3 |
| p3 | p5 | 2 |
| p3 | p1 | 3 |
| p4 | p2 | 4 |

You want to write a recursive view to process the data in this table, so you write the following recursive view definition.

```
CREATE RECURSIVE VIEW px (major, minor) AS (
  SELECT major, minor
  FROM parts
  WHERE major = 'p1'
  UNION ALL
  SELECT px.major, parts.minor
  FROM px, parts
  WHERE parts.major = px.minor);
```

Suppose you then ran the following SELECT request against the recursive view you have just defined, px.

```
SELECT major, minor
FROM px;
```

A quick analysis of the data indicates a loop, or cycle, in the data among parts *p1*, *p2*, and *p3* as follows.

- Part *p3* has part *p1* as a minor, or child, constituent.
- Part *p4* has part *p2* as a child constituent.
- Part *p2* has part *p3* as a child constituent.
- As noted in the first bullet, part *p3* also has part *p1* as a child constituent.

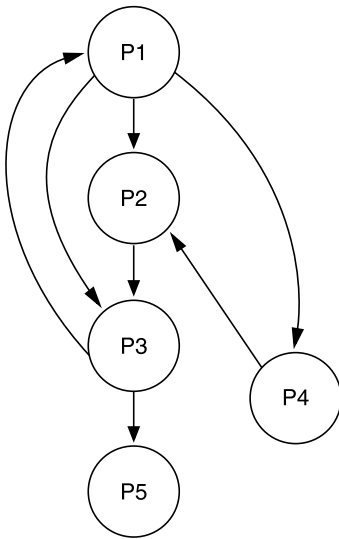
The cycles are perhaps easier to visualize when presented symbolically as follows.

p1 p2 p3 p1 p2 p3 and so on.

Similarly, there is a second cycle, as follows.

p1 p4 p2 p3 p1 p4 p2 and so on.

A graph of the hierarchy looks like this.



Consider the following informal definitions.

| Type of Graph | Description |
|---------------|---|
| cyclic | one or more of its nodes feeds back onto itself, either directly or indirectly. |
| acyclic | none of its nodes feeds back onto itself. |

It is only cyclic graphs that present the possibility for infinite recursion; acyclic graphs, by definition, cannot recurse infinitely.

In this example, node *p3* feeds back onto node *p1* and node *p4* feeds back onto node *p1* by traversing nodes *p2* and *p3*, so the graph is cyclic.

The cyclic relationships among these parts would cause an infinite recursion to occur, and, ignoring spool overflow, the query would loop forever if not stopped. Harkening back to [The Concept of Recursion](#), it could be said that the fixpoint of this recursion is.

So, what can be done to ensure that a runaway infinite recursion does not happen? After all, attempting to determine whether there are cycles in base table data would often require an extraordinary effort, and to little end.

One possible change to the definition for the example definition for the recursive view *px* is the following.

```

CREATE RECURSIVE VIEW px (major, minor, mycount) AS (
  SELECT major, minor, 0 AS mycount
  FROM parts
  WHERE major = 'p1'
  UNION ALL
  SELECT px.major, parts.minor, px.mycount + 1
  FROM px, parts

```

```
WHERE parts.major = px.minor
AND    px.mycount <= 40);
```

The AND condition `px.mycount <= 40` limits the recursion to 41 or fewer cycles. Forty-one because the condition is applied on `px.mycount`, which is initialized to 0, not 1.

Although not mandatory, good programming practice dictates that you always specify a constraint in the recursive query portion of a recursive view definition that limits the depth of recursion.

The following guidelines apply to coding a recursive view to control for infinite recursion by limiting recursion depth.

- Initialize a counter for each SELECT request in the seed query to a constant value. For most applications, the constant should be initialized to 0.
- Code each SELECT request in the recursive query to increment the counter by 1.
- Code each SELECT request in the recursive query to specify a condition that tests the value of the counter.

The form of this test condition should be one of the following.

- `counter < condition`
- `counter ≤ condition`

The value for condition is something that must be determined through prototyping or estimation. It is generally not possible to determine an optimum value without experimentation.

Without a limiting condition, runaway recursion is limited only by the quantity of spool space specified for the user who performs the query against the recursive view (see [CREATE USER](#) for information about assigning spool space to a user). Keep in mind that specifying a limiting condition does not eliminate the maximum quantity of spool space allotted to a user, so if you specify too large a value for the condition, a user can overflow its spool maximum without ever reaching the terminal value specified by that condition.

Preventing Infinite Recursion With Acyclic Data

Problems with infinite recursion are not restricted to cyclic data. A semantically incorrect query can also recurse indefinitely even if the data it queries is acyclic. Semantically incorrect join conditions that result in an infinite loop are a particular problem. A partial list of such conditions includes the following.

- Joining on the wrong columns.
- Selecting the wrong columns from the join.
- Using an OR operator instead of an AND operator with multiple join conditions.
- Giving a joined table a correlation name and then mistakenly referencing its original name in the join condition.

In this case, the system interprets the query as a specification to join 3 tables rather than 2. As a result, it then makes a cross join because there is no join condition for the “third” table.

- Specifying a tautological join condition.

The following example illustrates a tautological case that is limited only by the predicate `rec.depth <= 1000`. The tautology arises because the seed query WHERE condition returns only 1 row, having a `col_1` value of 2, while the recursive query WHERE condition returns only rows that have either the `col_1` value 2 or 3. Because every iteration of the recursion produces a row with a `col_1` value of 2, it would be an infinite loop were it not for the condition WHERE `rec.depth <= 1000`.

Suppose you have table *T* with the following data.

| T | |
|-------|-------|
| Col_1 | Col_2 |
| 2 | 3 |
| 4 | 5 |
| 6 | 7 |
| 8 | 9 |

You then create the following recursive view.

```
CREATE RECURSIVE VIEW rec (a, depth) AS (
  SELECT col_1, 0 AS depth
  FROM t
  WHERE col_1 = 2
  UNION ALL
  SELECT a, rec.depth + 1
  FROM t, rec
  WHERE t.col_1 IN (2, 3)
  AND   rec.depth <= 1000);
```

The following query on this view produces a result set with 1,001 rows, all having the value 2. The recursion stops only when the query reaches its limiting value. If no limit had been established, the query would have continued until it ran out of spool or disk space.

```
SELECT a
FROM rec;
```

The result set for the query looks like this, where the ellipsis character indicates 994 additional rows, each having the value 2.

| A |
|---|
| 2 |
| 2 |

| A |
|-----|
| 2 |
| 2 |
| 2 |
| ... |
| 2 |

The same problem occurs with the following recursive view definition, which is based on the same table *t*. This recursive view is also defined with a tautological search condition, but specifies an EXISTS clause in place of the IN clause specified in the previous example.

```
CREATE RECURSIVE VIEW rec (a, depth) AS (  
  SELECT col_1, 0 AS depth  
  FROM t  
  WHERE col_1 = 2  
  AND EXISTS  
    (SELECT col_1  
     FROM t)  
  UNION ALL  
  SELECT a, rec.depth + 1  
  FROM t, rec  
  WHERE rec.a = t.col_1  
  AND rec.depth <= 1000);
```

The following query on this recursive view produces the identical result set as the IN clause example: 1,001 rows, all having the value 2.

```
SELECT a  
FROM rec;
```

Breadth First and Depth First Searches

Hierarchical trees can be traversed either breadth first or depth first. These searches are abbreviated as indicated by the following table.

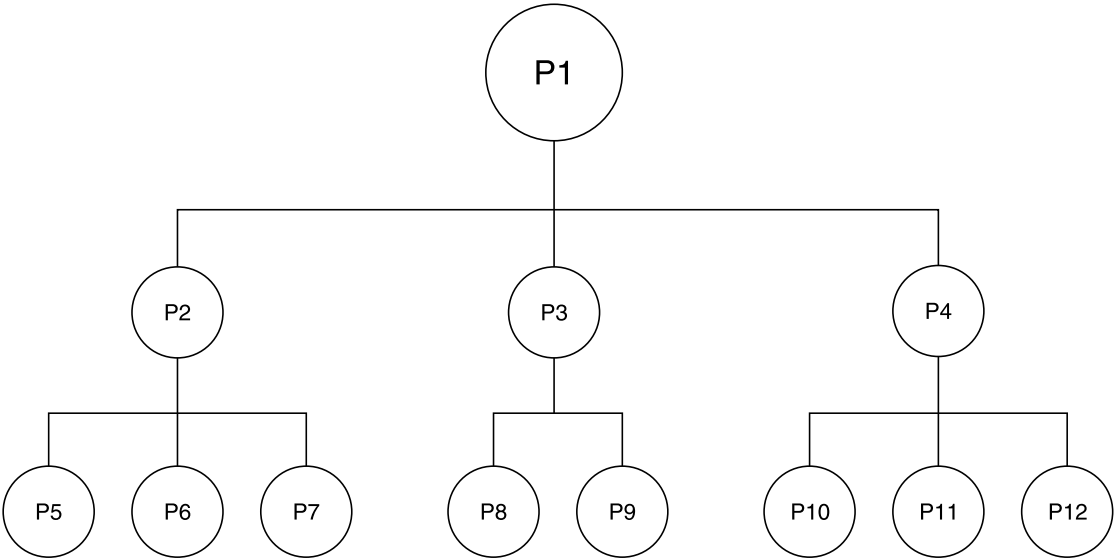
| Abbreviation | Definition |
|--------------|---|
| BFS | Breadth first search See Breadth First Reporting . |
| DFS | Depth first search |

| Abbreviation | Definition |
|--------------|---|
| | See Depth First Reporting . |

In a BFS, each immediate child of a node is visited before any of its grandchildren.

In a DFS, all descendants of each immediate node in the graph are visited before visiting other immediate children of that node.

Consider the following graph of a parts hierarchy.



If this graph is searched breadth first, its nodes are visited in the following order.

p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12

In contrast, if the graph is searched depth first, its nodes are visited in the following order.

p1 p2 p5 p6 p7 p3 p8 p9 p4 p10 p11 p12

The real effect of specifying a BFS versus a DFS is the ordering of the result set. As a result, it is simple to simulate a BFS or a DFS by simple ordering of the output.

Breadth First Reporting

You can simulate a BFS result by ordering the final result by the depth of the recursion. For example, consider the following view definition.

```
CREATE RECURSIVE VIEW reachable_from (destin, cost, depth) AS (  
  SELECT root.destin, root.cost, 0 AS depth  
  FROM flights AS root  
  WHERE root.source = 'paris'
```

```

UNION ALL
  SELECT result.destin, seed.cost+result.cost, seed.depth+1 AS depth
  FROM reachable_from AS seed, flights AS result
  WHERE seed.destin = result.source
  AND   depth <= 20);

```

By ordering the query result on the depth variable, the following SELECT request provides a breadth first report on this view.

```

SELECT *
FROM reachable_from
ORDER BY depth;

```

Depth First Reporting

You can simulate a DFS result by ordering the final result by the path of the recursion. The path in this context means how a row is found through recursive iterations. In this example, the path is the concatenation of the city names.

```

CREATE RECURSIVE VIEW reachable_from (destin,cost,depth,path) AS (
  SELECT root.destin, root.cost, 0 AS depth, destin AS path
  FROM flights AS root
  WHERE root.source = 'paris'
UNION ALL
  SELECT result.destin,seed.cost+result.cost,seed.depth+1 AS depth,
     seed.path||' '||result.destin
  FROM reachable_from AS seed, flights AS result
  WHERE seed.destin = result.source
  AND   depth <= 20);

SELECT *
FROM reachable_from
ORDER BY path;

```

Related Information

For more information about recursive queries and recursive views, refer to the following Teradata Orange Book.

- Nelly Korenevsky and Ahmad Ghazal, *Recursive Queries*, Teradata Orange Book 541-00004881B02, 2008.

CREATE TABLE Options

These topics provide supplemental usage information about the CREATE TABLE statement.

For CREATE TABLE syntax information and examples, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE TABLE

Block-Level Compression and Tables

Tables can be block compressed, depending on the DBS Control compression settings or whether the tables have been compressed or decompressed using the Ferret utility. For information about the DBS Control and Ferret utilities and how the block compression settings interact with other DBS Control file system settings, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Limits on data block sizes apply to the noncompressed size of a table. Block compression does not raise any of these limits, nor does it enable more data to be stored in a single data block than can be stored in an noncompressed data block of the same size.

Whether volatile table and global temporary tables are compressed or not depends on whether the Optimizer caches the relation in the request cache. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142. If a relation is cached, it is not compressed. If the relation is not cached, it is compressed.

Table Restrictions On Using UDT Columns

The general restrictions on the use of UDT columns in tables are the following.

- A table containing ARRAY, VARRAY, Period or Geospatial columns must have at least one non-Period or Geospatial column because an ARRAY, VARRAY, Period or Geospatial column cannot be a component of an index.
- The first column of the table cannot have a Geospatial, ARRAY, VARRAY, or Period data type if the PRIMARY KEY or UNIQUE attribute is not specified for any of the non-UDT columns.

An ARRAY, VARRAY, Geospatial, or Period column cannot be the primary index for a table, whether by default or by design.

- A UDT must have an ordering and transform group defined prior to its usage as a column type in any table. See [CREATE ORDERING and REPLACE ORDERING](#) and [CREATE TRANSFORM and REPLACE TRANSFORM](#).

Because ARRAY, VARRAY, Geospatial, and Period data types are implemented internally as UDTs, you do not have to define ordering and transform definitions for them.

See [Prerequisites for Specifying a UDT, ARRAY, VARRAY, Period, or Geospatial Column in a Table Definition](#).

- Table header space is reduced for each UDT, ARRAY, VARRAY, Geospatial, or Period column specified in a table definition.

Each UDT column declared consumes approximately 80 bytes of table header space. Depending on other options you declare for a table, including multivalued and algorithmic compression, partitioned primary indexes, secondary indexes, and so on, the table header size limit can eventually be exceeded, returning an error message to the requestor.

In the absence of other feature specifications, the limit on the number of UDT, ARRAY, VARRAY, Geospatial, and Period columns per table is approximately 1,600 columns. This assumes that your system permits fat table headers. For details about table headers, see *Teradata Vantage™ - Database Design*, B035-1094.

CREATE TABLE (Table Kind Clause)

Global Temporary Tables

Global temporary tables have a persistent definition but do not have persistent contents across sessions.

The following list describes characteristics of global temporary tables:

- Space usage is charged to the temporary space of the user.
A minimum of 4KB times the number of AMPs on the system of permanent space is also required to contain the table header for each global temporary table.
- A single session can materialize up to 2,000 global temporary tables at one time.
- You materialize a global temporary table locally by referencing it in a data manipulation request. To materialize a global temporary table, you must have the appropriate privilege on the base global temporary table or on the containing database or user as required by the request that materializes the table.
- Any number of different sessions can materialize the same table definition, but the contents are different depending on the DML requests made against the individual materialized tables during the course of a session.
- You can log global temporary table updates by specifying the LOG option for the CREATE TABLE statement. LOG is the default.
- You can save the contents of a materialized global temporary table across transactions by specifying ON COMMIT PRESERVE ROWS as the last keywords in the CREATE TABLE statement. The default is not to preserve table contents after a transaction completes (DELETE).
- The primary index for a global temporary table can be nonpartitioned or row-partitioned. The table can be defined without a primary index. See [Partitioned and Nonpartitioned Primary Indexes](#) and [Nonpartitioned NoPI Tables](#).

You cannot specify the following for global temporary tables:

- Referential integrity constraints
- Permanent journaling
- Column partitioning
- Primary AMP index
- Hash or join indexes

The database does not check privileges for the materialized instances of global temporary tables because those tables exist only for the duration of the session in which they are materialized.

See also [CREATE TABLE Global and Temporary](#).

For information about how block compression works with global temporary tables, see [Block-Level Compression and Tables](#). Also see *Teradata Vantage™ - Database Design*, B035-1094 and the DBS Control utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

Volatile Tables

The primary index for a volatile table can be nonpartitioned or row-partitioned. The table can also be defined without a primary index (NoPI). See [Partitioned and Nonpartitioned Primary Indexes](#) and [Nonpartitioned NoPI Tables](#).

The following options are not permitted for volatile tables:

- Referential integrity constraints
- CHECK constraints
- Permanent journaling
- DEFAULT clause
- TITLE clause
- Named indexes
- Column partitioning
- Primary AMP index
- Column with a DATASET data type that includes the WITH SCHEMA option

Otherwise, the options for volatile tables are the same as those for global temporary tables.

When you create an unqualified volatile temporary table, the login user space is used as the default database for the table, regardless of the default database that is currently specified.

When you create a qualified volatile table, you must specify the login user database. Otherwise, the system returns an error.

For information about how block-level compression works with volatile tables, see [Block-Level Compression and Tables](#). Also, see *Teradata Vantage™ - Database Design*, B035-1094 and the DBS Control utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

Volatile and Global Temporary Tables, Teradata Session Mode, and DML Performance

For sessions that use Teradata session semantics, be aware of certain critical performance issues with the LOG/NO LOG and DELETE/PRESERVE ON COMMIT options for global temporary and volatile tables. In Teradata session mode, each SQL request, such as DELETE, INSERT, and UPDATE, is treated as a transaction unless you explicitly define transaction boundaries with the BEGIN TRANSACTION and END TRANSACTION statements.

Because the implicit transaction semantics of Teradata session mode treat each SQL request as an individual transaction, a large number of all-AMPs operations can impact on overall CPU and Disk I/O performance unless you explicitly group DML requests into explicit transactions.

The following four volatile tables provide an example the performance issues associated with logging and committing various table update operations.


```

CREATE VOLATILE TABLE vt_ld LOG (
  column_1 INTEGER
  column_2 INTEGER)
ON COMMIT DELETE ROWS;

CREATE VOLATILE TABLE vt_lp LOG (
  column_1 INTEGER
  column_2 INTEGER)
ON COMMIT PRESERVE ROWS;

CREATE VOLATILE TABLE vt_nld NO LOG (
  column_1 INTEGER
  column_2 INTEGER)
ON COMMIT DELETE ROWS;

CREATE VOLATILE TABLE vt_nlp NO LOG (
  column_1 INTEGER
  column_2 INTEGER)
ON COMMIT PRESERVE ROWS;

```

For tables defined with NO LOG or ON COMMIT DELETE ROWS, each journaling (LOG) step makes an entry to empty the contents of its table in case of an abort. This is an all-AMPs operation.

Each deletion step deletes the contents of its table when a transaction ends. This is an all-AMPs operation that occurs for ON COMMIT DELETE ROWS tables.

Transient journaling steps for volatile and global temporary tables are generally generated once per table per transaction containing a DELETE, INSERT, or UPDATE operation.

Deletion steps for volatile and global temporary tables are generated once per each transaction that contains a DELETE, INSERT, or UPDATE operation.

For example, consider the following volatile table definitions.

```

CREATE VOLATILE TABLE vt1 (
  f1 INTEGER,
  f2 INTEGER)
ON COMMIT DELETE ROWS;

CREATE VOLATILE TABLE vt2 (
  f1 INTEGER,
  f2 INTEGER)
ON COMMIT DELETE ROWS;

CREATE VOLATILE TABLE vt3 (
  f1 INTEGER,

```

```
f2 INTEGER)
ON COMMIT DELETE ROWS;
```

For inserts to these tables using a multistatement request in Teradata session mode like the following, the overhead is only 1 transient journal step and 1 delete step.

```
INSERT vt1 (1,1)
;INSERT vt2 (1,1)
;INSERT vt1 (2,2)
;INSERT vt3 (1,1)
;INSERT vt2 (2,2);
```

For inserts to these tables using a single statement request in an explicit transaction like the following, the overhead is 1 journal step for each table and 1 delete step for all of them.

```
BT;
INSERT vt1 (1,1); ← 1 transient journal step for vt1
INSERT vt2 (1,1); ← 1 transient journal step for vt2
INSERT vt1 (2,2);
INSERT vt3 (1,1); ← 1 transient journal step for vt3
INSERT vt2 (2,2);
ET;                ← 1 delete step for all NO LOG or ON COMMIT
                   DELETE ROWS volatile tables
                   referenced in the transaction.
```

The deletes, inserts, and updates to the volatile tables result in following activity:

| Table Name | Transient Journal Step Performed? | Deletion Step Performed? | Number of All-AMPs Operations per Transaction |
|------------|-----------------------------------|--------------------------|---|
| vt_ld | Y | Y | 1 |
| vt_nld | Y | Y | 1 |
| vt_lp | N | N | 0 |
| vt_nlp | Y | N | 1 |

If the column labeled Transient Journal Step Performed? is yes, the operation is performed once per table per transaction.

If the column labeled Deletion Step Performed? is yes, the operation is performed once per transaction.

The following comparison of 200 individual Teradata session mode inserts are made to a permanent table and a volatile table. The volatile table was updated to add 200 inserts with ON COMMIT DELETE for:

- 200 implicit transactions.
- A single explicit transaction defined to encapsulate all 200 individual inserts.

| Table Type | Transaction Boundary Semantics | Sum (CPU Time) (seconds) | Sum (Disk I/O) |
|------------|--------------------------------|--------------------------|----------------|
| Permanent | Implicit | 13.49 | 5,761 |
| Volatile | Implicit | 48.50 | 31,889 |
| | Explicit | 6.03 | 4,702 |

These numbers are presented for comparison purposes only. Actual values depend on the configuration of your system. The ratios between values are comparable across configurations for volatile and global temporary tables. Note that the performance increment for the explicit transaction boundary case over implicit transactions is roughly an order of magnitude for CPU time and disk I/O measures where other factors are constant.

Resolving Table References

When encountering an unqualified table name, the system looks for a table by that name in all of the following databases:

- The default database
- Any databases referenced by the SQL request
- The logon user database for a volatile table by that name

The search must find the table name in only 1 of those databases. The database returns the “ambiguous table name” error message if the table name exists in more than 1 of those databases.

The following examples show the consequences of using unqualified table names with volatile tables.

- If the unqualified table name is a volatile table in the logon user database, but not a table in any other database, the reference defaults to the volatile table in the logon user database.

```
.LOGON u1

CREATE VOLATILE TABLE volatile_table_1 (
  f1 INTEGER,
  f2 INTEGER)
ON COMMIT PRESERVE ROWS;

DATABASE db1;

INSERT volatile_table_1 (1,1);
```

The reference here is to the volatile table, *u1.vt1*, in the *logon* user database, not the current *default* database, *db1*.

- If the unqualified table name is a volatile table in the logon user database and also a table in another database, an ambiguous table name error results, for example:

```
.LOGON u1

CREATE VOLATILE TABLE volatile_table_1 (
  f1 INTEGER,
  f2 INTEGER)
ON COMMIT PRESERVE ROWS;

DATABASE db1;

CREATE TABLE volatile_table_1 (
  g1 INTEGER,
  g2 INTEGER);

INSERT INTO volatile_table_1 VALUES (1,1);
```

The INSERT operation returns an error to the requestor because a table named *volatile_table_1* already exists in the logon user database, *u1*.

- If the unqualified table name matches a volatile table and a permanent table in the logon user database, the reference defaults to the volatile table.

This can occur if a user creates a volatile table in session 1 and then creates a permanent table by the same name in the user database in session 2. An unqualified reference to the table name in session 1 defaults to the volatile table. Session 2 does not see the volatile table in session 1, so an unqualified reference to the table name in session 2 defaults to the permanent table.

Table Kinds and UDT, Period, and Geospatial Data Types

The following table kind specifications all support UDT, ARRAY, VARRAY, Geospatial, and Period columns. Geospatial and Period data types are implemented internally as UDTs.

- SET
- MULTiset
- GLOBAL TEMPORARY
- VOLATILE

Global temporary trace tables do *not* support UDT columns. See [CREATE TABLE Global and Temporary](#).

Related Information

For information about the syntax used to specify table kinds, see the information about CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE TABLE (Table Options Clause)

Table Options and the ANSI SQL:2011 Standard

Table options are an extension to the ANSI SQL:2011 standard.

Specifying Data Protection

Specific protection for the data in a table is provided by the FALLBACK, JOURNAL, and CHECKSUM options.

FALLBACK causes a duplicate, or fallback, copy to be maintained by Vantage. The fallback copy is used if the primary copy becomes unavailable.

JOURNAL provides data protection through system-generated BEFORE- and AFTER-journals that contain the before and after images of data that changes as a result of any INSERT, UPDATE, or DELETE operations.

These journals are used by the database either to restore a table or to roll back the changes that were made to a table.

The CHECKSUM option provides disk I/O integrity checking of primary table data rows, fallback table data rows, and secondary index subtable rows.

Specifying FALLBACK

If you have specified FALLBACK in creating the table, either explicitly or by default, the system automatically maintains a duplicate copy of the data in the table. This fallback copy is then used if the primary copy becomes unavailable.

Fallback is very important when a system needs to reconstruct data from fallback copies when a single-bit read error occurs when it attempts to read the primary copy of the data. When a hardware read error occurs in this case, the file system reads the fallback copy of the rows and reconstructs a memory-resident image of them on their home AMP. Without this feature, the file system fault isolation logic would abort the transaction and, depending on the error, possibly mark the table as being down. See [SET DOWN and RESET DOWN Options](#).

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.

To avoid the overhead of this substitution, you must rebuild the primary copy of the data manually from the fallback copy using the Table Rebuild utility. For information about Table Rebuild, see *Teradata Vantage™ - Database Utilities*, B035-1102.

To enable the file system to detect all hardware read errors for tables, set CHECKSUM to ON.

To create a fallback copy of a new table, specify the FALLBACK option in your CREATE TABLE statement. If there is to be no fallback copy of a new table in a database or user space for which FALLBACK is in effect, specify NO FALLBACK as part of the CREATE TABLE statement defining the table. Alternatively, if you want to accept the database or user default for fallback, do not specify the option for the table.

Note:

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

Transient Journaling

The transient journal is a system-maintained dictionary table that provides a way to protect transactions from various system failures and from deadlock (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for a description of deadlock). Each transaction processed by Vantage records a before change image of rows that are touched by the transaction. Then if a transaction fails for some reason, the before change image of any modified rows can be retrieved from the transient journal and written over the modifications, returning the row to the same state it was in before the transaction occurred. Vantage removes all before change images from the transient journal after a transaction commits.

LOG and NO LOG

Global temporary and volatile tables permit you to define whether their activity is logged to the transient journal. While the NO LOG option reduces the system overhead of logging, it is also true that table modifications are lost and cannot be recovered upon an aborted SQL request.

Permanent Journaling

The permanent journal is a user-specified table that can be used to capture both before images and after images of database transactions. Journal entries in this table can be used to roll forward or roll back transactions during a recovery operation.

Assigning a Permanent Journal to a Table

To assign a permanent journal table to a new data table, the specified journal table must exist in the specified database. You can add or modify a permanent journal table by submitting appropriate MODIFY DATABASE or MODIFY USER statements. Then, you can resubmit your CREATE TABLE statement or submit an ALTER TABLE statement to assign the new permanent journal table to the existing data table.

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

Unless specified as part of the CREATE TABLE statement, fallback protection for the data in a new table defaults to that specified for the database or user in which the table is created.

Providing for a Permanent Journal

A permanent journal table is created and activated when the default JOURNAL table option and a journal table name are specified as part of the CREATE DATABASE, MODIFY DATABASE, CREATE USER, or MODIFY USER statement. See [CREATE DATABASE](#), [CREATE USER](#), [MODIFY DATABASE](#), and [MODIFY USER](#). The table resides in the database or user space that defined it. The specified JOURNAL parameters serve as the defaults for all the data tables created in that database or user space.

In defining a new table, you can override the established permanent journaling defaults by specifying the JOURNAL options as part of the CREATE TABLE statement. You can specify the journal table, the type of image (before-change, after-change, or both) and whether dual images are to be maintained for the created table, regardless of the option established when the containing database or user was created.

Whenever the structure of a table is altered, its version number increments. The following primary index changes qualify as changes to table structure:

- Modification to its partitioning
- Modification to its uniqueness
- Modification to its column set definition

As a result of changing the version number for a table, permanent journal rollforward and rollback operations from previous table versions are no longer valid.

You can activate permanent journaling for existing tables using the ALTER TABLE statement. See [ALTER TABLE \(Basic Table Parameters\)](#), in particular, [Activating Permanent Journaling](#).

Permanent journals are not supported for NoPI, column-partitioned, global temporary, or volatile tables.

FREESPACE PERCENT

FREESPACE = n PERCENT specifies the value for the percent freespace attribute, where n is an integer constant. n percent free space remains on each cylinder during bulk data load operations on the table. PERCENT is an unnecessary keyword and does not affect the semantics of the specification.

Note:

Not all database operations use the freespace percentage you specify in a CREATE TABLE statement. The following table lists which operations use the specified freespace percentage and which do not.

| Operations That Use FREESPACE | Operations That Do Not Use FREESPACE |
|---|---|
| <ul style="list-style-type: none"> • FastLoad data loads • MultiLoad data loads into unpopulated tables | <ul style="list-style-type: none"> • SQL INSERT operations |

| Operations That Use FREESPACE | Operations That Do Not Use FREESPACE |
|---|--|
| <ul style="list-style-type: none"> • Table Rebuild • System reconfiguration • Ferret PACKDISK • MiniCylPack If few cylinders are available, and storage space is limiting, MiniCylPack might not be able to honor the specified FREESPACE value. • ALTER TABLE statement that adds fallback protection to a table • CREATE INDEX statement 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. | <ul style="list-style-type: none"> • Teradata Parallel Data Pump INSERT operations • MultiLoad data loads into populated tables. |

The FREESPACE value returned in response to a SHOW TABLE statement is either the value specified in the CREATE TABLE statement for the table or the value specified by the most recent ALTER TABLE statement.

Note:

Do *not* use the Ferret utility PACKDISK command to change this value once you have created a table or have modified its FREESPACE PERCENT with an ALTER TABLE statement.

Instead, submit an ALTER TABLE statement to change the free space percent value for the table and then immediately afterward submit a PACKDISK command that specifies the same free space percent value you set with that ALTER TABLE statement. See [ALTER TABLE \(Basic Table Parameters\)](#). For information about running PACKDISK, see the Ferret utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

If you manually submit a PACKDISK command and specify a free space percentage on the command line, PACKDISK packs or unpacks the cylinders for the table to match the specified free space percentage. However, if the free space percentage you specify differs from the value specified for the table when it was created or last altered, Vantage starts a new AutoCylPack task immediately after PACKDISK completes, and this AutoCylPack operation nullifies the free space percentage you had just specified with PACKDISK to realign it with the free space percentage specified when you created or last altered the table.

MERGEBLOCKRATIO

When the size of a table only grows through SQL INSERT operations and bulk data loads and never becomes smaller, the average size of its data blocks varies from half their maximum up to the maximum size permitted according to the values you specify for the DATABLOCKSIZE option when you define the table. See [DATABLOCKSIZE](#).

However, the majority of tables fluctuate in size throughout their lifetime because of the mix of insert, update, and delete workloads that operate on them. When you delete rows from a table, its data blocks

can shrink to a size that is significantly smaller than half the defined maximum for the table. Once data blocks reach a size that is smaller than half their defined maximum, they remain small unless a roughly equal number of rows are added to the table. When a table has many such small data blocks, update and read operations against it must read a larger number of data blocks than would be necessary if those blocks were larger, and this causes the performance of updates to the table to become increasingly poor. The role of the merge block ratio is to enable the Teradata file system to merge such small data blocks into larger data blocks dynamically, but only when a full-table modification occurs on a table. Having fewer data blocks with a larger size reduces the number of I/O operations required to read a set of rows from disk.

The MERGEBLOCKRATIO option provides a way to combine existing small data blocks into a single larger data block during full table modification operations for permanent tables and permanent journal tables. This option is not available for volatile and global temporary files. The file system uses the merge block ratio that you specify to reduce the number of data blocks within a table that would otherwise consist mainly of small data blocks. It is not possible to define a general block size that is “small” in this context. The exact block size that fits this description is subjective and can differ based on the I/O performance of different systems, different workloads, and different numbers of active users.

For tables that are frequently modified by inserting new data, the average block size varies between 50% and 100% of their maximum supported multirow block size. This maximum supported size is defined as either the table-level attribute DATABLOCKSIZE or the system-level block size for the table as defined in the DBS Control record if you have not established a value for the DATABLOCKSIZE attribute for a table. The default percentage for the merge block ratio is 60, while its maximum value is 100. whether specified using the MERGEBLOCKRATIO option for CREATE TABLE or the DBS Control parameter MergeBlockRatio. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Note:

When you delete rows from a table, the blocks that previously held the deleted rows can become smaller than the sizes in the defined range, and their new sizes can vary considerably from the block size you specified when you created or last altered the table definition.

An extreme example of a block size distribution that benefits from having its data blocks merged together is a table whose blocks were created with an average size of 64 KB, but which have gradually shrunk or split over time to an average size of only 8 KB. If block splits were common for this table, its number of blocks can have increased by an order of magnitude. Requests rarely access every data block in a table.

If the rows in the table could be repackaged into blocks whose size were closer to the original average block size, you would probably experience improved response times for queries that read most or all of the blocks in the table because the number of logical or physical reads (or both) would then be reduced by an order of magnitude as well.

Merging blocks automatically is an enhancement of the functionality offered by the ALTER TABLE DATABLOCKSIZE option. See [ALTER TABLE \(Basic Table Parameters\)](#). Although this option can alleviate the problem of tables overrun with small data blocks, it also suffers from several drawbacks, including the following.

- Resolving the small data blocks problem requires a DBA to manually execute new SQL requests.

- You must determine which tables and block sizes are causing performance problems before you can submit the SQL requests necessary to resolve the problem. For example, table data block size statistics are available using the SHOWBLOCKS command of the Ferret utility or the equivalent SQL interface using the CreateFsysInfoTable and PopulateFsysInfoTable macros. For information on the Ferret utility, see *Teradata Vantage™ - Database Utilities*, B035-1102. For information on the CreateFsysInfoTable and PopulateFsysInfoTable macros, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.
- The operations performed by the problem resolving SQL request require an EXCLUSIVE table-level lock, which then blocks concurrent update activity on the table.
- If you specify the ALTER TABLE option IMMEDIATE, there is no way to prohibit some data blocks in the table from becoming smaller as soon as the table is updated. If you do not specify the ALTER TABLE option IMMEDIATE, some data blocks in the table might take a long time to grow larger again if they are not updated for some time. See [ALTER TABLE \(Basic Table Parameters\)](#).

The merge block ratio approach has none of these drawbacks and offers the following enhanced functionality, all of which is autonomous.

- Runs automatically without DBA intervention.
- Does not require the analysis of block size histograms.
- Does not require any specific AMP-level locking.
- Continuously searches for small data blocks to merge together even when some of those blocks are not being updated.

You can control the size threshold at which small data blocks are merged either by using the MERGEBLOCKRATIO option with a CREATE TABLE or ALTER TABLE statement or by changing the setting of the MergeBlockRatio DBS Control parameter. Note that you can set the merge block ratio percentage as low as 1% using either the MergeBlockRatio DBS Control parameter or the MERGEBLOCKRATIO option in an ALTER TABLE or CREATE TABLE statement. The default value for MERGEBLOCKRATIO is 60.

To disable data block merging for an individual table, specify the NO MERGEBLOCKRATIO option. To disable data block merging for all tables on your system, set the DBS Control parameter DisableMergeBlocks to TRUE.

Tables that are defined with a very small data block size do not gain a significant benefit from merging small data blocks because the sizes of most data blocks in such tables are close to the value established for a MERGEBLOCKRATIO, so there are few opportunities to merge data blocks in those tables.

DATABLOCKSIZE

DATABLOCKSIZE sets the maximum data block size for blocks that contain multiple rows. The data block is the physical I/O unit for the Teradata file system.

Larger block sizes enhance full table scan operations by selecting more rows in a single I/O. Smaller block sizes are best for transaction-oriented tables to minimize overhead by retrieving only what is needed.

A row that is larger than the DATABLOCKSIZE setting is stored in its own data block. Vantage does not split rows across data blocks.

A sector is normally 4 KB. If the computed value for the DATABLOCKSIZE attribute does not fall within the allowable range, the system returns an error message to the requestor.

If you do not specify a value for DATABLOCKSIZE, then the database uses the system-wide default data block size specified by the PermDBSize field in the DBS Control record. The default value is typically 254 sectors and the maximum is 255 sectors for all systems. For information about the DBS Control record, see *Teradata Vantage™ - Database Utilities*, B035-1102.

The default DATABLOCKSIZE settings depend on the cylinder size that is set for your system.

Following are release-specific rules for minimum data block sizes.

The DATABLOCKSIZE value returned in response to a SHOW TABLE statement is either the value specified in the CREATE TABLE statement for the table or the value specified in the most recently entered ALTER TABLE statement.

For additional information about the data block size, see:

- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Database Design*, B035-1094
- *Teradata Vantage™ - Database Utilities*, B035-1102

Large-Cylinder Systems

Systems initialized on Teradata Database 13.10 or later release use large cylinders. For large-cylinder systems, the tables below list the minimum and maximum data block sizes.

| Minimum Data Block Size | Description |
|-----------------------------|---|
| 42 sectors (21504 bytes) | The minimum data block size accepted by the CREATE TABLE and ALTER TABLE SQL statements is 21248 bytes (41.5 sectors). Tables created or altered to use this minimum size will have the value rounded up by the parser to 42 sectors, so these tables will have an actual minimum data block size of 21504 bytes. |

| Maximum Data Block Size | Description |
|---------------------------------|---|
| 2047 sectors (1048064 bytes) | The maximum data block size accepted by the CREATE TABLE and ALTER TABLE SQL statements is 1048319 bytes (2047.5 sectors). Tables created or altered to use this maximum size will have the value rounded down by the parser to 2047 sectors, so these tables will have an actual maximum data block size of 1048064 bytes. |

Small-Cylinder Systems

Systems initialized on a Teradata Database release prior to 13.10 and subsequently upgraded without running the Sysinit utility use small cylinders. For small-cylinder systems, the tables below list the minimum and maximum data block sizes.

| Minimum Data Block Size | Description |
|---|---|
| 18 or 19 sectors (9216 bytes or 9728 bytes) | The minimum data block size accepted by the CREATE TABLE and ALTER TABLE statements is 8960 bytes (17.5 sectors). Tables created or altered to use this minimum size will have the value rounded up by the parser to 18 sectors, so these tables will have an actual minimum data block size of 9216 bytes. |

| Maximum Data Block Size | Description |
|----------------------------|---|
| 512 sectors (262144 bytes) | The maximum data block size accepted by the CREATE TABLE and ALTER TABLE SQL statements is 262399 bytes (512.5 sectors). Tables created or altered to use this maximum size will have the value rounded down by the parser to 512 sectors, so these tables will have an actual maximum data block size of 262144 bytes. |

BLOCKCOMPRESSION

Use this option to set the temperature-based block compression state of a table.

Teradata Virtual Storage tracks data temperatures at the level of cylinders, not tables, and the file system obtains its temperature information from Teradata Virtual Storage, managing temperature-related compression at the cylinder level. See *Teradata Vantage™ - Teradata® Virtual Storage*, B035-1179.

Teradata Virtual Storage tracks the cylinder temperature metrics for all tables when temperature-based block-level compression is enabled. Temperatures are in effect system-wide, and Teradata Virtual Storage takes all temperatures into account when determining which data on the system is associated with a specific temperature.

The file system determines whether a table is using temperature-based block-level compression and applies selective compression or decompression based on the cylinder temperatures. For example, assuming that the DBS Control parameter TempBLCThresh is set to COLD, the threshold defined for COLD applies to all tables in the system. If the cylinders of an AUTOTEMP table are in the defined COLD band, the cylinders are eligible for block-level compression. If temperature-based block-level compression is disabled but block-level compression is enabled, the database manages AUTOTEMP tables in the same way as MANUAL tables.

The file system cannot compress the cylinders of a MANUAL table using either ordinary block-level compression or temperature-based block-level compression. Instead, you must use the Ferret commands COMPRESS or UNCOMPRESS to change the compression status of an existing MANUAL table or subtable manually. You can also use the BlockCompression query band to load rows into new MANUAL tables or subtables using the appropriate block-level compression.

You can activate temperature-based block-level compression for a new MANUAL table by using an ALTER TABLE statement to change its BLOCKCOMPRESSION definition from MANUAL to AUTOTEMP. You can also use one of the following TVSTemperature query bands to load data into a new AUTOTEMP table or subtable.

- TVSTEMPERATURE_PRIMARY
- TVSTEMPERATURE_PRIMARYCLOBS
- TVSTEMPERATURE_FALLBACK
- TVSTEMPERATURE_FALLBACKCLOBS

You can also use the Ferret commands COMPRESS and UNCOMPRESS to manually change the state of an AUTOTEMP table. Ferret returns a warning message when you change these parameters. For information about the Ferret utility, see *Teradata Vantage™ - Database Utilities*, B035-1102.

If the compressed state of the data does not match its temperature, the block-level compression changes you make using Ferret can be undone by the file system over time if the BLOCKCOMPRESSION option for the table is set to AUTOTEMP. Because of this, the best practice is not to use temperature-based block-level compression for a table determined to require compression consistency for the entire table.

The same set of subtables that can be compressed with ordinary block-level compression are compressed with temperature-based block-level compression.

- Primary and fallback base tables
- Primary and fallback CLOB data subtables

The following table lists the available options for BLOCKCOMPRESSION.

| Option | Description |
|----------|--|
| AUTOTEMP | <p>AUTOTEMP identifies tables whose block-level compression should be automatically managed based on the temperature of their data. The file system can change the block-compressed state of the data in the table at any time based on its Teradata Virtual Storage temperature.</p> <p>AUTOTEMP tables normally exist in a mixed compression state because temperature-based block-level compression does not directly apply compression to an entire table all at once. As various cylinders within a table grow warmer or colder over time, the file system either compresses or decompresses those cylinders as is appropriate for their Teradata Virtual Storage temperature.</p> <p>The cylinders in an AUTOTEMP table become eligible for temperature-based block-level compression or decompression only when they reach, become lower than, or exceed the defined threshold for the specified TempBLCThresh option.</p> |
| | <p>For all of the data in a table to be block compressed (or decompressed) at once, Teradata Virtual Storage must become aware that all cylinders in the table had reached or exceeded the threshold for the specified TempBLCThresh option. This would only occur if all of the cylinders in the table had not been accessed in some time and had been classified as COLD with respect to all other cylinders for compression or as HOT for decompression.</p> <p>AUTOTEMP tables normally exist in a mixed compression state because temperature-based block-level compression does not directly apply compression to an entire table all at once. As various cylinders within a table grow warmer or colder over time, the file system either compresses or decompresses those cylinders as is appropriate for their Teradata Virtual Storage temperature.</p> <p>Teradata Virtual Storage tracks the cylinder temperature metrics for all tables when temperature-based block-level compression is enabled at the level of cylinders, not tables. Because the file system obtains its temperature information from Teradata Virtual Storage, it also handles temperature-related compression at cylinder level. Temperatures are in effect system-wide, and Teradata Virtual Storage takes all temperatures into account when it determines which data on the system is associated with which temperature.</p> |

| Option | Description |
|---------|--|
| | <p>Once that determination has been made, the file system judges whether a table is using temperature-based block-level compression and if so, applies selective compression or decompression based on the cylinder temperatures. For example, assuming that the DBS Control parameter TempBLCThresh is set to COLD, the threshold defined for COLD applies to all tables in the system, so if an AUTOTEMP table cylinders are in the COLD band, they are eligible for block-level compression.</p> <p>Temperature-based thresholds for the block-level compression of AUTOTEMP tables work as follows.</p> <ul style="list-style-type: none"> • If data blocks are initially block-level compressed and then become warmer than the defined threshold for compression, the file system decompresses them. • If data blocks are initially not block-level compressed and then become colder than the defined threshold for decompression, the file system compresses them. <p>If temperature-based block-level compression is disabled but block-level compression is enabled, the database treats AUTOTEMP tables the same as MANUAL tables.</p> <p>You can still issue TVSTemperature 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 file system over time. Because of this, the best practice is not to use temperature-based block-level compression for a table that you think requires compression consistency for the entire table.</p> |
| DEFAULT | <p>DEFAULT identifies tables whose temperature-based block-level compression is determined by the DBS Control parameter DefaultTableMode. For more information, see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</p> <p>Note:</p> <p>The value of DefaultTableMode is not saved in the table definition as part of a CREATE TABLE or ALTER TABLE statement, so a table set to BLOCKCOMPRESSION=DEFAULT is affected by any future change to the DefaultTableMode parameter.</p> |
| MANUAL | <p>MANUAL identifies tables that are not managed automatically like AUTOTEMP tables are. You can specify a TVSTemperature query band option to determine the block-level compression state of the data before you load rows into an empty table.</p> <p>Once a table is populated, the file system does not change the compressed state of the data in the table unless you take specific action to do so using Ferret commands.</p> <p>Note:</p> <p>You can use the non-temperature-related BlockCompression query band to load an empty table with non-temperature-related data to block-level compress.</p> |
| NEVER | <p>NEVER identifies tables that should never be temperature-based block-level compressed or decompressed, even if a query band or the applicable DBS Control parameter defaults indicate otherwise. The file system does not compress or decompress table or subtable data even if the DBS Control block-level compression settings indicate otherwise.</p> <p>This means that the database rejects Ferret commands to manually compress table data, but Ferret commands to decompress table data are valid.</p> |

A table is fully functional in a mixed block-level compression state, but with inconsistent block compression. Best practice is not to use the AUTOTEMP option or any form of temperature-based block-level compression for a table that requires compression consistency for the entire table.

If you submit an ALTER TABLE statement to change the BLOCKCOMPRESSION option for a table to MANUAL, when you modify the block-level compressed blocks of the table, the newly created blocks are compressed. If the table was changed to NEVER, when you modify the compressed blocks of the table, the file system decompresses the newly created blocks. If you modify the noncompressed blocks of the table and have changed the BLOCKCOMPRESSION option to either MANUAL or NEVER, newly created blocks remain noncompressed.

For tables that are not defined with BLOCKCOMPRESSION=AUTOTEMP, you must control their block-level compression states yourself using Ferret commands or, if a table is not populated with rows, you can use one of the TVSTemperature query bands to specify the type of block-level compression to use for the newly loaded rows. If temperature-based block-level compression is disabled but block-level compression is enabled, the database treats AUTOTEMP tables the same as MANUAL tables.

For all of the data in a table to be block compressed or decompressed at once in an AUTOTEMP table, Teradata Virtual Storage must recognize that all cylinders in the table have reached the threshold specified by the DBS Control parameter TempBLCThresh. For example, suppose the threshold value for TempBLCThresh is set to WARM.

| All Cylinders in the Table | Eligibility |
|---|--------------------------|
| Reach or fall below the WARM or COLD thresholds | Block-level compression. |
| Reach or exceed the HOT threshold | Decompression. |

For information about the actions necessary to make the block compression consistent, see [BLOCKCOMPRESSION](#).

You can combine multivalue compression, algorithmic compression, 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.

Performance Aspects of Merging Data Blocks

Merging small data blocks into larger data blocks only affects the performance of data modification workloads that scan an entire table, so there is no direct effect on SELECT statements. However, because there are fewer data blocks to scan after data block merges complete, there is an indirect positive effect on the performance of all future full-table scan and full-table modification requests. This occurs because full-table scan operations are mostly I/O bound, and reducing the number of data blocks in the table has a direct effect on the I/O load.

Merging data blocks can have both positive and negative performance effects for table modification operations, though the negative effects should be negligible once the majority of the small data blocks in a table have been merged into larger data blocks.

For example, consider a table that has overly small data blocks. This generally occurs either because of a reconfiguration or because you have deleted a substantial number of rows from the table.

When you submit the first full-table modification request on this table after data block merging has been enabled, there is a higher cost to merging data blocks than there would have been if the feature had not been enabled. This occurs because of all the work required to merge large number of small data blocks.

When you submit a second full-table modification request on this table, it could either be more expensive or less expensive than the same operation would have been without data block merging enabled. The factors involved in this uncertainty of this expense are the following.

- The operation might be *more* expensive if there are a number of data blocks remaining that were not merged during the first full-table modification.
- The operation might be *less* expensive if a substantial number of data blocks were merged during the first full-table modification.
- The cost of the operation might be a mix of both of these factors.

These factors continue to apply until you have performed enough full-table modifications that the table reaches a point where the majority of its data blocks do not need to be merged. At this point, only the second bullet in the previous list applies, and any full-table modifications should continue to perform significantly better than they would have if you had never enabled data block merges.

Tables with the smallest number of data blocks should provide the best performance during full-table scan modification operations that are mostly I/O bound.

Full-table modifications that are done early in the life of a table with data block merging enabled tend to demonstrate a negative I/O performance impact, as noted in the first bullet because more data blocks are likely to be written or read (or both) from disk more frequently than you would see if this feature were not enabled. However, later full-table scan modifications of such a table are likely to show improvements in I/O operations because the table has a smaller number of blocks to scan or modify, as noted by the second bullet. You should see similar behavior for index subtable updates (with the exception of NUSI subtable updates, which are not affected by merging smaller data blocks into a larger data block).

You also should not see a performance impact for tables on which the principal operation is to append new data to the end of a subtable, because this feature does not scan or update sets of data blocks in the parts of a table that would not otherwise be scanned by the underlying update request.

No more than 8 data blocks can be merged during a single data block merge operation. By default, the resulting merged data block cannot be greater than or equal to 60% of the maximum multirow data block size for a table. This percentage threshold defines the merge block ratio for a table.

The performance of operations on tables that are defined with a minimal block size is not enhanced by merging small data blocks. This is because there are fewer opportunities to merge data blocks in such tables given that the size of most of their data blocks is already close to the merge block ratio threshold. Furthermore, it is unlikely that a database schema that is carefully designed to process its heaviest workloads would be affected by merging smaller data blocks anyway, because you will have already tuned the block sizes of your tables to optimize the most common update operations that common workloads make against them.

You can adjust the merge block ratio for an individual table by modifying the value for MERGEBLOCKRATIO using an ALTER TABLE statement. See [ALTER TABLE \(Basic Table Parameters\)](#).

You also should not see a significant performance impact for INSERT and INSERT ... SELECT operations into unpopulated tables because in such a case you are only creating new rows and there are no preexisting blocks to examine for merging. However, INSERT ... SELECT operations into *populated* tables are likely to experience degraded performance for the initial series of update operations on the table. But this should be offset by improved performance for later series of update operations.

Note that when an INSERT ... SELECT operation occurs for the case where the source and destination tables have different primary indexes, the rows must be redistributed across the AMPs, so the operation does not proceed smoothly from the beginning of the table to its end. Because this case does not simply append rows to the end of a table, it is possible that you might see degraded performance.

If you find that merging small data blocks for a large number of tables is exerting an unwieldy performance burden on your system, it is possible to deactivate the feature globally by changing the setting of the DisableMergeBlocks DBS Control parameter from its default value of FALSE to TRUE rather than performing a large number of ALTER TABLE statements to eliminate the merge block ratio value on a table-by-table basis. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Table Options and UDT, ARRAY, VARRAY, Geospatial, and Period Data Types

The table options are supported for UDT, ARRAY, VARRAY, Geospatial, and Period columns. Geospatial, Period, ARRAY, and VARRAY data types are implemented internally as UDTs.

CREATE TABLE (Column Definition Clause)

About Table Column Elements

This clause defines the table column elements.

You must specify a name and data type for each column defined for a table. When you create a table copy using the CREATE TABLE ... AS syntax, you either carry over the source table column types by default or specify new column data types using a query expression. Each data type can be further defined with one or more attribute definitions.

You can specify the following optional column attributes:

- Data type attribute declaration
- Column nullability
- Multivalue and algorithmic compression
- Column storage attributes
- Output format attributes
- Default value attributes

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for details about the preceding list of attributes.

- Column constraint attributes
 - PRIMARY KEY constraint definition
 - UNIQUE constraint definition
 - FOREIGN KEY ... REFERENCES constraint definition
 - CHECK constraint definition
 - Row-level security constraint assignments.

You can also specify PRIMARY KEY, UNIQUE, FOREIGN KEY, and CHECK constraints as table attributes, but you cannot specify a row-level security constraint as a table attribute.

Data Type, Column Storage, and Constraint Attributes

Column storage attributes are a Teradata extension to ANSI SQL. 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. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102. Explicit formatting applies to the parsing and to the retrieval of character strings.

If you change the `tdlocaledef.txt` file and issue a `tpareset` command, the new format string settings affect only those tables that are created after the reset. Existing table columns continue to use the existing format string in `DBC.TVFields` unless you submit an `ALTER TABLE` statement to change it. For more information, see [ALTER TABLE \(Basic Table Parameters\)](#).

Not all data type attributes are supported for UDT, ARRAY, VARRAY, Geospatial, and Period column declarations, and no column storage or column constraint attributes are supported for those types.

You can specify server character sets for character data by specifying CHARACTER SET *server_character_set* for character columns.

The database supports the following server character sets.

- LATIN
- UNICODE
- KANJISJIS
- GRAPHIC

You cannot specify the server character set KANJI1.

The database automatically converts any existing character columns with a server character set of Kanji1 to a server character set of Unicode.

Null Handling for UDT, ARRAY, VARRAY, Geospatial, and Period Columns

To support null attributes for structured UDT, ARRAY, VARRAY, Geospatial, or Period columns, or if a UDT, Geospatial, or Period column has a map ordering routine that can return nulls, CREATE TABLE or ALTER TABLE statements that specify that type should specify the NOT NULL attribute for that UDT, ARRAY, VARRAY, Geospatial, or Period column.

If nulls are permitted in the column and either the map ordering routine or the system-generated observer method for the type can return nulls, then a situation exists that is similar to the situation in which queries are executed against a column attributed as NOT CASESPECIFIC. In this situation, for some queries, the results returned for identical requests can be nondeterministic, varying from query to query.

The ordering routine for a UDT determines whether or not column values are equal and also determines the collation order for sort operations.

If you do not follow this recommendation, then it is possible for the following cases to be treated equally:

- A column null.
- A structured type that contains null attributes and whose map or observer routine returns nulls.

Sometimes a column null is returned in the result set and other times the non-null structured type that contains null attributes is returned in the result set.

The following simple example indicates how you should specify the NOT NULL attribute for UDT columns in a table definition.

```
CREATE TABLE udtTable (
  id          INTEGER,
  udtColumn   myStructUdtWithNullAttributes NOT NULL);
```

Prerequisites for Specifying a UDT, ARRAY, VARRAY, Period, or Geospatial Column in a Table Definition

A UDT must have both an ordering and a transform defined for it before you can use it as the column data type of any table. Because ARRAY, VARRAY, Geospatial and Period data types are predefined as internal Teradata UDTs, their ordering and transforms are also predefined. If you attempt to create a table that contains a UDT column when that UDT does not yet have either an ordering or transform definition, the system returns an error to the requestor.

Not all the possible Vantage column attributes are valid when specified for a UDT column. For a list of the valid data type attributes for UDT, Geospatial, and Period columns, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Table Size and Maximum Number of Columns Per Table

A CREATE TABLE statement can define up to 2,048 columns.

A base table can have a maximum of 32 LOB columns. This limit includes both predefined LOB columns and LOB UDT, Period, and Geospatial columns. A LOB UDT column counts as 1 LOB column even if the type is a structured UDT that has multiple LOB, Period, or Geospatial attributes.

BLOB, CLOB, Period, and Geospatial columns cannot be a component of any index. Unless at least 1 non-LOB column is constrained as a PRIMARY KEY or as UNIQUE when a primary index, primary AMP index, or no primary index is not explicitly specified, the first column defined for the table cannot have a BLOB, CLOB, Period, or Geospatial data type. This restriction is enforced because, for this particular set of circumstances, Vantage assigns the first column specified in a table definition to be the primary index.

Identity Columns

Identity columns are used mainly to ensure row uniqueness by taking a system-generated unique value. They are valuable for generating simple unique indexes and primary and surrogate keys when composite indexes or keys are not desired.

Identity columns are also useful for ensuring column uniqueness when merging several tables or to avoid significant preprocessing when loading and unloading tables.

You should not create an identity column for a table that you intend to use with Teradata Unity. Teradata Unity provides its own mechanism for generating deterministic unique values that do not present the problems that identity columns do for managing multiple database instances. See the Teradata Unity documentation.

Note:

Loading a row from a client system that duplicates an existing row in an identity column table is permitted because the assignment of the identity column to the row makes it unique. If this presents a problem, you must filter your data for duplicates before you load it into an identity column table. See [Identity Columns, Duplicate Column Values, and Duplicate Rows](#) for further duplicate value and duplicate row issues for identity column tables.

The various parameters for identity columns are maintained by the *DBC.IdCol* system table.

Identity columns have the following properties.

- Begin with the value 1 unless a different START WITH value is specified.
- Increment by 1 unless a different INCREMENT BY value is specified.
Can decrement if a negative INCREMENT BY value is specified.
- Can specify maximum values for incrementing or minimum values for decrementing a value series.
- Values can be recycled.

Identity columns have the following rules and restrictions.

- Cannot be used for tables that are managed by Teradata Unity.
- Support the following bulk insert operations only.
 - Single statement INSERT statements through multiple concurrent sessions.
For example, parallel BTEQ imports into the same table.
 - Multistatement INSERT statements through multiple concurrent sessions.
For example, Teradata Parallel Data Pump inserts.
 - INSERT ... SELECT statements.
- If a triggered event is an INSERT into a table with an identity column, then the triggered action statement block or the WHEN clause of the trigger cannot reference the identity column.
- GENERATED ALWAYS columns cannot be updated.
- GENERATED ALWAYS column values that also specify NO CYCLE are always unique.
- GENERATED BY DEFAULT columns generate a value only when the column is set null by an INSERT statement in one of the following ways.
 - Explicit specification of NULL in the multivalue.
For example, INSERT INTO table VALUES (1,NULL);
 - Implicit specification of NULL by the omission of a value in the multivalue when no column name list is provided.
For example, INSERT INTO table VALUES (1,);
 - Omission of a column name from the column name list.
For example, INSERT INTO table (x) VALUES (1);

where table has more than a single column.

- GENERATED BY DEFAULT columns can be updated.
- GENERATED BY DEFAULT column values are not guaranteed to be unique.

You must comply with all of the following restrictions if you want to guarantee the uniqueness of GENERATED BY DEFAULT column values.

- You must specify NO CYCLE.
- Any user-specified values you specify must be outside the range of any system-generated values.
- You must enforce the uniqueness of the user-specified values yourself.
- Identity columns cannot have a UDT, Geospatial, ARRAY, VARRAY, or Period data type.
- GENERATED ALWAYS identity columns cannot be null.
- The cardinality of a table with unique identity column values is limited by the maximum value for the identity column data type.

Because of this, you should specify a numeric type that ensures the largest possible number of unique values for the identity column can be generated.

The maximum numeric data type ranges are DECIMAL(18,0), NUMERIC(18,0), and exact NUMBER(18,0), or approximately 1×10^{18} rows. You cannot use approximate NUMBER columns for identity columns.

This is true even when the DBS Control parameter MaxDecimal is set to 38. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143 and *Teradata Vantage™ - Database Utilities*, B035-1102. You can define an identity column with more than 18 digits of precision, or even as a large fixed NUMBER or BIGINT type, without the CREATE TABLE or ALTER TABLE statement returning a warning message, but the values generated by the identity column feature remain limited to the DECIMAL(18,0) type and size.

- Inserts into GENERATED BY DEFAULT identity columns using Teradata Parallel Data Pump cannot reuse Teradata Parallel Data Pump field variables in another parameter of the same insert operation.

The following examples are based on this table definition.

```
CREATE MULTISET TABLE test01 (
  a1 INTEGER GENERATED BY DEFAULT AS IDENTITY
    (START WITH 1
     INCREMENT BY 20
     MAXVALUE 1000
    ),
  a2 INTEGER);
```

The following simple INSERT statements fail because they reuse Teradata Parallel Data Pump field variables.

```
.LAYOUT layoutname;
.FIELD uc1 INTEGER;
.FIELD uc2 INTEGER;
.DML LABEL labelname;

INSERT INTO test01 VALUES (:uc1, :uc1);

.LAYOUT layoutname;
.FIELD uc1 INTEGER;
.FIELD uc2 INTEGER;
.DML LABEL labelname;

INSERT INTO test01 VALUES (:uc2, (:uc2 + :uc2));
```

The following simple INSERT statements succeed because they do not reuse Teradata Parallel Data Pump field variables.

```
.LAYOUT layoutname;
.FIELD uc1 INTEGER;
.FIELD uc2 INTEGER;
.DML LABEL labelname;

INSERT INTO test01 VALUES (:uc1, :uc2);

.LAYOUT layoutname;
.FIELD uc1 INTEGER;
.FIELD uc2 INTEGER;
.DML LABEL labelname;

INSERT INTO test01 VALUES (:uc2, (:uc1 + :uc1));
```

- INSERT operations into GENERATED BY DEFAULT identity columns using Teradata Parallel Data Pump field variables cannot specify field variables if the value inserted into the identity column is derived from an expression that includes a Teradata Parallel Data Pump field variable.
- INSERT operations into GENERATED BY DEFAULT and GENERATED ALWAYS identity columns using Teradata Parallel Data Pump field variables cannot insert into the identity column of more than 1 identity column table.
- You cannot specify an identity column for a nonpartitioned NoPI table, but you can specify an identity column for a column-partitioned table.

The GENERATED ... AS IDENTITY keywords introduce a clause that specifies the following.

- The column is an identity column.

- Vantage generates values for the column when a new row is inserted into the table except in certain cases described later.

| GENERATED | Description |
|------------|--|
| ALWAYS | Always generates a unique value for the column when Vantage inserts a new row into the table and NO CYCLE is specified. If you load the same row twice into an identity column table, it is not rejected as a duplicate because it is made unique as soon as an identity column value is generated for it. Some preprocessing must still be performed on rows to be loaded into identity column tables if real world uniqueness is a concern. |
| BY DEFAULT | Generates a unique value for the column when Vantage inserts a new row into the table only if the INSERT statement does not specify a value for the column. The generated value is guaranteed to be unique within the set of generated values only if you specify the NO CYCLE option. |

| Identity Column Purpose | Option |
|---|---------------------|
| Ensure a UPI, USI, PK, or some other row uniqueness property. | ALWAYS ... NO CYCLE |
| <ul style="list-style-type: none"> Load data into or unload data from a table. Copy rows from one table to another table. Fill in gaps in the sequence. Reuse numbers that once belonged to now-deleted rows. | BY DEFAULT |

- Teradata Unity does not support identity columns for bulk data loads because Vantage processes rows in different orders on different systems. It is not possible to guarantee that the same data row is assigned the same identity value on all systems. Also, it is not possible to assign Teradata Unity-determined values to specific rows across systems, especially when the identity column is the primary index for the table.

If your system has existing tables with identity columns that must run under Teradata Unity, you can use an ALTER TABLE statement to drop the identity attribute from an identity column without dropping the column itself or its data. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Process for Generating Identity Column Numbers

Vantage allocates identity column numbers differently depending on whether an operation is a single row or USING request modifier-based insert or an INSERT ... SELECT operation.

| Type of Insert Operation | Location of Identity Column Numbers Cache |
|--|---|
| <ul style="list-style-type: none"> Single row USING clause | PE. |

| Type of Insert Operation | Location of Identity Column Numbers Cache |
|--|---|
| Examples are BTEQ imports and Teradata Parallel Data Pump INSERT operations. | |
| INSERT ... SELECT | AMP. |

Vantage uses a batch numbering scheme to generate identity column numbers. When the initial batch of rows for a bulk insert arrives at a PE or AMP, the system reserves a range of numbers before it begins to process the rows. Each PE or AMP retrieves the next available value for the identity column from dictionary table *DBC.IdCol* and immediately increments it by the value of the setting for the *IdCol Batch Size* parameter in the DBS Control record.

After a range of numbers is reserved, the system stores the first number in the range in a vproc-local identity column cache. Different tasks doing concurrent inserts on the same identity column table allot a number for each row being inserted and increment it in the cache. When the last reserved number has been issued, the PE or AMP reserves another range of numbers and updates the entry for this identity column in *DBC.IdCol*.

This process explains the following apparent numbering anomalies.

- Because the Teradata architecture is highly parallel, generated identity column numbers do not necessarily reflect the chronological order in which rows are inserted.
- Sequential numbering gaps can occur. Because the cached range of reserved numbers is not preserved across system restarts, exact increments cannot be guaranteed. For example, the identity column values for 1,000 rows inserted into a table with an INCREMENT BY value of 1 might not be numbered from 1 to 1,000 if a system restart occurs before the identity column number pool is exhausted.

The following properties of an identity column INSERT operation determine whether the identity number for a row is generated on the PE before being distributed to its destination AMP or if the identity number for a row is generated on the destination AMP after having been distributed.

- If you insert a value into an identity column that is also the primary index for the table and defined as GENERATED ALWAYS, Vantage generates the identity column value on the PE before it hashes the row to its AMP.
- If you insert a value into an identity column that is not the primary index for a table, Vantage generates the identity column value on the destination AMP for the row.

Rules for Specifying Identity Columns

The following rules and restrictions apply to specifying identity columns as part of a table definition.

- Only one identity column can be specified per table.
- An identity column need not be the first column defined for a table.
- An identity column need not be a primary index column.

For example, a NUPI table can also have an identity column.

- You cannot specify an identity column for a nonpartitioned NoPI table.
- You can specify an identity column for a column-partitioned table.
- Do not specify an identity column for a table that will be under the control of Teradata Unity and that will be loaded using bulk data load utilities because Vantage processes rows in different orders on different systems. Because of this, it is not possible to guarantee that the same data row is assigned the same identity value on all systems.
- An identity column cannot be part of any of the following types of index:
 - Composite primary index
 - Composite secondary index
 - Join index
 - Hash index
 - Value-ordered index
- You cannot perform atomic upsert operations on tables with an identity column as their primary index.
- An identity column must be defined with an exact numeric data type. For the ANSI SQL:2011 definition of an exact numeric data type, see *Teradata Vantage™ - Database Design*, B035-1094 or *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Note:

Even when the data type for an identity column is specified as BIGINT, the largest identity column value the system generates is restricted to a size that can be accommodated by a DECIMAL(18,0), NUMBER(18,0), or NUMERIC(18,0) data type.

The following are examples of valid numeric data types for identity columns.

- BYTEINT
- DECIMAL(*n*,0)
- INTEGER
- NUMERIC(*n*,0)
- NUMBER(*n*,0)
- SMALLINT
- BIGINT

The following table lists the MAXVALUE and MINVALUE defaults for various identity column data types.

| Data Type | Default MAXVALUE | Default MINVALUE |
|------------------------------|------------------|------------------|
| DECIMAL(1,0) NUMERIC(1,0) | 9 | -9 |
| DECIMAL(2,0) NUMERIC(2,0) | 99 | -99 |

| Data Type | Default MAXVALUE | Default MINVALUE |
|---|---------------------------|----------------------------|
| ... | ... | ... |
| DECIMAL(18,0) NUMERIC(18,0) NUMBER(18,0) Even when the value for the DBS Control parameter MaxDecimal is set to 38, the largest identity column value the system generates is restricted to a size that can be accommodated by any of the following data types. <ul style="list-style-type: none"> ◦ DECIMAL(18,0) ◦ NUMBER(18,0) ◦ NUMERIC(18,0) | 999,999,999,999,999,999 | -999,999,999,999,999,999 |
| BYTEINT | 127 | -127 |
| SMALLINT | 32,767 | -32,767 |
| INTEGER | 2,147,483,647 | -2,147,483,647 |
| BIGINT Even when the data type for an identity column is specified as BIGINT, the largest identity column value the system generates is restricted to a size that can be accommodated by a DECIMAL(18,0), NUMBER(18,0), or NUMERIC(18,0) data type. | 9,223,372,036,854,775,807 | -9,223,372,036,854,775,807 |

The following are examples of non-valid data types for identity columns.

- All non-numeric types, including DateTime, INTERVAL, Period, Geospatial, UDT, ARRAY, VARRAY, BLOB, and CLOB types. This is true even if the UDT is based on 1 (distinct UDT) or several (structured UDT) of the valid exact numeric types.
- DECIMAL(n,m) where m is not 0
- DOUBLE PRECISION
- FLOAT
- NUMERIC(n,m) where m is not 0
- NUMBER (approximate type)
- REAL

| Table Type | Identity Column | Description |
|------------|--|-----------------------------|
| MULTISET | GENERATED ALWAYS | Cannot have duplicate rows. |
| SET | <ul style="list-style-type: none"> ◦ GENERATED ALWAYS ◦ GENERATED BY DEFAULT | |
| MULTISET | GENERATED BY DEFAULT | Can have duplicate rows. |

- If you insert values into a GENERATED BY DEFAULT IDENTITY column SET table, the inserted row (but not the inserted value for the identity column) must be unique or the system does not accept it.
- If you insert values into a GENERATED BY DEFAULT IDENTITY column in a MULTiset table, and those values must always be unique, then you must explicitly constrain the column to be unique by specifying it to be a UPI, a USI, a PRIMARY KEY, or UNIQUE. Because Vantage treats nulls in this case as if they were all equal to one another, you can only insert 1 null into a column constrained to be UNIQUE. Because of this, it is usually considered to be preferable to define any such column to be both UNIQUE and NOT NULL.

| Identity Column | Action for User-Defined Insert Values |
|----------------------|--|
| GENERATED ALWAYS | Replaces them with system-generated identity values. |
| GENERATED BY DEFAULT | Accepts them unless otherwise constrained. |

A column specified to be GENERATED BY DEFAULT AS IDENTITY with NO CYCLE only generates a unique column value for inserted rows that contain a null identity column.

To ensure the uniqueness of GENERATED BY DEFAULT column values:

- You must specify NO CYCLE.
- Any user-specified values you specify must be outside the range of any system-generated values.
- You must enforce the uniqueness of the user-specified values yourself.
- You cannot specify any of the following column attributes for an identity column.
 - DEFAULT
 - BETWEEN
 - COMPRESS
 - CHECK constraints
 - FOREIGN KEY ... REFERENCES constraints

Identity Columns, Duplicate Column Values, and Duplicate Rows

The use of identity columns presents 2 different duplication issues.

- Duplicate identity column values
- Unintentionally duplicated rows

Values for identity columns are guaranteed to be unique only when the column is specified using GENERATED ALWAYS ... NO CYCLE unless otherwise constrained.

Duplicate identity column values can occur in either of the following situations.

- The column is specified to be GENERATED ALWAYS, but the CYCLE option is specified.

In this case, the column can reach its maximum or minimum value and then begin recycling through previously generated values.

- The column is specified to be GENERATED BY DEFAULT and an application specifies a duplicate value for it.

Duplicate rows can occur in any of the following situations.

- A previously completed load task is resubmitted erroneously.
Tables that do not have identity columns, but that are either specified to be SET tables or have at least 1 unique index do not permit the insertion of duplicate rows.
- Teradata Parallel Data Pump runs without the ROBUST option enabled and a restart occurs.
- A session aborts, but rows inserted before the abort occurred are not deleted before the session is manually restarted.

NOTICE

In many cases, such rows are not duplicates in the sense defined by the relational model. For example, in the case of a load task mistakenly being run multiple times, the new rows are not considered to be duplicates in the strict relational sense because even though they are the same client row (where they do not have the uniqueness-enforcing identity column value that is defined for them on the server), they have *different* identity column values on the server and, therefore, are not duplicates of one another.

Suppose, for example, you accidentally load employee Reiko Kawabata into an *employee* table twice, where the *employee_number* column is an identity column. After doing this, you have 2 employee rows that are identical except for their different *employee_number* values. While this is an error from the perspective of the enterprise, the 2 rows are *not* duplicates of one another because they have different *employee_number* values. The problem is not with the feature, which works exactly as it is designed to work.

This means that it is imperative for you to enforce rigorous guidelines for dealing with identity column tables at your installation to ensure that these kinds of nebulous duplications do not corrupt your databases.

Identity Columns and Bulk Inserts

Identity column values for rows bulk-inserted into a table defined with a GENERATED ALWAYS identity column are replaced automatically with system-generated numbers. A warning of this replacement is issued only once for all rows inserted using an INSERT ... SELECT statement. Other bulk-inserting utilities such as Teradata Parallel Data Pump and BTEQ imports receive the same warning for each row inserted.

Rows bulk-inserted into tables with either of the following identity column definitions are not guaranteed to be unique unless the appropriate row columns are constrained by a UPI, USI, PRIMARY KEY, or UNIQUE constraint.

- GENERATED BY DEFAULT
- GENERATED ALWAYS with CYCLE

Optimizing Bulk Inserts Into an Identity Column

The performance of bulk INSERT operations into a table with an identity column depends on the DBS Control setting for the *IdCol* batch size parameter (see *Teradata Vantage™ - Database Utilities*, B035-1102 for information about DBS Control). You should use this variable to optimize the trade off between bulk insert performance and your tolerance for the identity column numbering gaps that can occur with a system restart. The performance enhancements realized by larger batch size are due to the decrease or elimination of updates to *DBC.IdCol* that must be performed to reserve identity column numbers for a load.

While a larger *IdCol* batch size setting enhances bulk insert performance, it also means that more identity column numbers are lost if a system restart occurs. Reserved identity column numbers are memory-resident. If the database must be restarted, the batch insert transaction is rolled back, and those reserved identity column numbers are permanently lost.

The optimal batch size for your system is a function of the number of AMPs. The *IdCol* batch size default value of 100,000 rows should be sufficient for most workloads, but you should perform your own tests to determine what works best for your applications and system configuration.

When you are bulk inserting rows into an identity column table during a Teradata mode session, be careful not to upgrade the default lock severity of a SELECT operation on *DBC.IdCol* from ACCESS to READ or higher severity if the SELECT is performed within the boundaries of an explicit transaction because you run the risk of creating a deadlock on the *IdCol* table.

Identity Column Inserts and Transaction-Bound Table Locks on *DBC.IdCol*

An identity column cache is populated by a range of column numbers whenever an INSERT operation into an identity column table is processed. The values for the range are determined by the following factors.

- The next available value stored in *DBC.IdCol.AvailValue*
- The defined incremental identity column batch size for the system

This quantity is specified by the DBS Control parameter *IdCol* batch size. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102 and [Optimizing Bulk Inserts Into an Identity Column](#).

When the pool of available identity column numbers in the cache is exhausted, Vantage must update *DBC.IdCol.AvailValue* and obtain a new range of numbers.

If an ongoing transaction has *DBC.IdCol* locked, as it would during a SELECT statement against the table, then system access to the column number pool is temporarily blocked and all inserts into the relevant identity column table are suspended until the transaction either commits or aborts. A SELECT statement against *DBC.IdCol* places an ACCESS lock on the table.

The SELECT privilege on *DBC.IdCol* is restricted to user *DBC* and any user explicitly granted the privilege by *DBC*.

When performing a SELECT statement against *DBC.IdCol* in Teradata session mode, you run the risk of creating a deadlock if you upgrade the default lock from ACCESS to READ or higher severity when both of the following conditions are true.

- The SELECT statement is run within the boundaries of an explicit transaction.
- Concurrent INSERT operations are being performed on an identity column table.

Comparing the Characteristics of the Various Compression Methods Available to Vantage

For a comparison of the compression methods that are available, see *Teradata Vantage™ - Database Administration*, B035-1093.

Compressing Column Values Using Only Multivalue Compression

Teradata systems use a variety of compression methods. Multivalue and row compression are lossless. The original data can be reconstructed exactly from their compressed forms. Algorithmic compression can be a non-loss or lossy process.

Multivalue compression refers to the storage of column values values one time only in the table header, not in the row itself, and pointing to them using an array of presence bits in the row header. This method of *value* compression is called Dictionary Indexing. See *Teradata Vantage™ - Database Design*, B035-1094.

You can specify multivalue compression only for columns having one of the following data types.

- All numeric types
- DATE
- CHARACTER and CHARACTER SET GRAPHIC
- VARCHAR and CHARACTER SET VARGRAPHIC
- BYTE
- VARBYTE
- A distinct UDT type based on any of the predefined data types in this list.

Because the list of compressed data values is memory-resident any time the table is being accessed, the system can access compressed values in 2 ways, depending on the presence bits for the column.

- Using a pointer reference to the value in the current row
- Using a pointer reference to the value in the multivalue

There is a small cost for compressing a value when a row is stored for the first time, and a one-time cost to convert an existing noncompressed column to a compressed column. But for queries, even those made against small tables, multivalue compression is a net win as long as the chosen compression reduces the size of the table. See *Teradata Vantage™ - Database Design*, B035-1094.

For multivalue-compressed spools, if a column is copied to spool with no expressions applied against it, Vantage copies just the compressed data into the spool, saving CPU and I/O costs. Once in spool,

compression works exactly as it does in a base table. There is a compressed multivalue list in the table header of the spool that is memory-resident while the system is operating on the spool.

You can compress nulls and as many as 255 distinct values per column using standard default multivalue compression. You can compress the values for an unlimited number of columns per table. Although there is no defined limit on the number of columns that can be compressed per table, the actual number of columns with compression that can be achieved is a function of the amount of space available in the table header that is not consumed by other optional table characteristics. See *Teradata Vantage™ - Database Design*, B035-1094.

| Type of Multivalue Compression | Description |
|--------------------------------|--|
| Single-valued | The default is null if no value is specified explicitly. |
| Multivalued | <p>You must specify all values explicitly. There is no default.</p> <p>There is no essential ordering for specifying values to be compressed for a column.</p> <p>You cannot specify the same value or NULL more than once in the compression multivalue for a column.</p> |

Note that multivalue compression, once defined for table columns, is retained as a property of spools containing data from the table as well, so compression is not just a property of data stored on disk.

Multivalue Compression Capacity

Using multivalue compression to compress a substantial number of values in a table can cause the following capacity issues.

Table Header Overflow

Compressed multivalue information is stored in the table header, and like other database rows, table header rows have a maximum length of 64 KB. Null storage is indicated by presence bits in the row header and has no impact on the size of the table header. The maximum size of a table header is 1 MB. If table header overflow occurs, you must resubmit the CREATE TABLE statement fewer compressed column values.

For additional information about modifying multivalue compressed columns, see [Rules for Adding or Modifying Multivalue Compression for a Column](#).

Several factors contribute to the length of table header rows, including partitioning and UDT columns, so it is not possible to provide rigid guidelines to avoid overflow. For more information about table headers, see *Teradata Vantage™ - Database Design*, B035-1094.

Dictionary Cache Overflow

When this happens, the system reports a 3930 error. Increase the size of your dictionary cache from its current value to the maximum size of 1 MB. The default value for the size of the dictionary cache is also 1 MB, so you would only need to increase its size if it had reduced for some reason.

Compressing Column Values Using Only Algorithmic Compression

The term *compression* is used to mean different things for Teradata systems. Algorithmic compression can be done using either a non-loss method or a lossy method.

- When describing algorithmic compression of column values, compression refers to whatever methods you choose to implement to compress byte, character, and graphic data. Algorithmic compression is implemented by assigning a compression and a decompression UDF to a table column. Although you can only specify one compression UDF and one decompression UDF per column, those functions can compress and decompress multiple column values. For the syntax used to assign algorithmic compression UDFs to columns, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. For details about coding UDFs to implement algorithmic compression, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- When you specify algorithmic compression for a column, you might be using a form of lossless compression, or you might be using lossy compression for sound, video, graphic, or picture files.
- When you specify a UDF name for algorithmic compression, you must also specify its containing database or user.
- You can specify algorithmic compression only for columns having one of the following data types.
 - BYTE
 - CHARACTER
 - VARBYTE
 - VARCHAR
 - All UDT types that are not based on BLOBs or CLOBs.

You can specify algorithmic compression for Period types.

- Algorithmic compression is not supported for the columns of join indexes
- Algorithmic compression automatically compresses nulls.
- Algorithmic compression is carried to spools whenever possible.
- You can specify algorithmic compression for an unlimited number of columns in a table.
- The column *DBC.TVFields.CompressValueList* stores the names of the compression and decompression algorithms specified for a column as well as the compressed multivalue for any multivalue compression specified for a column.

The maximum size of *DBC.TVFields.CompressValueList* is 8,192 characters. If the composite size of the value and algorithmic compression information exceeds this limit, the system reports an error.

The following rules apply to the compression and decompression algorithms you create as UDFs.

- The UDFs must be created in either the SYSUDTLIB or the TD_SYSFNLIB database.
- The UDFs must be scalar UDFs.

Neither aggregate nor table UDFs are valid for algorithmic compression.

- The UDFs must be deterministic.

- The UDFs can only have 1 input parameter, and the data type of the value passed must be a type that is valid for algorithmic compression.
- The input to an DECOMPRESS USING UDF must have a variable data type.

The output from an DECOMPRESS USING UDF must be compatible with the data type of the column being decompressed.

- Algorithmic compression cannot truncate data values.

The length of the column must be less than or equal to the length of the input parameter passed to the COMPRESS USING UDF.

Similarly, the length of the column must be less than or equal to the length of the output parameter returned by the DECOMPRESS USING UDF.

- A user who references a column defined with algorithmic compression must have the privilege on the COMPRESS USING and DECOMPRESS USING UDFs specified for that column.
- Vantage supports function overloading for the COMPRESS USING and DECOMPRESS USING UDFs you write.

For example, you can overload an algorithm that handles different data types, and the system executes the appropriate function depending on the data type of the input parameter value passed to it.

- Vantage supports algorithmic compression of Period columns.

The costs of compressing and decompressing column data algorithmically depends on the algorithms used, and you must evaluate those costs yourself.

Teradata provides the following system-defined external UDFs that you can use for algorithmic compression and decompression. See *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210. These functions are stored in the TD_SYSFNLIB database.

Common Rules for Multivalue and Algorithmic Compression

With standard multivalue compression, compressed data values can be accessed without decompression processing overhead. Vantage does not need to decompress blocks or other large chunks of data to be able to access a single row or value. This removes the significant CPU cost tradeoff common to most compression implementations.

The following rules apply to multivalue and algorithmic compression.

- For table columns, you can specify multivalue compression, algorithmic compression, or both.
- If you specify multivalue and algorithmic compression for a column, you can specify the options in any order: multivalue compression followed by algorithmic compression or algorithmic compression followed by multivalue compression.
- When you specify multivalue compression and algorithmic compression on the same column, Vantage applies algorithmic compression only to values that are not also specified for multivalue compression.
- Specifying algorithmic compression alone or in combination with multivalue compression for a column has the following data type size limitations:

| Type of Column Compression | Data Type Size |
|-----------------------------|--|
| Algorithmic only | Has no limit. |
| Algorithmic and multivalued | Limited by the maximum valid size for multivalued compression. |

- You cannot specify multivalued compression or algorithmic compression on a column that is a component of the primary index or primary AMP index for a table.
- You can specify multivalued compression and algorithmic compression on a column that is a component of a secondary index for a table.
- You can specify multivalued compression, algorithmic compression, or both on a column that is a component of a referential integrity constraint.
- You can specify multivalued compression, algorithmic compression, or both on the columns of nonpartitioned data tables and global temporary tables.
- Algorithmic compression and multivalued compression are supported by default for the columns of spools. If a column is defined with multivalued compression, algorithmic compression, or both, the compression defined for the column is inherited for the pool column.
- You can copy the definition of a table to a different name using the COPY TABLE ... AS option, and the target table retains the algorithmic or multivalued compression defined for the source table.

Other Forms of Compression Used by Vantage

Hash and Join Index Row Compression

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

Block-Level Compression

Block-level compression, or BLC, is a lossless physical compression method that can be specified in a CREATE TABLE statement. Block-level compression can be applied to row-level security constraint columns, which are not eligible for multivalued compression and algorithmic compression.

For information about BLC and how the system applies compression to various types of tables, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Autocompression for Column-Partitioned Tables

When describing system-selected autocompression of containers in column-partitioned tables, compression refers to various compression methods that Vantage can use to compress the column values of a container.

Combining Multivalue, Algorithmic, and Block-Level Compression

You can combine multivalue compression, algorithmic compression, and block-level compression for the same table to achieve better compression. However, 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.

Integrity Constraint Levels

You can specify integrity constraints as either column attributes or as table attributes.

A constraint that applies only to an element of a single column in a base table is referred to as a *column attribute constraint*.

Consider the following table definition.

```
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));
```

Table *good_1* defines the following three column attribute constraints.

- A simple PRIMARY KEY constraint named *primary_1* on *good_1.column_1*.
For this constraint, the values inserted into *good_1.column_1* must uniquely identify the row. If the constraint is violated, the database rejects the insertion of the row.
Because PRIMARY KEY values should not be updated, this constraint is generally not violated for updates, but if is, the database rejects the update.
- A simple UNIQUE constraint named *unique_1* on *good_1.column_2*.
For this constraint, the values inserted into *good_1.column_2* must always be unique. If the constraint is violated, the database rejects the insertion of the row or the update of *good_1.column_2* in an existing row.
- A simple CHECK constraint named *check_1* on *good_1.column_3*.
For this constraint, the values inserted into *good_1.column_3* must either always be greater than 0 or null. If the constraint is violated, the database rejects the insertion of the row or the update of *good_1.column_3* in an existing row.

A constraint that applies either to multiple columns within a base table or to 1 or more columns in another base table is referred to as a *table attribute constraint*.

Now consider the following table definition.

```
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 ref_1 FOREIGN KEY (column_5, column_6)
    REFERENCES parent_1 (column_2, column_3));
```

Table *good_2* defines the following 4 table-level constraints.

- A composite PRIMARY KEY constraint named *primary_1* on *column_1* and *column_2*.
For this constraint, the values inserted into *good_2.column_1* and *good_2.column_2* must uniquely identify the row. If the constraint is violated, the database rejects the insertion of the row.
Because PRIMARY KEY values should not be updated, this constraint is generally not violated, but if it is, the database rejects the update.
- A composite UNIQUE constraint named *unique_1* on *column_3* and *column_4*.
For this constraint, the values inserted into *good_2.column_3* and *good_2.column_4* must always be unique within both of those columns. If the constraint is violated, the database rejects the insertion of the row or the update of columns *good_2.column_3* or *good_2.column_4* in an existing row.
- A multicolumn CHECK constraint on *column_3* and *column_4*.
For this constraint, either the value inserted into *good_2.column_3* must be greater than zero, or the value inserted into *good_2.column_4* must not be null. If both of these constraints is violated, the database rejects the insertion of the row or the update of those columns in an existing row in *good_2*.
Because this constraint is defined using an OR condition, the constraint is satisfied if either *good_2.column_3* is greater than zero or *good_2.column_4* is not null.
- A composite FOREIGN KEY constraint on *good_2.column_5* and *good_2.column_6* that references *parent_1.column_2* and *parent_1.column_3*, with *good_2.column_5* mapping to *parent_1.column_2* and *good_2.column_6* mapping to *parent_1.column_3*.
For this constraint, if the respective values of *good_2.column_5* and *good_2.column_6* do not match any pairs of values equal to *parent_1.column_2* and *parent_1.column_3*, the database rejects the insertion of the row into *good_2* or the update of those columns in an existing row in *good_2*.
This constraint is also valid when all of columns *good_2.column_5*, *good_2.column_6*, *parent_1.column_2* and *parent_1.column_3* are null, though it is never good design practice to permit any column of a referential integrity constraint to be null.

PRIMARY KEY Constraints

The primary key of a table is a column set that uniquely identifies each row of that table. See *Teradata Vantage™ - Database Design*, B035-1094. You cannot define more than 1 primary key for a table. Primary keys, which are a logical construct in the relational model, are usually implemented physically as the unique primary index.

To create a composite, or multicolumn, primary key for a table, you must specify the PRIMARY KEY constraint at the table level, not the column level.

Columns defined with a data type from the following list cannot be a component of a PRIMARY KEY constraint.

- BLOB
- CLOB
- BLOB-based UDT
- CLOB-based UDT
- ARRAY
- VARRAY
- Period
- Geospatial

When a table has a nonunique primary index, you should consider defining its primary key explicitly using the PRIMARY KEY clause. Primary and other alternate keys are also used with foreign keys to enforce referential integrity relationships between tables (see [Standard Referential Integrity Constraints](#), [Batch Referential Integrity Constraints](#), and [Referential Constraints](#)).

Surrogate Keys

Situations sometimes occur where the identification and choice of a simple primary key is difficult, if not impossible. There might be no single column that uniquely identifies the rows of a table or there might be performance considerations that argue against using a composite key. In these situations, surrogate keys are an ideal solution.

A surrogate key is an artificial simple key used to identify individual rows uniquely when there is no natural key or when the situation demands a simple key, but no natural non-composite key exists. Surrogate keys do not identify individual rows in a meaningful way: they are simply an arbitrary method to distinguish among them.

Surrogate keys are typically arbitrary system-generated sequential integers. You can generate surrogate keys in the database using the identity column feature (see [Identity Columns](#)). To use an identity column to generate surrogate key values, specify the GENERATED ALWAYS and NO CYCLE options and ensure that the data type for the column is either NUMERIC(18,0) or DECIMAL(18,0).

UNIQUE Constraints

UNIQUE constraints specify that the column set they modify must contain unique values. Vantage implements UNIQUE constraints as either a unique secondary indexes or as a single-table join index.

The following rules apply to UNIQUE constraints.

- UNIQUE constraints should always be paired with a NOT NULL attribute specification.
See [Nullable Columns Are Valid for Unique Indexes](#).
- UNIQUE constraints can be defined at column-level (simple) or at table-level (composite).
- Column-level UNIQUE constraints refer only to the column on which they are specified.
- Table-level UNIQUE constraints can be defined on multiple columns using a column name list.
- A table-level UNIQUE constraint can be defined on a maximum of 64 columns.
- A maximum of 100 table-level constraints can be defined for any table.
- UNIQUE constraints cannot be defined on a global temporary trace table.
- You can define a UNIQUE constraint for a column-partitioned table.

You cannot define a UNIQUE constraint for a nonpartitioned NoPI table.

Otherwise, the system does not return an error, but instead converts the UNIQUE constraint specification to a UNIQUE NOT NULL secondary index specification. As a result of this conversion, if you submit a SHOW TABLE statement on such a table, the create text that the system returns does not show a UNIQUE constraint specification, but instead returns a UNIQUE NOT NULL secondary index or single-table join index specification on the column set that had been specified for the UNIQUE constraint.

Note that this is different from the ordinary *implementation* of a UNIQUE constraint column set as a UNIQUE NOT NULL secondary index because Vantage actually changes the stored create text for a NoPI table defined with a UNIQUE constraint, it does not simply implement the constraint as a USI.

- UNIQUE constraints cannot be defined on columns defined with any of the following data types.
 - BLOB
 - CLOB
 - BLOB-based UDT
 - CLOB-based UDT
 - ARRAY
 - VARRAY
 - Period
 - Geospatial

PRIMARY KEY Constraints Versus UNIQUE Constraints

UNIQUE and PRIMARY KEY constraints can *only* be defined on a column set that is also constrained to be NOT NULL.

To create a composite, or multicolumn, primary key for a table, you must specify the PRIMARY KEY constraint at the table level, not the column level.

Both UNIQUE and PRIMARY KEY constraints can be defined on a UDT column.

Vantage also supports the related constraints UNIQUE INDEX and UNIQUE PRIMARY INDEX.

CHECK Constraints

CHECK constraints are the most general type of SQL constraint specification. Depending on its position in the CREATE TABLE or ALTER TABLE SQL text, a CHECK constraint can apply either to an individual column or to an entire table.

Vantage derives a table-level partitioning CHECK constraint from the partitioning expression for a partitioned table. The text for this derived constraint cannot exceed 16,000 characters. Otherwise, the system returns an error to the requestor. For more information, see [Rules and Usage Notes for Partitioned Tables](#).

The following rules apply to all CHECK constraints.

- You can define CHECK constraints at either the column level or the table level.
- You can define multiple CHECK constraints for a table.
- You can define CHECK constraints at column-level or at table-level.
- The specified predicate for a CHECK constraint can be any *simple* boolean search condition.
Subqueries, aggregate expressions, and CASE expressions are not valid search conditions for CHECK constraint definitions.
- You cannot invoke an SQL UDF from a CHECK constraint expression.
- You cannot specify CHECK constraints at any level for volatile tables or global temporary trace tables.
- Note that a combination of table-level, column-level, and WITH CHECK OPTION on view constraints can create a constraint expression that is too large to be parsed for INSERT and UPDATE statements.
- 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 violated for another, even though the identical data is inserted or updated.

The following is an example of the potential importance of this. A CHECK constraint is checked on insert and update of a base table character column, and might affect whether a sparse join index defined with that character column is updated or not for different session character collations, in which case different request results might occur if the index is used in a query plan compared to the case where there is no sparse join index to use.

- Vantage considers unnamed CHECK constraints specified with identical text and case to be duplicates, and returns an error when you submit them as part of a CREATE TABLE or ALTER TABLE statement.

For example, the following CREATE TABLE statement is valid because the case of *f1* and the case of *F1* are different.


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

The following CREATE TABLE statement, however, is *not* valid because the case of the 2 unnamed *f1* constraints is identical. This request returns an error to the requestor.

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

- The principal difference between defining a CHECK constraint at column-level or at table-level is that column-level constraints cannot reference other columns in their table, while table-level constraints, by definition, *must* reference other columns in their table.
- Columns defined with a data type from the following list cannot be a component of a CHECK constraint.
 - BLOB
 - CLOB
 - UDT
 - ARRAY
 - VARRAY
 - Period
 - Geospatial
- You cannot define a CHECK constraint on a row-level security constraint column of a row-level security-protected table.
- If a row-level security-protected 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.

The following rules apply only to column-level CHECK constraints.

- You can specify multiple column-level CHECK constraints on a single column.
If you define more than 1 *unnamed* distinct CHECK constraint for a column, Vantage combines them into a single column-level constraint.
However, Vantage handles each *named* column-level CHECK constraint separately, as if it were a table-level named CHECK constraint.
- A column-level CHECK constraint cannot reference any other columns in its table.

The following rules apply only to table-level CHECK constraints.

- A table-level constraint usually references at least 2 columns from its table.
- Table-level CHECK constraint predicates cannot reference columns from other tables.
- You can define a maximum of 100 table-level constraints for a table at one time.

FOREIGN KEY Constraints

If you specify a foreign key as a table attribute, the FOREIGN KEY *column_name* list must specify columns defined in the table definition for the target table, and the same column name cannot be specified in the foreign key definition more than once.

You can define a simple foreign key at column level, but you must define composite, or multicolumn, foreign keys at table level. Note that the syntax for simple and composite foreign keys is different.

You can define multiple foreign key constraints for a table.

Columns defined with a data type from the following list cannot be a component of a FOREIGN KEY constraint.

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

The specified *column_name* list must be identical to an alternate key in the referenced table that is defined as the PRIMARY KEY column set for that table, as a set of columns defined with the UNIQUE attribute, or as a USI. This is not mandatory for Referential Constraints. See [Rules for Using Referential Constraints](#) for details. The *table-name* variable refers to the referenced table, which must be a user base data table, not a view.

Referential Integrity Constraints

Vantage supports referential integrity constraints between candidate key column sets of a parent table and foreign key column sets of a child table. Both explicitly declared PRIMARY KEY constraints and alternate key constraints are supported.

You can define multiple referential integrity constraints for a table.

Vantage supports the following types of referential integrity constraints.

- Standard (see [Standard Referential Integrity Constraints](#)).
- Batch (see [Batch Referential Integrity Constraints](#)).
- Referential (see [Referential Constraints](#)).

If a referential integrity constraint is defined where either or both of the tables have row-level security constraints, execution of the referential integrity constraint does not execute any security policy UDFs defined for the table. Execution continues as if the tables were not row-level security-protected.

Standard Referential Integrity Constraints

Standard referential integrity is a form of referential integrity that incurs modest performance overhead because it tests, on a row-by-row basis, the referential integrity of each insert, delete, and update operation on a column set with a defined referential integrity constraint. Each such operation is checked individually by AMP software, and if a violation of the specified referential integrity relationship occurs, the operation is rejected and an error message is returned to the requestor.

You cannot use bulk data loading utilities like FastLoad, MultiLoad, and Teradata Parallel Transporter to load rows into tables defined with standard referential integrity.

Because referential integrity is checked at the granularity of individual rows, there is no opportunity for so-called dirty reads to occur during an insert, delete, or update operation on a table.

Batch Referential Integrity Constraints

Batch referential integrity is a form of referential integrity checking that is less expensive to enforce in terms of system resources than standard referential integrity because it is enforced as an all-or-nothing operation rather than on a row-by-row basis, as standard referential integrity is checked.

Batch RI also conserves system resources by not using reference indexes. Eliminating the use of reference indexes has the following effects.

- System performance is enhanced because the system does not need to build and maintain them.
- Disk storage is saved because the space that would otherwise be used to store them is instead available for other storage needs.

See *Teradata Vantage™ - Database Design*, B035-1094 for further information about reference indexes.

Batch referential integrity relationships are defined by specifying the WITH CHECK OPTION phrase for a REFERENCES constraint. When you specify this phrase, the database enforces the defined RI constraint at the granularity of a single transaction or request.

This form of RI is tested by joining the relevant parent and child table rows. If there is an inconsistency in the result, then the system rolls back the entire transaction in Teradata session mode or the problem request only in ANSI session mode.

In some situations, there can be a tradeoff for this enhanced performance because when an RI violation is detected, the entire transaction or request rolls back instead of 1 integrity-breaching row.

In many instances, this is no different than the case for standard RI because for most transactions, only 1 row is involved.

NOTICE

There is a significant potential for performance problems when updating tables with batch RI constraints in situations where multiple update operations are involved: for example, an INSERT ... SELECT or MERGE operation involving thousands or even millions of rows. This would be an expensive operation to have to roll back, cleanse, and then rerun.

Similar very large batch RI rollbacks can occur with ALTER TABLE and CREATE TABLE statements whose referential integrity relationships do not verify.

Because of its all-or-none nature, batch RI is best used only for tables whose normal workloads you can be very confident are not going to cause RI violations.

You cannot use bulk data loading utilities like FastLoad, MultiLoad, or Teradata Parallel Transporter to load rows into tables defined with batch referential integrity.

Batch referential integrity is a Teradata extension to the ANSI SQL:2011 standard.

Referential Constraints

In some circumstances, the Optimizer is able to create significantly better query plans if certain referential relationships have been defined between tables specified in the request. The Referential Constraint feature, also referred to as *soft referential integrity*, permits you to take advantage of these optimizations without incurring the overhead of enforcing the suggested referential constraints.

NOTICE

The Referential Constraint feature presumes a high level of trust between Vantage and its users, assuming that users do not violate any of the defined Referential Constraints even though the system does not enforce those constraints. Otherwise, you can produce incorrect results and corrupt your databases in some situations when Referential Constraints are defined on application-critical columns and appropriate care is not taken to ensure that data integrity is maintained. See [Validating the Integrity of Base Tables In a Referential Constraint Relationship](#). You should specify Referential Constraints *only* when the potential for data corruption and the possibility of query errors can be managed in such a way that it is not critical to your application. For an example, see [Scenario for Data Corruption With Referential Constraints](#).

Referential Constraint relationships are defined by specifying the WITH NO CHECK OPTION phrase for a FOREIGN KEY ... REFERENCES constraint. When you specify this phrase, Vantage does not enforce the defined RI constraint. This implies the following things about child table rows.

- A row having a value for a referencing column can exist in a table even when no equivalent parent table value exists.
- A row can, in some circumstances, match *multiple* rows in its parent table when the referenced and referencing column values are compared. This can happen because the candidate key acting as the primary key for the referenced table in the constraint need not be explicitly declared to be unique. See [Rules for Using Referential Constraints](#)).

The potential for such disintegrity has the following implications.

- Events that would have been flagged as referential integrity violations by SQL if RI were being enforced are not flagged, and those violations of integrity are permitted *without warning*.
- Similarly, events that would have been flagged as RI errors by FastLoad, MultiLoad, and Teradata Parallel Transporter are permitted against columns defined with Referential Constraints *without warning*.

Temporal Relationship Constraints are another form of Referential Constraint that can sometimes be specified for temporal tables. For information on temporal tables, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

Referential Constraints and Temporal Relationship Constraints are a Teradata extension to the ANSI SQL:2011 standard.

Comparison of Referential Integrity Constraint Types

The following table compares the three referential integrity constraint types supported by Vantage.

| Referential Constraint Type | DDL Definition Clause | Does It Enforce Referential Integrity? | Level of Enforcement |
|-----------------------------|------------------------------|--|---|
| Standard | REFERENCES | Yes | Single row |
| Batch | REFERENCES WITH CHECK OPTION | Yes | All child table rows must match a parent table row. |

The different types have different applications, as the following table documents.

| Referential Constraint Type | Application |
|-----------------------------|--|
| Standard | <ul style="list-style-type: none"> Tests each individual inserted, deleted, or updated row for referential integrity. Not valid for temporal tables. If insertion, deletion, or update of a row would violate referential integrity, then AMP software rejects the operation and returns an error message. Permits special optimization of certain queries. |
| Batch | <ul style="list-style-type: none"> Tests an entire insert, delete, or update request operation for referential integrity. In this context, a batch operation is defined as an implicit SQL transaction. Not valid for temporal tables. If the result of the request violates referential integrity, the system returns an error to the requestor. Permits special optimization of certain queries. |

An Example of How the Optimizer Takes Advantage of Referential Constraints

To understand how the Optimizer can produce more high-performing query plans when a Referential Constraint is defined, consider the following table definitions. The only difference between the second and third tables is that *tb2* defines a Referential Constraint on column *b* referencing column *a* in *tb1*, while *tb3* does not define that constraint.

```

CREATE TABLE tb1 (
  a INTEGER NOT NULL PRIMARY KEY,
  b INTEGER,
  c INTEGER);

CREATE TABLE tb2 (
  a INTEGER NOT NULL,
  b INTEGER,
  c INTEGER,
  CONSTRAINT ref1
  FOREIGN KEY (b) REFERENCES WITH NO CHECK OPTION tb1(a));

CREATE TABLE tb3 (
  a INTEGER NOT NULL,
  b INTEGER,
  c INTEGER);

```

The following EXPLAIN report shows the plan for a query on tables *tb1* and *tb3* when no Referential Constraints have been defined. In particular, notice the join in Step 5.

```

EXPLAIN
SELECT tb1.a, tb3.a, MAX(tb3.c)
FROM tb1, tb3
GROUP BY tb1.a, tb3.a
WHERE tb1.a = tb3.b
ORDER BY 1;

```

Explanation

- 1) First, we lock MyDB.tb3 for read on a reserved RowHash to prevent global deadlock.
- 2) Next, we lock MyDB.tb1 for read on a reserved RowHash to prevent global deadlock.
- 3) We lock MyDB.tb3 for read, and we lock MyDB.tb1 for read.
- 4) We do an all-AMPs RETRIEVE step from MyDB.tb3 by way of an all-rows scan with a condition of ("NOT (MyDB.tb3.b IS NULL)") into Spool 4 (all_amps), which is redistributed by the hash code of (MyDB.tb3.b) to all AMPs. Then we do a SORT to order Spool 4 by row hash. The size of Spool 4 is estimated with low confidence to be 4 rows (100 bytes). The estimated time for this step is 0.03 seconds.
- 5) We do an all-AMPs JOIN step from Spool 4 (Last Use) by way of a RowHash match scan, which is joined to MyDB.tb1 by way of a RowHash match scan with no residual conditions. Spool 4 and MyDB.tb1 are

joined using a merge join, with a join condition of ("MyDB.tb1.a = b"). The result goes into Spool 3 (all_amps), which is built locally on the AMPs. The size of Spool 3 is estimated with index join confidence to be 4 rows (108 bytes). The estimated time for this step is 0.11 seconds.

- 6) We do an all-AMPs SUM step to aggregate from Spool 3 (Last Use) by way of an all-rows scan , grouping by field1 (MyDB.tb1.a ,MyDB.tb3.a). Aggregate Intermediate Results are computed locally, then placed in Spool 5. The size of Spool 5 is estimated with low confidence to be 4 rows (164 bytes). The estimated time for this step is 0.08 seconds.
 - 7) We do an all-AMPs RETRIEVE step from Spool 5 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1 (MyDB.tb1.a). The size of Spool 1 is estimated with low confidence to be 4 rows (132 bytes). The estimated time for this step is 0.08 seconds.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.29 seconds.

The following EXPLAIN shows the query plan for the same query, but this time on tables *tb1* and *tb2*, with a Referential Constraint defined between column *b* of table *tb2* and column *a* of table *tb1*. Notice that the Optimizer has recognized that the join in step 5 of the previous query plan is unnecessary and can be eliminated. The same query plan would be produced if *tb1.a* and *tb2.b* had an explicitly declared standard Referential Integrity constraint.

```
EXPLAIN
SELECT tb1.a, tb2.a, MAX(tb2.c)
FROM tb1, tb2
GROUP BY tb1.a, tb2.a
WHERE tb1.a = tb2.b
ORDER BY 1;
```

Explanation

-
- 1) First, we lock MyDB.tb2 for read on a reserved RowHash to prevent global deadlock.
 - 2) Next, we lock MyDB.tb2 for read.
 - 3) We do an all-AMPs SUM step to aggregate from MyDB.tb2 by way of an all-rows scan with a condition of ("NOT (MyDB.tb2.b IS NULL)" , grouping by field1 (MyDB.tb2.b ,MyDB.tb2.a). Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with low confidence to be 4 rows (164 bytes). The estimated time for this step is 0.07 seconds.

- 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1 (MyDB.tb2.b). The size of Spool 1 is estimated with low confidence to be 4 rows (132 bytes). The estimated time for this step is 0.08 seconds.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.15 seconds.

Rules for Using Referential Constraints

Other than their not actually enforcing referential integrity, most of the rules for Referential Constraints are identical to those documented by [FOREIGN KEY Constraints](#) and REFERENCES constraints.

The exceptions are documented by the following set of rules that apply specifically to the specification and use of Referential Constraints.

- You can specify Referential Constraints for both of the following constraint types.
 - FOREIGN KEY (*FK_column_set*) REFERENCES (*parent_table_PK_column_set*)
 - (*NFK_column_set*) REFERENCES (*parent_table_AK_column_set*)

where NFK indicates non-foreign key and *parent_table_AK_column_set* indicates an alternate key in the parent table.
- Referential Constraint references count toward the maximum of 64 references permitted for a table referenced as a parent even though they are not enforced by the system.
- INSERT, DELETE, and UPDATE statements are not permitted against tables with unresolved, inconsistent, or non-valid Referential Constraints. This rule is identical to the rule enforced for standard and batch RI.
- The candidate key acting as the primary key in the referenced table in the constraint 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.

Validating the Integrity of Base Tables In a Referential Constraint Relationship

If you decide to specify a non-enforced Referential Constraint for a table, the responsibility for ensuring or validating the referential integrity of that table is yours alone. The system does not enforce referential integrity when you specify Referential Constraint relationships between tables, because no declarative constraints are declared in the appropriate column definitions.

From the aspect of integrity assurance, the best way to guarantee the referential integrity of a table without taking advantage of a declarative standard or batch referential constraint is to use a procedural constraint such as a set of triggers to handle inserts, updates, and deletions to the tables in the relationship.

For example, you want to create DELETE/UPDATE triggers on parent tables, and INSERT/UPDATE triggers on child tables to enforce referential integrity. The reasons for preferring declarative constraints over procedural constraints are described briefly in *Teradata Vantage™ - Database Design*, B035-1094.

Also, actively firing triggers can have a greater negative effect on system performance than the simple declarative constraint they are intended to replace.

If you decide not to enforce any form of referential integrity constraint, then you are strongly advised to enforce a set of validation procedures that can detect when and where referential integrity violations occur.

The following scenario uses a basic SQL query to interrogate a table set for violations of the referential integrity rule.

Suppose you create the tables `pk_tbl1` and `softri_tbl1`, with column `b2` of `softri_tbl1` having a Referential Constraint relationship with column `a1` of `pk_tbl1`. In this relationship, table `pk_tbl1` is the parent and table `softri_tbl1` is the child. The DDL for defining these tables might look like the following CREATE TABLE statements.

```
CREATE TABLE pk_tbl1 (
  a1 INTEGER NOT NULL PRIMARY KEY,
  a2 INTEGER);

CREATE TABLE softri_tbl1 (
  b1 INTEGER,
  b2 INTEGER CONSTRAINT softri_1
    REFERENCES WITH NO CHECK OPTION pk_tbl1(a1));
```

Column `softri_tbl1.b2` is an implicit foreign key referencing the primary key column `pk_tbl1.a1`.

Now populate the tables with data as follows.

```
INSERT INTO pk_tbl1 (a1, a2) VALUES (11, 111);
INSERT INTO pk_tbl1 (a1, a2) VALUES (22, 222);

INSERT INTO softri_tbl1 (b1, b2) VALUES (100, 11);
INSERT INTO softri_tbl1 (b1, b2) VALUES (200, 22);
INSERT INTO softri_tbl1 (b1, b2) VALUES (300, 33);
INSERT INTO softri_tbl1 (b1, b2) VALUES (400, 44);
```

The third and fourth inserts into table `softri_tbl1` violate the implicit referential integrity relationship between the tables because the set of distinct values for `softri_tbl1.b2` should be identical to the set of values for `pk_tbl1` (which, as the primary key value set, constitute a distinct set by default). In this scenario, the `softri_tbl1.b2` values 33 and 44 identify the rows that violate the implied referential integrity relationship.

The following SELECT statement is a generic query to test for this type of corruption. The exclusion of foreign key nulls is included in the query because it is not possible to determine what values they represent.

```

SELECT DISTINCT childtable.*
FROM childtable, parenttable
WHERE childtable.fk NOT IN (SELECT pk
                           FROM parenttable)
AND   childtable.fk IS NOT NULL;

```

The scenario defines the following set of correspondences.

| Generic Query Element | Specific Query Element |
|-----------------------|------------------------|
| childtable | softri_tbl1 |
| parenttable | pk_tbl1 |
| childtable.fk | softri_tbl1.b2 |

From these correspondences, the specify query to test for corrupt rows in this scenario would look like the following SELECT statement.

```

SELECT DISTINCT softri_tbl1.*
FROM softri_tbl1, pk_tbl1
WHERE softri_tbl1.b2 NOT IN (SELECT a1
                           FROM pk_tbl1)
AND   softri_tbl1.b2 IS NOT NULL;

*** Query completed. 2 rows found. 2 columns returned.
*** Total elapsed time was 1 second.

```

```

      b1          b2
-----
      300         33
      400         44

```

The report generated by this query returns the rows with the implicit foreign key values 33 and 44, which is what was expected: the set of rows from the child table whose foreign key values do not match any element of the primary key value set from the parent table. This set of non-matching rows must be deleted from the child table to maintain the referential integrity of the database.

Perform this validation query regularly, particularly after any substantial updates are made to the tables in the implied relationship.

Your data can be corrupt for the entire interval between the times you run this query, and you still have the burden of repairing any problems that it detects. The query does not solve the problem of failing to enforce referential integrity in your database, it only allows you to detect whether the referential integrity defined by one primary or alternate key in a parent table and one foreign key in a child table has been violated. Each such unenforced referential relationship must be probed regularly to detect violations.

Scenario for Data Corruption With Referential Constraints

Always exercise caution when manipulating data within an unenforced Referential Constraints environment. If you define referential relationships using unenforced Referential Constraints, then INSERT, DELETE, and UPDATE statements can corrupt data in the parent tables of those relationships if the DML request contains a redundant RI join and the PK-FK row pairs for the eliminated join do not match. The RI join is eliminated based on the assumption that PK=FK, and non-valid results are returned in SELECT results or incorrect updates are made to the database.

For example, suppose you have the following 2 tables related by means of a Referential Constraint between *col_2* in child table *table_2* and *col_1* in parent table *table_1*.

```
CREATE TABLE table_1 (
  col_1 INTEGER NOT NULL,
  col_2 INTEGER
  UNIQUE PRIMARY INDEX (col_1));

CREATE TABLE table_2 (
  col_1 INTEGER,
  col_2 INTEGER,
  FOREIGN KEY (col_2) REFERENCES WITH NO CHECK OPTION table_1(col_1));
```

Insert some rows into *table_1* and *table_2*. Assume that the populated tables now look like this.

| table_1 | | table_2 | | |
|---------|-------|---------|-------|-------|
| PK | | | | FK |
| col_1 | col_2 | | col_1 | col_2 |
| 1 | 2 | | 5 | 6 |
| 2 | 2 | | 6 | 7 |
| 3 | 2 | | 7 | 8 |

The defined referential relationship between the tables specifies that each value for *col_2* in *table_2* should correspond to an identical value in *col_1* of *table_1*.

Because the system is instructed not to enforce referential integrity for this relationship, the unpaired *table_2.col_2* values are accepted for insertion even though they violate the defined constraint on the relationship. At this point, referential integrity errors have been introduced, but equally importantly, the preconditions for further data corruption have been set.

Suppose you now want to update *table_2* using the following UPDATE statement.

```
UPDATE table_2
SET col_1=1
WHERE table_1.col_1=table_2.col_2;
```

Because *table_2.col_2* references *table_1.col_1* WITH NO CHECK OPTION, this update eliminates the *table_1.col_1 = table_2.col_2* join, thus ignoring the specified WHERE clause condition filter, and then incorrectly updates all rows in table *table_2* with a *col_1* value of 1, corrupting the database.

Normalizing Database Tables

Normalizing database tables is different than normalizing a relational database. In this context, to normalize a table is to coalesce 2 Period or derived Period values that meet or overlap. The result of a normalization operation on 2 input Period or derived Period values that meet or overlap is a third Period value that is the relational union of the 2 input Period values.

Note:

In Vantage, any reference to Period columns or a Period data type with respect to normalization operations applies equally to Period data type columns and derived Period columns. A derived Period identifies 2 DateTime columns in a table that constitute a period. A derived Period is not an actual data type.

Any database table can be created as a normalized table. All you need to do to normalize a table is to identify a Period column in the table as the column on which to normalize. This Period column is known as the normalize column for a table. A normalized table does not have 2 rows with the same data values whose normalize column values meet or overlap. This means that if a new row is inserted or an existing row is updated into a normalized table, it is coalesced with all of the existing table rows whose values are the same as those of the new row and whose normalize column values either overlap or meet. All DML updates on a table that is defined as a normalized table normalize the new row with rows that previously existed in the table.

You can optionally specify a set of table columns that the database should ignore when determining the value equivalence of rows for normalization purposes.

For example, the following CREATE TABLE statement specifies the NORMALIZE option where the data values to be considered for normalization do not include values in the *dept_id* 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);
```

For information about using DML statements to process and update normalized tables, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Rules and Restrictions for the NORMALIZE Option

- The column you specify for the NORMALIZE option must have either a Period data type or be a derived Period column.
- If you do not specify an explicit normalization condition, the default is ON MEETS OR OVERLAPS.
- If a table that is to be normalized contains columns with a BLOB, CLOB, JSON, or XML data type, those columns must be explicitly specified in the *normalize_ignore_column_name* list for the NORMALIZE option.

If a table definition contains such a column that is not included in the *normalize_ignore_column_name* list, the system returns an error to the requestor.

- Vantage validates both UNIQUE and PRIMARY KEY constraints with normalized rows. If a normalized row violates a UNIQUE or PRIMARY KEY constraint, the system returns an error to the requestor.
- Vantage validates CHECK constraints on a row inserted into a normalized table, and if the constraint is violated, the system returns an error to the requestor.

This action prevents a security issue that can occur if a constraint is specified on the beginning or end of a normalized Period column. In this case the input row violates the CHECK constraint but the normalized row does not. This situation cannot occur with UNIQUE constraints.

Related Information

For information about the syntax used to create table columns, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

For information about modifying, adding, or dropping table columns, see [ALTER TABLE \(Basic Table Parameters\)](#) and ALTER TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

For detailed information about row-level security, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

CREATE TABLE Indexes

These topics provide supplemental usage information about the CREATE TABLE statement.

For CREATE TABLE syntax information and examples, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE TABLE (Index Definition Clause)

About Primary-Indexing, Row-Partitioning, Column-Partitioning, NoPI Tables, and Secondary Indexes

This CREATE TABLE clause permits you to create 1 or no primary indexes and optional secondary indexes for a table.

A table can have no more than 1 primary index, which can optionally be partitioned. The partitioning option is available for global temporary tables, volatile tables, and standard base tables that have a primary index. Global temporary trace tables cannot be partitioned. See [CREATE TABLE Global and Temporary](#).

If you want a table to have a primary index, you should *always* define the primary index explicitly. The same is true if you do *not* want a table to have a primary index.

Because, with the exception of nonpartitioned and column-partitioned tables (see [Nonpartitioned NoPI Tables](#) and [Column-Partitioned Tables](#)), Vantage assigns rows to AMPs based on the row hash of their primary index value, it is important to select a column set that distributes table rows fairly evenly when a nonunique primary index is defined for a primary-indexed table. This is critical whether the table is partitioned or not. When you allow the database to select a column set as the primary index for a table by default, you have no control over the evenness of its distribution across the AMPs, and this may result in a table with performance bottlenecks.

The Optimizer uses index definitions to plan how to access data in the least costly manner and AMP software uses them to physically access rows on disk in the least costly manner possible.

To define additional secondary indexes on a table after it has been created, use the CREATE INDEX statement (see [CREATE INDEX](#)).

To create a join index incorporating this table, use the CREATE JOIN INDEX statement (see [CREATE JOIN INDEX](#)).

To create a hash index on a table, use the CREATE HASH INDEX statement (see [CREATE HASH INDEX](#)).

Note that you cannot reference BLOB, CLOB, Period, or Geospatial columns in any index definition. This means that none of the following clauses can specify a reference to a BLOB, CLOB, Period, or Geospatial column.

- INDEX (see information about NUSIs in the topic [Secondary Indexes](#)).

- PRIMARY INDEX (see [Primary Indexes](#) and [Partitioned and Nonpartitioned Primary Indexes](#)).
- UNIQUE INDEX (see information about USIs in the topic [Secondary Indexes](#)).
- UNIQUE PRIMARY INDEX (see [Primary Indexes](#) and [Partitioned and Nonpartitioned Primary Indexes](#)).

For more information about primary and secondary indexes, see *Teradata Vantage™ - Database Design*, B035-1094.

Index Types

Index determination and definition is an important aspect of database design. See *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - SQL Fundamentals*, B035-1141

The basic index types are:

- No primary index
- Primary indexes
- Hash and join indexes
- Secondary indexes

No primary index

For this and column-partitioned tables and join indexes, the absence of a primary index is counted as an index type.

- Nonpartitioned NoPI
- Column-partitioned

Primary indexes

Primary indexes include:

- Unique nonpartitioned
- Nonunique nonpartitioned primary
- Single-level unique partitioned primary
- Multilevel unique partitioned primary

Hash and join indexes

The various join indexes are not necessarily mutually exclusive types. Both multitable and single-table simple join indexes can also be sparse, for example. A join index composed of virtual rows, with multiple fixed column sets appended to a single repeating column set is said to be row compressed. Whether compressed or not compressed, a join index can be any of the following types.

- Row-compressed
- noncompressed
- Column-partitioned
- Partitioned primary index

- Single-table simple
- Single-table aggregate
- Single-table sparse
- Multitable simple
- Multitable aggregate
- Multitable sparse

Secondary indexes

Secondary indexes include:

- Unique
- Nonunique hash-ordered on all columns with the ALL option
- Nonunique hash-ordered on a single column with the ALL option
- Nonunique value-ordered on a single column with the ALL option
- Nonunique hash-ordered on all columns without the ALL option
- Nonunique hash-ordered on a single column without the ALL option
- Nonunique value-ordered on a single column without the ALL option

Primary Indexes

Vantage hashes the primary index value set to determine on which AMPS the rows of a primary-indexed table are to be stored. For information about how the rows of tables and join indexes that do not have a primary index are distributed to the AMPs, see *Teradata Vantage™ - Database Design*, B035-1094.

Selection and update operations that use the primary index column in a WHERE clause use hashing to determine the location of any row with maximum efficiency.

You cannot compress the values of columns that are members of the primary index column set, nor can you compress the values of the columns of a partitioning expression.

You cannot include BLOB, CLOB, JSON, XML, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Geospatial, ARRAY, VARRAY, Period, or row-level security constraint columns in the primary index column set, nor can you include row-level security constraint columns. Any table that contains BLOB, CLOB, Geospatial, ARRAY, VARRAY, Period, or row-level security constraint columns must contain at least 1 other non-BLOB, non-CLOB, non-Geospatial, non-ARRAY/VARRAY, or non-Period column that can be used to define its primary index.

Nonpartitioned NoPI Tables

Tables that do not have a primary index and are not column-partitioned, referred to as nonpartitioned NoPI tables, should generally be used for only 2 reasons.

- As staging tables for FastLoad and Teradata Parallel Data Pump array INSERT load operations. For more information, see *Teradata Vantage™ - Database Design*, B035-1094.

After the data is loaded into these tables, you can use INSERT ... SELECT, MERGE, or UPDATE ... FROM to copy the rows to their destination primary-indexed tables. INSERT ... SELECT and UPDATE statements can also be used to copy rows from a primary-indexed source table into a NoPI target table, while MERGE statements cannot.

- As temporary holding, or sandbox, tables when an appropriate primary index has not yet been defined for the table they will eventually populate.

There are 2 types of NoPI tables.

- Nonpartitioned NoPI tables, which the current topic describes.
- Column-partitioned tables, which are described in [Column-Partitioned Tables](#).

You can use the following SQL DML statements to manipulate nonpartitioned NoPI table data.

- DELETE
- INSERT
- SELECT
- UPDATE

Nonpartitioned NoPI tables have the following restrictions.

- You cannot create a nonpartitioned NoPI table as a SET table.
The unalterable default table type for nonpartitioned NoPI tables in all session modes is MULTiset.
- You cannot specify a column name list following the NO PRIMARY INDEX specification.
- If you do not specify PRIMARY INDEX (*column_list*) or NO PRIMARY INDEX in your CREATE TABLE statement, then whether the table is created with a primary index depends on whether a PRIMARY KEY or UNIQUE constraint is specified for any of the columns and on the setting of the DBS Control parameter PrimaryIndexDefault. For details and exceptions, see *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - Database Utilities*, B035-1102.

The default setting for PrimaryIndexDefault is D. If you do not specify either an explicit PRIMARY INDEX or NO PRIMARY INDEX option, Vantage creates a UPI on a declared PRIMARY KEY, and if none exists, on the first column defined with a UNIQUE attribute. If none is found, Vantage creates the table with a NUPI on the first index-eligible column that is defined.

- You cannot specify partitioning of the primary index for a nonpartitioned NoPI table because it cannot have a primary index to partition. However, you can define a NoPI table to have column-partitioning. See [Column-Partitioned Tables](#).
- Nonpartitioned NoPI tables cannot have a permanent journal.
- Nonpartitioned NoPI tables cannot have an identity column.
- Hash indexes cannot be specified on nonpartitioned NoPI tables because hash indexes inherit the primary index of their underlying base table, and nonpartitioned NoPI tables have no primary index.
- SQL MERGE statements cannot update or insert into either nonpartitioned NoPI tables or column-partitioned target tables.

SQL MERGE statements can update or insert into a primary-indexed target table from either a nonpartitioned or a column-partitioned source table.

- You cannot load rows into a nonpartitioned NoPI table or any normalized table using the MultiLoad utility.

Note:

You can load rows into a nonpartitioned NoPI table using the FastLoad utility with the exception of normalized NoPI tables, Teradata Parallel Data Pump array INSERT operations, and INSERT ... SELECT statements.

You can define all of the following features for nonpartitioned NoPI tables.

- TRANSACTIONTIME columns, VALIDTIME columns, or both.
- Fallback
- Secondary indexes
- Join indexes
- UNIQUE column constraints
- CHECK constraints
- PRIMARY KEY and FOREIGN KEY constraints
- Triggers
- BLOB, CLOB, ARRAY, VARRAY, UDT, Period, Geospatial, and row-level security constraint columns.

Note:

There is a limit of approximately 64K rows per row hash value for LOBs. Because there is normally only 1 row hash value per AMP for nonpartitioned NoPI tables, there is also a limit of approximately 64K rows per AMP for nonpartitioned NoPI tables that contain columns typed as BLOBs or CLOBs.

You can define any of the following table types as nonpartitioned NoPI tables.

- Nonpartitioned base data tables
- Global temporary tables
- Volatile tables
- Queue tables (see [CREATE TABLE \(Queue Table Form\)](#)).
- Error tables (see [CREATE ERROR TABLE](#)).

You cannot define any of the following table types as an nonpartitioned NoPI table.

Column-Partitioned Tables

The term Teradata Columnar describes tables with a Primary Index (PI), Primary AMP Index (PA), or No Primary Index (NoPI) that are column-partitioned. These tables are referred to as column-partitioned tables. Teradata Columnar is disabled by default and it can only be enabled by Teradata Services personnel.

Teradata Columnar partitioning is a physical database design implementation that consists of an integrated set of options that support column partitioning, columnar storage, autocompression, and other capabilities that you can specify for column-partitioned tables. Columnar storage packs the data values of a column partition into a series of containers, which significantly reduces the number of row headers required to store the data.

You can store data in a column partition using either traditional row storage (indicated by the ROW keyword) or columnar storage (indicated by the COLUMN keyword), or a mix of both.

You can define the following features for column-partitioned tables.

- TRANSACTIONTIME columns, VALIDTIME columns, or both.
- Fallback
- Secondary indexes
- Join indexes
- UNIQUE column constraints
- CHECK constraints
- PRIMARY KEY and FOREIGN KEY constraints
- Identity columns
- AUTO COMPRESS or NO AUTOCOMPRESS in the column definition list.
- AUTO COMPRESS or NO AUTOCOMPRESS in the PARTITION BY clause.
- Triggers
- BLOB, CLOB, XML, ARRAY, VARRAY, UDT, Period, Geospatial, and row-level security constraint columns.

Note:

There is a limit of approximately 64K rows per row hash value for LOBs. Because there is normally only 1 row hash value per AMP for column-partitioned tables, there is also a limit of approximately 64K rows per AMP for column-partitioned tables that contain columns typed as BLOBs or CLOBs.

You cannot define permanent journaling for column-partitioned tables.

You cannot define any of the following table types as column-partitioned tables.

- Global temporary tables
- Volatile tables
- Global temporary trace tables
- Queue tables
- Error tables

The default syntax for Teradata Columnar incorporates the following features.

- The default index definition clause specification for column-partitioned tables is NO PRIMARY INDEX.

The setting of the DBS Control parameter PrimaryIndexDefault does not affect this default.

- Vantage implicitly assigns each individual column to its own column partition.

While this is the default method of storing column partition values, you can also explicitly group multiple columns into a column partition.

- Vantage determines whether column partitions are stored in row or column format.

While this is the default format for storing column partition values, you can also explicitly specify either COLUMN or ROW format for a partition.

You can further define multiple columns to be assigned to the same column partition either in the column list for the table or in its partitioning expression. You do this by delimiting columns to be grouped into the same column partition between parenthesis characters.

- Vantage implicitly detects and applies a number of different autocompression methods to enable more efficient storage. Autocompression methods are only applied if they reduce the size of the physical row.

Note:

This is the default compression option. You can also explicitly specify NO AUTO COMPRESS if you do not want Vantage to implicitly assign autocompression methods to your column partitions.

See CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for details of the syntax used to create a column-partitioned table. Column storage packs the values of a column partition into a series of containers (column partitions stored using COLUMN format) or subrows (column partitions stored using ROW format). COLUMN format significantly reduces the number of row headers that would otherwise be required per stored container.

Teradata Columnar is optimal for:

- Improving the I/O performance of accessing a subset of the columns from a table either for evaluating predicates or projections. Because sets of one or more columns can be stored in separate column partitions, only the column partitions that contain the columns referenced by the query need to be accessed from storage thereby significantly reducing the number of data blocks that are read.
- Sandbox tables when an appropriate primary index has not yet been defined for the table the rows will eventually populate.

The expected usage for a column-partitioned table is the case where its partitions are loaded using an INSERT ... SELECT statement (possibly followed by intermittent minor maintenance) and then used to run data mining analytics.

After the analytical work or data mining has been completed, the expectation is that you will either delete the entire table or that you will delete individual row partitions from it.

Following that, the expectation is that the rows will be loaded into a staging table and the identical scenario will repeat with new data inserted into the column-partitioned table.

This type of scenario is referred to as an *insert once* scenario. Column-partitioned tables are not intended to be used for OLTP activities.

For more information about how to optimize your use of column-partitioned tables, see [Usage Guidelines for Column-Partitioned Tables](#), [Rules and Restrictions for Column-Partitioned Tables](#), and [Performance Issues for Column-Partitioned Tables](#).

You can partition a column-partitioned table in 3 different ways.

- By column
- By row
- By a combination of columns and rows using multilevel partitioning

Column partitioning enables column partition elimination based on the columns that are required to process a query. If a column is not needed by a request, the column partition containing that column does not need to be read, significantly enhancing query performance.

Column partitioning also enables DML requests to efficiently access selected data from column partitions, significantly reducing query I/O.

You can use the following SQL DML statements to manipulate column-partitioned table data.

- DELETE
- INSERT
- SELECT
- UPDATE

Note:

The MERGE statement is not included in this list. You cannot use MERGE statements to insert rows into a column-partitioned table nor can you use MERGE statements to update the data in a column-partitioned table. This is because you must specify the primary index of the target table when you use a MERGE statement to insert or update the data of a table, and column-partitioned tables do not have a primary index.

Usage Guidelines for Column-Partitioned Tables

The following usage guidelines apply to column-partitioned table applications.

- Queries that access a small, but variable, subset of columns run more efficiently against a column-partitioned table when compared with a nonpartitioned table. The majority of queries against a column-partitioned table should be selective on a variable subset of columns, and project a variable subset of the columns, where the subset accessed is less than 10% of the partitions for any particular query. The number of partitions that need to be accessed for queries should not exceed the number of available column partition contexts.
- Use column partitioning for tables that have the following properties.
 - They are accessed using analytic queries.
 - They are not frequently refreshed using bulk data loading methods.
 - The table partitions are seldom modified in the interim.
- Do not use column-partitioned tables for highly volatile data.

- Place the COLUMN partitioning level at the lowest partitioning level possible, ideally at a lower level than any ROW partitioning levels in the partitioning expression. The COLUMN partitioning level should be the first partitioning level you specify.

The following are some of the considerations that might lead to putting column partitioning at a lower level.

- Potential improvements for cylinder migration.
- Temperature-based block compression effectiveness for both hot and cold data.
- For queries that frequently access columns, but where the specific set of columns accessed varies from request to request, you should place the frequently accessed columns into single-column partitions.

This action enables the Optimizer to use column partition elimination to optimize queries against the table.

- For queries that frequently access the same set of columns across the requests in a workload, you should group the frequently accessed columns into the same partition.
- If you find that autocompression is effective for a column, consider placing that column in a single-column partition even if it is not frequently accessed.
- Autocompression is most effective for single-column partitions with COLUMN format, less so for multicolumn partitions, particularly as the number of columns increases, and not effective for column partitions with ROW format.
- Group columns into a column partition for applications where either of the following is true.
 - Queries frequently access the columns.
 - Queries *do not* frequently access the columns, and autocompression of the individual columns or subsets of columns is not effective.
- Use COLUMN format for narrow column partitions, especially if you find that autocompression is effective for a partition.

If the system-determined format is not COLUMN for a column partition, but you determine that COLUMN is more appropriate, specify COLUMN explicitly when you create or alter the table.

You might need to specify COLUMN format explicitly for a column partition that has a column with a VARCHAR, CHARACTER SET VARGRAPHIC, or VARBYTE data type defined with a large maximum value, but where the partition values are actually relatively short in most cases.

This can happen when the system-determined format is ROW because of a large maximum value length.

- Use ROW format for wide column partitions because it has less overhead than a container that holds only 1 or a few values.

If the system-determined format is not ROW for a column partition, but you determine that ROW is more appropriate, specify ROW explicitly when you create or alter the table.

- The Optimizer uses the DBS Control parameter PPICacheThrP to determine the number of available file contexts that can be used at a time to access a partitioned table.

| IF PPICacheThrP determines the number of available file contexts to be ... | THEN Vantage considers this many file contexts to be available ... |
|--|--|
| < 8 | 8 |
| > 256 | 256 |

Vantage uses the number of file contexts as the number of available column partition contexts for a column-partitioned table.

Note:

Vantage might associate a file context with each column partition context for some operations, and in other cases it might allocate a buffer with each column partition context.

Ideally, the number of column partition contexts should be at least equal to the number of column partitions that need to be accessed by a request. Otherwise, performance can degrade because the required column partitions cannot be read at one time.

Performance and memory usage can be impacted if PPICacheThrP is set too high, which can lead to memory thrashing or a system crash. At the same time, the benefits of partitioning can be lessened if the value for the DBS Control parameter PPICacheThrP is set unnecessarily low, causing performance to degrade significantly.

The default is expected to be applicable to most workloads, but you might need to make adjustments to get the best balance.

-
- You should accept the default DATABLOCKSIZE or the default data block size as defined by the DBS Control parameter PermDBSize for a column-partitioned table unless performance analysis indicates otherwise.
 - If you expect to add rows to a table incrementally, you should consider allocating some additional free space if a column-partitioned table has small internal partitions. This can occur, for example, if a table is also row-partitioned.

To allocate additional space, you can either specify a value for the FREESPACE option that is larger than the system default when you create the table, or accept the default as set by the DBS Control parameter FreeSpacePercent if your DBA has set it to a higher default value. For more information about FreeSpacePercent, see *Teradata Vantage™ - Database Utilities*, B035-1102.

If you plan to load the table with rows using a large INSERT ... SELECT statement, and the internal partitions for the table are either large or unpopulated, little or no free space should be required.

The reserved free space enables table data to expand on current table cylinders, preventing or delaying the need for additional table cylinders to be allocated, which prevents or delays data migration associated with new cylinder allocations.

- Keeping new table data physically close to existing table data, and avoiding data migrations, can improve overall system performance.

Rules and Restrictions for Column-Partitioned Tables

The rules for column-partitioned tables are grouped into the following topics:

- [General Rules for Column-Partitioned Tables](#)
- [Rules for Column Partitions in a Column-Partitioned Table](#)
- [Rules for Partition Formats in a Column-Partitioned Table](#)
- [Rules for Primary Indexes and Their Defaults for a Column-Partitioned Table](#)
- [Rules for Specifying Column Grouping in a Column-Partitioned Table](#)
- [Rules for Using the ADD Option for the Partitioning Levels of a Column-Partitioned Table](#)
- [Rules for the Maximum Partition Numbers and Partitioning Levels of a Column-Partitioned Table](#)
- [Rules for Using Client Load Utilities With A Column-Partitioned Table](#)

General Rules for Column-Partitioned Tables

- You can only define standard base data tables as column-partitioned tables.

You *cannot* define any of the following table types as column-partitioned tables.

- Global temporary tables
- Volatile tables
- Global temporary trace tables
- Queue tables

Note:

You *can* specify NO QUEUE for a column-partitioned table

- Error tables
- You cannot create a column-partitioned table as a SET table.

The default and only valid table type for column-partitioned tables in all session modes is MULTiset. This means that if you do not specify an explicit table type of MULTiset, a column-partitioned table is always a MULTiset table by default.

- If you specify the COLUMN keyword for a partitioning level in a partitioning expression of a PARTITION BY clause, the table is column-partitioned.

You cannot specify COLUMN for more than one partitioning level in a table definition.

- A column-partitioned table can be assigned row-level security privileges.

Columns of a column-partitioned table can be defined as row-level security columns if they qualify as row-level security constraints. See *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 and *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for more information about row-level security constraints and privileges.

Autocompression is applicable to a column partition that contains 1 or more row-level security constraint columns.

- You can specify either `AUTO COMPRESS` or `NO AUTO COMPRESS` as the default for column partitions. If you do not explicitly specify either, Vantage uses `AUTO COMPRESS`.
- You can group partition columns in either the column list or in the `PARTITION BY` clause.

Grouping partition columns in the column list enables a simpler, but less flexible, specification of column groupings than grouping in the `COLUMN` specification of a `PARTITION BY` clause.

- Vantage derives a partitioning `CHECK` constraint from the partitioning level definitions for a column-partitioned table. For further information about partitioning `CHECK` constraints, see [Partitioning CHECK Constraints for Partitioned Tables With 2-Byte Partitioning](#) and [Partitioning CHECK Constraint for Partitioned Tables with 8-Byte Partitioning](#).

The text for this partitioning constraint cannot exceed 16,000 characters.

- You cannot specify `CHARACTER SET KANJI1` for a character data column in a column-partitioned table.

Instead, specify `CHARACTER SET UNICODE`.

- A `CREATE TABLE ... AS source_table` statement copies the indexes and partitioning for the column-partitioned source table to the target table if you do not specify an index list for the target table.

If Vantage copies the partitioning and the source partitioning includes column partitioning, it copies the column partitioning definition as part of the partitioning with the following exception: if you specify grouping in the column list for the target table, Vantage uses the specified grouping for the column partitioning of the new table instead of the grouping defined for the source table.

- A `CREATE ERROR TABLE` statement for a column-partitioned table creates an error table associated with that table.

The system creates the error table the same as it would for a non-column-partitioned table except that the error table is created as a nonpartitioned `NoPI` table.

- You cannot create a hash index on a column-partitioned tables because hash indexes inherit the primary index of their underlying base table, and column-partitioned tables have no primary index.
- You cannot use `MERGE` statements to update the rows of a column-partitioned target table or to insert rows into a column-partitioned target table.

Note:

`MERGE` statements can update or insert into a primary-indexed target table from a column-partitioned source table.

- If a table is partitioned, its fallback rows are partitioned identically to its primary data rows.

If the primary data rows have a primary index or primary AMP index, the fallback data rows have the same primary index or primary AMP index.

If the primary data rows do not have a primary index or primary AMP index, the fallback data rows also do not have a primary index or primary AMP index.

- You can define a large variety of partitioning expressions and column groupings for column partitioning with a large range in the number of combined partitions. You must consider the usefulness of defining a particular partitioning and its impact on performance and storage.

See [Performance Issues for Column-Partitioned Tables](#) for column partition-related performance considerations.

- The following table presents the rules for using the NO AUTO COMPRESS option.

| Option | Description |
|--|---|
| AUTO COMPRESS for a column partition | applies autocompression for physical rows if it finds a compression method that reduces the size of a partition. If Vantage cannot find a compression method that reduces the size of a partition, it does not apply a compression method. |
| NO AUTO COMPRESS for a column partition in the column list for the table | does not apply autocompression for physical rows. In this case, Vantage <i>does</i> apply any user-specified compression and, for column partitions with COLUMN format, row header compression, for the column partition. |

Rules for Column Partitions in a Column-Partitioned Table

- You cannot specify a column more than once within a column partition, nor can you specify a column to be in more than one column partition.
- The number of defined column partitions for a column partitioning level is the number of user-specified column partitions plus two column partitions reserved for internal use.

Vantage uses one of the reserved partitions as a sparse bit map to indicate deleted rows and the other is reserved for future use. There is always at least one column partition number that is not assigned to a column partition.

- The number of defined partitions for a row partitioning level is the number of user-defined row partitions specified by the PARTITION BY clause.

If you do not specify a RANGE_N or CASE_N function in the PARTITION BY clause, Vantage uses a maximum of 65,535 row partitions for the table by default.

- Vantage initially assigns numbers to user-specified column partitions starting at 1 and increasing upward in increments of 1 up to a maximum of *cs*.

The following definitions apply to this rule.

| Variable | Definition |
|-----------|---|
| <i>cs</i> | Number of user-specified column partitions. |
| <i>cm</i> | Maximum partition number for the column partitioning level. |

Vantage assigns an internal partition a partition number of $cm-1$ and the delete column internal partition is assigned a partition number of cm .

At first, no column partitions are assigned to column partition numbers $cs+1$ to $cm-2$. Initially, there is at least 1 unused column partition number because $cm-cs-2$ is greater than 0.

As you drop or alter partitions, there can be gaps in the numbering. As you add or alter column partitions, unused column partition numbers between 1 and $cm-2$ can be assigned to the newly added or altered column partitions as long as 1 column partition number remains unused. This is necessary because at least 1 column partition number must be available for use by ALTER TABLE to alter a column partition.

Vantage uses the column partition number to compute the combined partition number for a column partition value of a table row. Apart from that, there is no significance to the column partition number assigned to a column partition.

Rules for Partition Formats in a Column-Partitioned Table

- If you specify COLUMN format for a column partition, Vantage stores 1 or more column partition values in a physical container.

A column partition value consists of the values of the columns contained in the column partition.

- The following rule applies to autocompression for a column partitioning level.

| Autocompression Option or Default Column Partitioning Level | Override Setting for Partition |
|---|--------------------------------|
| COLUMN AUTO COMPRESS | NO AUTO COMPRESS. |
| COLUMN NO AUTO COMPRESS | AUTO COMPRESS. |

- If you specify COLUMN without also specifying AUTO COMPRESS or NO AUTO COMPRESS for a column partitioning level, Vantage assigns AUTO COMPRESS for the partitioning level.
- If you specify ROW format for a column partition, Vantage stores only 1 column partition value in a physical subrow.

A subrow is a standard Teradata row format, and the term is used to emphasize that it is part of a column partitioning.

- If you specify a column or constraint grouping or both with COLUMN format in the column list, the grouping defines a column partition and Vantage stores 1 or more column partition values in a physical container using COLUMN format.

If you specify a column or constraint grouping or both with ROW format in the column list, the grouping defines a column partition that stores only 1 column partition value in a physical subrow using ROW format.

If you specify neither COLUMN nor ROW format for a column or constraint grouping, the grouping defines a column partition and Vantage determines whether to use a COLUMN or ROW format for that column partition.

- If you do not specify an explicit COLUMN or ROW format for a partition, Vantage makes the determination for you implicitly.

When the COLUMN or ROW format is system-determined, Vantage bases its choice of format on the size of a column partition value for the column partition and other factors such as whether a column partition value for the column partition has fixed or variable length and whether the column partition is a single-column or multicolumn partition.

Vantage generally assigns a COLUMN format to a narrow column partition and assigns a ROW format to a wide column partition. In this context, a column partition is considered to be narrow when its size is roughly 256 or fewer bytes.

The width of a variable length column in a column partition is estimated as

$$\left(\frac{\text{maximumlength}}{3}\right) + 2$$

You can submit a HELP COLUMN statement or select the ColumnPartitionFormat column from the *DBC.ColumnsV(X)* view to determine which format Vantage elected to use for a column partition.

Rules for Primary Indexes and Their Defaults for a Column-Partitioned Table

- A column-partitioned table can have a primary index, a primary AMP index, or no primary index.
- If you do not specify a PRIMARY INDEX or PRIMARY AMP clause or if you do specify a NO PRIMARY INDEX clause and you also specify a PARTITION BY clause, the partitioning must specify a COLUMN partitioning level.
- If you do not specify a PRIMARY INDEX (*column_list*) clause, PRIMARY AMP INDEX clause, or NO PRIMARY INDEX clause explicitly in your CREATE TABLE statement, but you do specify a PARTITION BY clause, then Vantage creates the table without a primary index.

Rules for Specifying Column Grouping in a Column-Partitioned Table

- When you do not specify a column grouping for a COLUMN specification, Vantage defines a separate column partition for each column and column group specified in the column list for a CREATE TABLE statement.
- Grouping columns in a COLUMN specification of the PARTITION BY clause enables you to specify which partitions columns belong to and also enables you to specify the display order of those columns when selecting the columns from the table when you specify an ASTERISK character in the select list.

You can also group columns in the column list for a table. Grouping in the column list allows for a simpler, but less flexible, specification of column groupings than grouping them in the COLUMN specification of a PARTITION BY clause.

You cannot group columns in both the column list and the PARTITION BY clause.

- When you specify a column grouping for a COLUMN specification, you can only specify the name of columns that are defined in the same CREATE TABLE statement.

If you attempt to specify a column by something other than its name as specified in the column list, the system returns an error to the requestor.

- The following column grouping rules apply when the grouping is defined in the column list of the table definition rather than in the PARTITION BY clause.
 - A column partition either has COLUMN format or it has ROW format.

You cannot mix both formats in the same column partition.

However, different partitions of a column-partitioned table can have different formats. For example, the partitions of a column-partitioned table can have all COLUMN format and be stored in containers, all ROW format and be stored in subrows, or 1 using COLUMN format and the others using ROW format.

- If you specify COLUMN as a partitioning level of a PARTITION BY clause, it does not specify column grouping, and if a column definition is not delimited by parentheses defining a group, Vantage treats each column as a single-column partition with autocompression and system-determined COLUMN or ROW format.
- Vantage defines a column partition for each non-group and group column partition specified in the column group list.

To specify the column partition format (COLUMN or ROW) or NO AUTO COMPRESS for a column partition defined on a single column, the column definition must be delimited by parentheses as a single-column group.

You can also group column definitions into a column partition using parentheses to delimit the group.

You can optionally group one or more column definitions in a COLUMN specification in the PARTITION BY clause.

- If you specify a column grouping with COLUMN format in the column list, the grouping defines a column partition, and Vantage stores one or more column partition values in a physical container.
- If you specify a column grouping with ROW format in the column list, the grouping defines a column partition and only one column partition value is stored in a physical subrow.
- If you specify neither COLUMN format nor ROW format for a column list column grouping, the grouping defines a column partition and Vantage determines whether to assign COLUMN or ROW format to it.
- You can only specify column grouping in the column list clause if you also specify COLUMN partitioning in the PARTITION BY clause.

This means that you cannot group columns or constraints in the column list for tables that are not column-partitioned.

- You cannot specify column grouping in both the column list and in the COLUMN option of the PARTITION BY clause of the same CREATE TABLE statement. All grouping must be specified in the PARTITION BY clause of the request or it must all be specified in the column list of the request.

- You can specify either the same or different column grouping for a target table as the source table has in the column list of a CREATE TABLE ... AS statement.
- The following table presents the rules for using the ALL BUT option to group columns.

| IF ... | THEN Vantage ... |
|---------------------------|---|
| you specify ALL BUT | <ul style="list-style-type: none"> ◦ defines a single-column partition with autocompression, unless default is NO AUTO COMPRESS, for any column that is not specified in the column group list. ◦ defines a system-determined COLUMN or ROW format for any column that is not specified in the column group list. |
| do not specify ALL BUT | groups any columns that you do not specify in the column group list into 1 column partition with autocompression and a system-determined COLUMN or ROW format. |

Rules for Using the ADD Option for the Partitioning Levels of a Column-Partitioned Table

- You can specify an ADD option for a partitioning level. The BIGINT number following the ADD keyword plus the number of partitions defined for the level is the maximum number of defined partitions allowed for that level.

The maximum is 9,223,372,036,854,775,807.

For single-level partitioning, the maximum combined partition number is the same as the maximum partition number for the partitioning level.

- If you do not specify an ADD option for a column partitioning level, and the level is the only partitioning level, the maximum number of partitions, including the 2 for internal use, is 65,534.
- If you do not specify an ADD option for a column partitioning level and the following list of items is also true, the partitioning is stored using 2 bytes in the row header and the maximum number of partitions for the column partitioning level is the number of column partitions defined plus 10. The default for this case is ADD 10.
 - The table also has 1 or more row partitioning levels
 - At least 1 of the row partitioning levels does not specify the ADD option

The maximum number of partitions for the column partitioning level is the largest value that does not cause the partitioning to be 8-byte partitioning; otherwise, the maximum number of partitions for the column partitioning level is the largest value that does not exceed 9,223,372,036, 854,775,807.

For single-level partitioning, the maximum combined partition number is the same as the maximum partition number for this partitioning level.

- For each row partitioning level that does not specify the ADD option in level order, Vantage determines the maximum number of partitions for that row partitioning level as follows.
 - Using the number of row partitions defined as the maximum for this and any lower row partitioning level without an ADD clause, the partitioning is 2-byte partitioning and is the largest value that does not cause the partitioning to be 8-byte partitioning.

- Otherwise, the maximum number of partitions for this level is the largest value that does not exceed 9,223,372,036,854,775,807.

If there is no such largest value, the system returns an error to the requestor.

- You can specify ADD 0 for a partitioning level to specify explicitly that the maximum number of partitions for this level is the same as the number of defined partitions.

This is useful for a column partitioning level if you want to override the default of ADD 10 to enable other levels to have more partitions.

This can also be useful for a row partitioning level if you want a lower level that does not specify the ADD clause to have any excess partitions.

- The following table summarizes to which partitioning level any excess partitions are added after the partitions are initially assigned explicitly in the CREATE TABLE statement.

| IF partitioning is ... | AND ... | THEN Vantage adds as many leftover combined partitions as possible to ... |
|------------------------|--|--|
| single-level | | the only row or column partitioning level. If you specify an ADD clause, Vantage overrides it. |
| multilevel | all the row partitioning levels have an ADD clause, but there is a column partitioning level without an ADD clause, | the column partitioning level. This does not need to be added at the first partitioning level. |
| | a column partitioning level and at least 1 of the row partitioning levels does not have an ADD clause, including the case where none of the row partitioning levels has an ADD clause specified, | the first row partitioning level without an ADD clause after using a default of ADD 10 for the column partitioning level. This is repeated for each of the other row partitioning levels without an ADD clause, if any, in level order. |
| | a column partitioning level has an ADD clause and at least 1 of the row partitioning levels does not have an ADD clause there is no column partitioning level and at least 1 of the row partitioning levels does not have an ADD clause. This includes the case where none of the row partitioning levels has an ADD clause specified. | the first row partitioning level without an ADD clause. This is repeated for each of the other row partitioning levels without an ADD clause, if any, in level order. |
| | all of the partitioning levels have an ADD clause or after applying any leftover combined partitions as listed in the next column of this table. | the first row or column partitioning level and overrides the ADD clause for the first partitioning level if one is specified. Vantage repeats this for each of the other levels, if any, from the second level to the last level. |

Rules for the Maximum Partition Numbers and Partitioning Levels of a Column-Partitioned Table

- The maximum partition number for row partitioning is the same as the maximum number of partitions for that level.
- The maximum partition number for a column partitioning level is 1 more than the maximum number of column partitions for that level.

This ensures that there is at least 1 unused column partition number that is always available for altering a column partition.

- Specifying a maximum number of partitions for a partitioning level that is larger than the number of defined partitions for that level can enable the number of defined partitions for that level to be increased using an ALTER TABLE statement.

Note:

This does *not* mean that you can increase the maximum number of partitions for a level.

- The maximum number of partitions for a row partitioning level must be *at least 2*.

An error occurs when only 1 partition is defined for a row partitioning level with an ADD 0 or with no ADD option and the maximum is not increased to at least 2.

- The number of combined partitions for a table is the product of the number of partitions defined by each partitioning level.
- The *maximum* number of combined partitions for a table is the product of the maximum number of partitions defined for each partitioning level.

For single-level partitioning, the maximum number of combined partitions is the same as the maximum number of partitions for the partitioning level.

If a table is defined with column partitioning, the maximum number of combined partitions is smaller than the maximum combined partition number. Otherwise, it is the same.

- The maximum combined partition number for a table is the product of the maximum partition numbers for each partitioning level, up to 9,223,372,036, 854,775,807.

For single-level partitioning, the maximum number of combined partitions is the maximum number of partitions for this partitioning level.

- If the maximum combined partition number for a table is greater than 65,535, Vantage the partition number in an 8-byte field in the row header. This is referred to as 8-byte partitioning.

If the maximum combined partition number for a table is less than or equal to 65,535, Vantage stores the partitioning in a 2-byte field in the row header. This is referred to as 2-byte partitioning.

Single-level column partitioning with no ADD option is stored in a 2-byte field in the row header.

- For 2-byte partitioning, the maximum number of partitions for the first level is the largest value that does not cause the maximum combined partition number to exceed 65,535. This is repeated for each of the other levels, if any, from the second level to the last level.
- For 2-byte partitioning, the maximum number of partitioning levels is 15.
- For 8-byte partitioning, the maximum number of partitions for the first level is the largest value that does not cause the maximum combined partition number to exceed 9,223,372,036,854,775,807. This is repeated for each of the other levels from the second level to the last level.
- For 8-byte partitioning, the maximum number of partitioning levels is 62.

If the maximum column partition number is more than 2 at 1 or more levels, the number of partitioning levels may be further limited because of the limit placed by the rule that the product of the maximum combined partition numbers at each level cannot exceed 9,223,372,036,854,775,807.

Rules for Using Client Load Utilities With A Column-Partitioned Table

- You cannot load rows into a column-partitioned table using the MultiLoad or FastLoad utilities.
- You can load rows into a column-partitioned table using Teradata Parallel Data Pump array INSERT operations.

You can also use INSERT ... SELECT statements to load rows from a source table into a column-partitioned target table. The source can be a primary-indexed table, a column-partitioned table, or an nonpartitioned NoPI table.

For more information about using the Teradata Parallel Data Pump utility to load rows into a column-partitioned table, see *Teradata® Parallel Data Pump Reference*, B035-3021 and *Teradata Vantage™ - Database Design*, B035-1094.

Performance Issues for Column-Partitioned Tables

Column partitioning is a physical database design choice that can improve the performance of some classes of workloads. Columnar partitioning is expected to have the following performance impacts.

- You should expect a significant I/O performance improvement for queries that access a variable small subset of the columns either in predicates or as projections, and rows of a column-partitioned table.

For example, if 20% of the data in rows is needed to satisfy a query, the I/O for a column-partitioned table should be approximately 20% of the I/O for a table without column partitioning.

I/O might be further reduced depending on the effectiveness of autocompression. Additional I/O might be needed to reconstruct rows if many columns are projected or used in predicates.

- A potentially significant negative performance impact can occur for requests that access more than a small subset of the columns, rows, or both of a column-partitioned table.
- CPU usage might increase for handling autocompression, decompression, and containers. With a reduction in I/O and a possible increase in CPU use, workloads can change from being I/O-bound to being CPU-bound.

The performance of CPU-bound workloads might not improve with column partitioning.

- A potentially significant negative performance impact on INSERT operations for a column-partitioned table or join index, especially for single-row INSERT operations, but less so for bulk insert operations such as array INSERT and INSERT ... SELECT statements.
- A potentially significant negative performance impact for UPDATE operations that select more than a small subset of rows to be updated, as described by the first 2 bullets. Vantage does update operations as a DELETE operation followed by an INSERT operation; therefore, all the columns of rows selected for UPDATE operations must be accessed.

There is a range of from 1 or more orders of magnitude of improved performance to 1 or more orders of magnitude of degraded performance for accessing and updating column-partitioned tables.

The greatest improvement occurs when there is a highly selective predicate on a column in a single-column partition of a table with hundreds or thousands of columns and only a few columns are projected.

The worst case is when a query is not very selective, most or all of the columns in the table are projected, there are thousands of column partitions, only 8 available column partition contexts, and the table is row-partitioned in a way that there are very few rows in a populated partition.

Comparing Nonpartitioned NoPI Tables With Column-Partitioned Tables With and Without Autocompression

This topic compares nonpartitioned NoPI tables and column-partitioned tables with and without autocompression.

The following CREATE TABLE statement constructs the first version of the table that is examined in this topic, *album_artist*.

The first version of test table *album_artist* is created as the nonpartitioned NoPI table *album_artist_ord_nopi*.

```
CREATE TABLE album_artist_ord_nopi (
  artist      CHARACTER(25) NOT NULL,
  album       CHARACTER(50) NOT NULL,
  release_date DATE          NOT NULL,
  sold        INTEGER        NOT NULL,
  producer    CHARACTER(40))
NO PRIMARY INDEX;
```

The populated nonpartitioned NoPI table for *album_artist_ord_nopi* looks like this.

| album_artist_ord_nopi | | | | | | | | |
|-----------------------|----|-------|---------|--------|-------|--------------|-------------|----------|
| part | HB | row # | 1s & 0s | artist | album | release_date | sold | producer |

| | | | | | | | | |
|---|---|---|-----|-----------------------|-------------------|------------|-----------|-------------------|
| 0 | n | 2 | 0,1 | Keith Rowe | Harsh | 06-25-1980 | 1,850,000 | Felix Klopotek |
| 0 | n | 3 | 1,1 | Derek Bailey | Lot '74 | 07-04-2010 | 1,000,000 | Derek Bailey |
| 0 | n | 1 | 1,1 | Albert Ayler | Spiritual Unity | 11-28-1964 | 2,375,000 | Bernard Stollman |
| 0 | n | 5 | 1,1 | Albert Ayler | Bells | 01-28-1965 | 975,000 | Bernard Stollman |
| 0 | n | 6 | 1,1 | Albert Ayler | Spirits Rejoice | 06-25-1965 | 3,100,000 | Bernard Stollman |
| 0 | n | 8 | 1,0 | Pierre Boulez | Pli Selon Pli | 07-04-1973 | 3,250,000 | Paul Myers |
| 0 | n | 7 | 0,1 | Keith Rowe | Duos for Doris | 04-14-2003 | 2,500,000 | Jon Abbey |
| 0 | n | 9 | 1,1 | Karlheinz Stockhausen | Sternklang | 11-28-1976 | 750,000 | Dr. Rudolf Werner |
| 0 | n | 4 | 1,1 | Bix Beiderbecke | Singin' the Blues | 04-14-1990 | 3,125,000 | Tommy Rockwell |

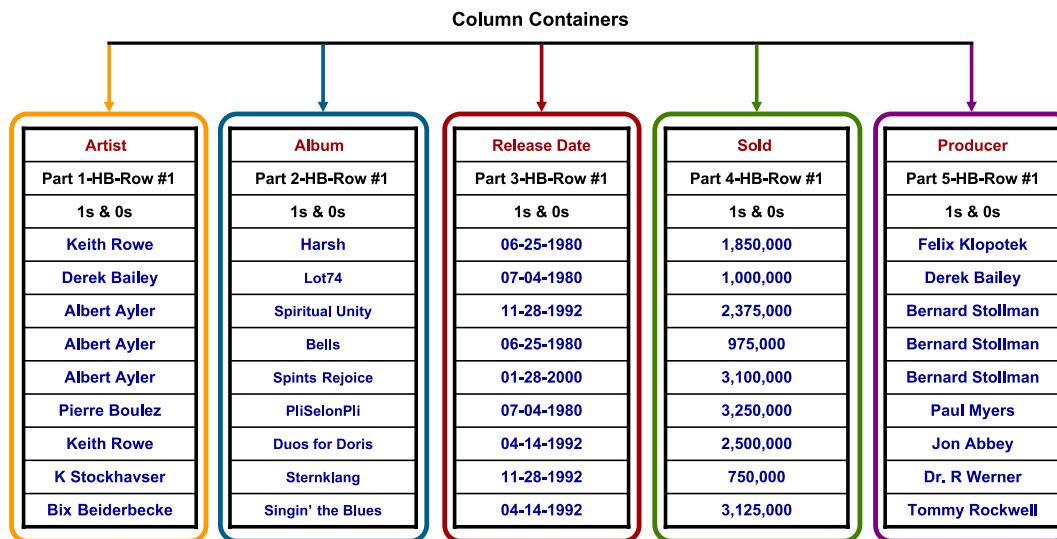
where:

| Row header element ... | Represents the ... |
|------------------------|---------------------------------------|
| part | partition number for the container. |
| HB | hash bucket number for the container. |
| Row # <i>n</i> | row number for the container. |
| 1s & 0s | presence bits for the container. |

The second version of test table *album_artist* is created as column-partitioned table *album_artist_cp_nopi_no_ac* without autocompression.

```
CREATE TABLE album_artist_cp_nopi_no_ac (
  artist      CHARACTER(25) NOT NULL NO AUTO COMPRESS,
  album       CHARACTER(50) NOT NULL NO AUTO COMPRESS,
  release_date DATE NOT NULL          NO AUTO COMPRESS,
  sold        INTEGER      NOT NULL NO AUTO COMPRESS,
  producer    CHARACTER(40)          NO AUTO COMPRESS)
NO PRIMARY INDEX
PARTITION BY COLUMN;
```

The column containers for the populated column-partitioned table for *album_artist_cp_nopi_no_ac* with no autocompression looks like this.

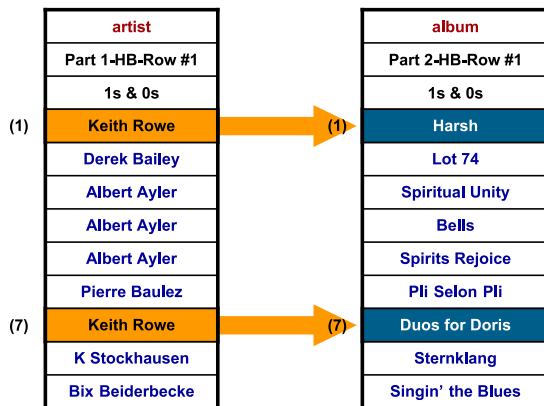


where:

| Column container element ... | Represents the ... |
|------------------------------|---------------------------------------|
| Part <i>n</i> | partition number for the container. |
| HB | hash bucket number for the container. |
| Row # <i>n</i> | row number for the container. |
| 1s & 0s | presence bits for the container. |

You might query this table to ask what the albums by Keith Rowe in the collection are.

```
SELECT album
FROM album_artist_cp_nopi_no_ac
WHERE artist = 'Keith Rowe';
```



The third version of test table *album_artist* is created as the column-partitioned table *album_artist_cp_nopi_ac* with autocompression.

```
CREATE TABLE album_artist_cp_nopi_ac (
  artist      CHARACTER(25) NOT NULL,
  album       CHARACTER(50) NOT NULL,
  release_date DATE          NOT NULL,
  sold        INTEGER        NOT NULL,
  producer    CHARACTER(40))
NO PRIMARY INDEX
PARTITION BY COLUMN;
```

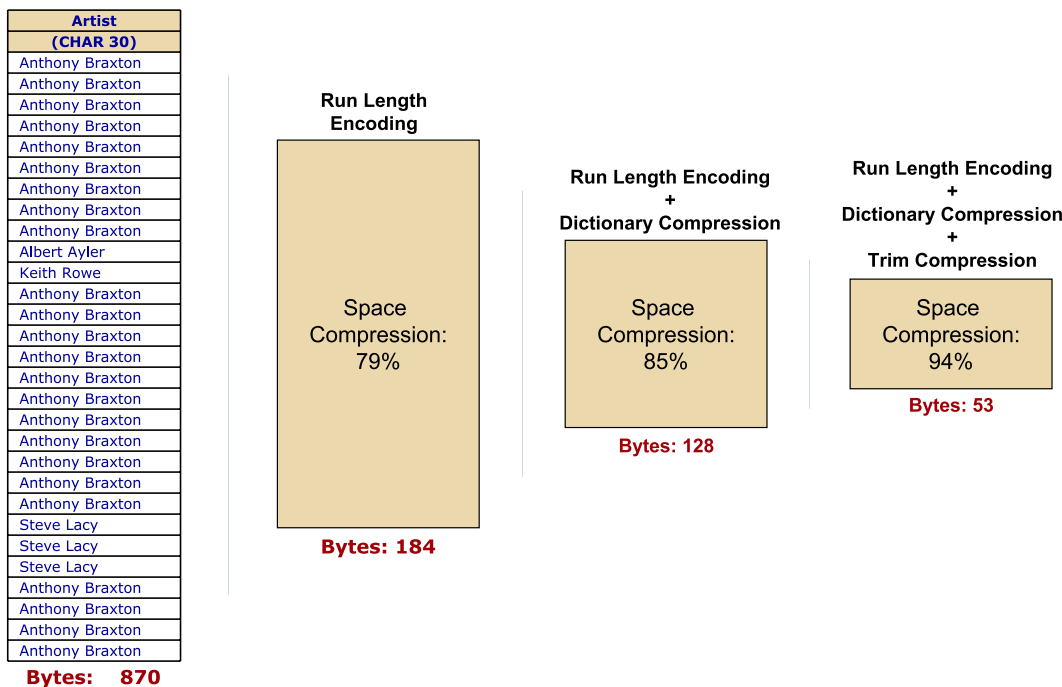
The column containers for the populated column-partitioned table for *album_artist_cp_nopi_ac* with autocompression look like this.

| artist | album | release_date | sold | producer |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| Part 1-HB-Row # 1 | Part 1-HB-Row # 1 | Part 1-HB-Row # 1 | Part 1-HB-Row # 1 | Part 1-HB-Row # 1 |
| 1s and 0s | 1s and 0s | 1s and 0s | 1s and 0s | 1s and 0s |
| Keith Rowe (2) | Harsh | 06-25-1980 | 1,850,000 | Felix Klopotek |
| Derek Bailey | Lot '74 | 07-04-1980 | 1,000,000 | Derek Bailey |
| Albert Ayler (3) | Spiritual Unity | 11-28-1992 | 2,375,000 | Bernard Stollman |
| Pierre Boulez | Bells | 06-25-1980 | 975,000 | Bernard Stollman |
| Karlheinz Stockhausen | Spirits Rejoice | 01-28-2000 | 3,100,000 | Bernard Stollman |
| Bix Beiderbecke | Pli Selon Pli | 07-04-1980 | 3,250,000 | Paul Myers |

| | | | | |
|----------------------------|-----------------------------|-------------------------------|-----------------------|-------------------------|
| | Duos for Doris | 04-14-1992 | 2,500,000 | Jon Abbey |
| | Sternklang | 11-28-1992 | 750,000 | Dr. Rudolf Werner |
| | Singin' the Blues | 04-14-1992 | 3,125,000 | Tommy Rockwell |
| ... | ... | ... | ... | ... |
| Run-Length Encoding | Trim Trailing Spaces | Value List Compression | No compression | Null compression |

The caption at the bottom of each container states the type of autocompression that Vantage chose to apply to it.

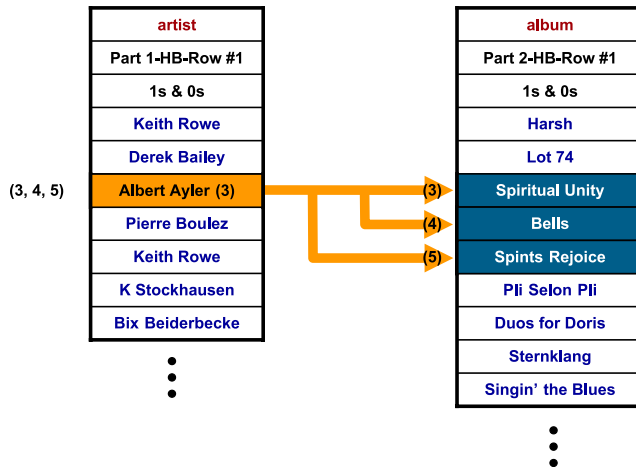
The following graphic indicates the extent of the space savings that can be realized by the autocompression of column-partitioned table containers. Beginning with 29 rows of 30 characters each (a total of 870 bytes) from entries in the *Artist* container that are not displayed elsewhere in this example set, Vantage compresses the container data to a final size of 53 bytes by combining several different autocompression methods, a reduction of 99.4%.



You can query this table container set to ask what the albums by artist Albert Ayler are.

```
SELECT album
FROM album_artist_cp_nopi_ac
WHERE artist = 'Albert Ayler';
```

Because the entries in the artist container are compressed using run-length encoding, there is only 1 entry for artist Albert Ayler, but that entry points to the 3 different albums that are associated with Albert Ayler, as the following graphic indicates.



The fourth version of *album_artist*, *album_artist_mlpcp_nopi_ac*, table uses multilevel partitioning with default autocompression.

```
CREATE TABLE album_artist_mlpcp_nopi_ac (
  artist      CHARACTER(25) NOT NULL,
  album       CHARACTER(50) NOT NULL,
  release_date DATE          NOT NULL,
  sold        INTEGER        NOT NULL,
  producer    CHARACTER(40))
NO PRIMARY INDEX
PARTITION BY (COLUMN,
              RANGE_N(release_date BETWEEN DATE '1980-01-01'
                      AND DATE '2000-01-28'
                      EACH INTERVAL '1' YEAR));
```

The containers for the populated column-partitioned table for *album_artist_mlpcp_nopi_ac* with autocompression look like this.

| artist | album | release_date | sold | producer |
|-------------------|-------------------|-------------------|-------------------|-------------------|
| Part 1-HB-Row # 1 | Part 1-HB-Row # 1 | Part 1-HB-Row # 1 | Part 1-HB-Row # 1 | Part 1-HB-Row # 1 |

| 1s and 0s | 1s and 0s | 1s and 0s | 1s and 0s | 1s and 0s |
|---------------|---------------|------------|-----------|------------------|
| Keith Rowe | Harsh | 06-25-1980 | 1,850,000 | Felix Klopotek |
| Albert Ayler | Bells | 06-25-1980 | 975,000 | Bernard Stollman |
| Derek Bailey | Lot '74 | 07-04-1980 | 1,000,000 | Derek Bailey |
| Pierre Boulez | Pli Selon Pli | 07-04-1980 | 3,250,000 | Paul Myers |

| artist | album | release_date | sold | producer |
|-----------------------|-------------------|--------------------|--------------------|--------------------|
| Part 101-HB-Row #1 | Part 201-HB-Row#1 | Part 301-HB-Row #1 | Part 401-HB-Row #1 | Part 501-HB-Row #1 |
| 1s and 0s | 1s and 0s | 1s and 0s | 1s and 0s | 1s and 0s |
| Keith Rowe | Duos for Doris | 04-14-1992 | 2,500,000 | Jon Abbey |
| Bix Beiderbecke | Singin' the Blue | 04-14-1992 | 3,125,000 | Tommy Rockwell |
| Albert Ayler | Spiritual Unity | 11-28-1992 | 2,375,000 | Bernard Stollman |
| Karlheinz Stockhausen | Sternklang | 11-28-1992 | 750,000 | Dr. Rudolf Werner |

| artist | album | release_date | sold | producer |
|--------------------|-------------------|--------------------|--------------------|--------------------|
| Part 601-HB-Row #1 | Part 201-HB-Row#1 | Part 301-HB-Row #1 | Part 401-HB-Row #1 | Part 501-HB-Row #1 |
| 1s and 0s | 1s and 0s | 1s and 0s | 1s and 0s | 1s and 0s |
| Albert Ayler | Spirits Rejoice | 01-28-2000 | 3,100,000 | Bernard Stollman |

where:

| Column container element ... | Represents the ... |
|------------------------------|---------------------------------------|
| Part <i>n</i> | partition number for the container. |
| HB | hash bucket number for the container. |
| Row # <i>n</i> | row number for the container. |
| 1s & 0s | presence bits for the container. |

You might query this table container set to ask which artists released albums in 1992 and what the titles of those albums were.


```
SELECT artist, album, EXTRACT(YEAR FROM release_date) AS year
FROM album_artist_mlcpc__nopi_ac
WHERE year = 1992;
```

The fifth version of *album_artist*, *album_artist_mccp_nopi_noac*, creates a multicolumn container without autocompression. The multicolumn container is defined on the *release_date*, *sold*, and *producer* columns.

```
CREATE TABLE album_artist_mccp_nopi_noac (
  artist CHARACTER(25) NOT NULL,
  album CHARACTER(50) NOT NULL,
  (release_date DATE NOT NULL,
   sold INTEGER NOT NULL,
   producer CHARACTER(40)))
NO PRIMARY INDEX
PARTITION BY COLUMN NO AUTO COMPRESS;
```

The three containers for this table look like this.

| artist | album | release_date | sold | producer |
|-------------------------|-------------------------|-------------------------|-----------|-------------------|
| Part 1-HB-Row #1 | Part 2-HB-Row #1 | Part 3-HB-Row #1 | | |
| 1s and 0s | 1s and 0s | 1s and 0s | | |
| Keith Rowe | Harsh | 06-25-1980 | 1,850,000 | Felix Klopotek |
| Derek Bailey | Lot '74 | 07-04-1980 | 1,000,000 | Derek Bailey |
| Albert Ayler | Spiritual Unity | 11-28-1992 | 2,375,000 | Bernard Stollman |
| Albert Ayler | Bells | 06-25-1980 | 975,000 | Bernard Stollman |
| Albert Ayler | Spirits Rejoice | 01-28-2000 | 3,100,000 | Bernard Stollman |
| Pierre Boulez | Pli Selon Pli | 07-04-1980 | 3,250,000 | Paul Myers |
| Keith Rowe | Duos for Doris | 04-14-1992 | 2,500,000 | Jon Abbey |
| Karlheinz Stockhausen | Sternklang | 11-28-1992 | 750,000 | Dr. Rudolf Werner |
| Bix Beiderbecke | Singin' the Blues | 04-14-1992 | 3,125,000 | Tommy Rockwell |

The following CREATE TABLE statement creates a sixth version of the *album_artist* table, *album_artist_hybrid_row_col_noac*, with hybrid ROW and COLUMN storage without autocompression. There is an individual COLUMN partition for the *artist* and *album* columns and a grouped ROW partition on the columns *release_date*, *sold*, *producer*, and *lyric*.

```
CREATE TABLE album_artist_hybrid_row_col_noac (
  artist CHARACTER(35) NOT NULL NO AUTOCOMPRESS,
```

```

album          CHARACTER(50) NOT NULL NO AUTOCOMPRESS,
ROW (release_date DATE          NOT NULL NO AUTOCOMPRESS,
     sold         INTEGER        NOT NULL NO AUTOCOMPRESS,
     producer     CHARACTER (40)          NO AUTO COMPRESS,
     lyric        LONG VARCHAR))
NO PRIMARY INDEX
PARTITION BY COLUMN;

```

The two containers and the single multicolumn subrow storage partition look like this.

| artist | album | | | | | | | |
|-----------------------|-------------------|------|----|------|--------------|-----------|-------------------|-------|
| Part 1-HB-Row #1 | Part 2-HB-Row #1 | | | | | | | |
| 1s and 0s | 1s and 0s | Part | HB | Row# | release_date | sold | producer | lyric |
| Keith Rowe | Harsh | 0 | n | 2 | 06-25-1980 | 1,850,000 | Felix Klopotek | null |
| Derek Bailey | Lot '74 | 0 | n | 3 | 07-04-1980 | 1,000,000 | Derek Bailey | null |
| Albert Ayler | Spiritual Unity | 0 | n | 1 | 11-28-1992 | 2,375,000 | Bernard Stollman | null |
| Albert Ayler | Bells | 0 | n | 5 | 06-25-1980 | 975,000 | Bernard Stollman | null |
| Albert Ayler | Spirits Rejoice | 0 | n | 6 | 01-28-2000 | 3,100,000 | Bernard Stollman | null |
| Pierre Boulez | Pli Selon Pli | 0 | n | 8 | 07-04-1980 | 3,250,000 | Paul Myers | null |
| Keith Rowe | Duos for Doris | 0 | n | 7 | 04-14-1992 | 2,500,000 | Jon Abbey | null |
| Karlheinz Stockhausen | Sternklang | 0 | n | 9 | 11-28-1992 | 750,000 | Dr. Rudolf Werner | null |
| Bix Beiderbecke | Singin' the Blues | 0 | n | 4 | 04-14-1992 | 3,125,000 | Tommy Rockwell | null |

See *Teradata Vantage™ - Database Design*, B035-1094 for some example comparisons of the number of I/Os required to read the same data from primary-indexed tables, partitioned tables, nonpartitioned NoPI tables, and column-partitioned tables that are partitioned in different ways and for a number of other performance issues related to column-partitioned tables and join indexes.

Partitioned and Nonpartitioned Primary Indexes

Primary indexes for global temporary, volatile, and standard base tables can be partitioned or nonpartitioned. Nonpartitioned NoPI tables, with the exception of column-partitioned tables, *cannot* be partitioned because they have no primary index to partition. See [Column-Partitioned Tables](#).

An nonpartitioned primary index is the traditional primary index by which rows are assigned to AMPs. Apart from maintaining their storage in row hash order, no additional assignment processing of rows is performed once they are hashed to an AMP.

A partitioned primary index (PPI) permits rows to be assigned to user-defined data partitions on the AMPs, enabling enhanced performance for range queries that are predicated on primary index values. For details, see *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

You can define partitioning on a single level or on multiple levels. A PPI defined on multiple levels is referred to as a multilevel PPI, or MLPPI.

Depending on the number of combined partitions for a table, its partition numbers consume either 2 bytes or 8 bytes in the row header, as the following table specifies.

| IF a table has this many combined partitions ... | Its partition number consumes this many bytes in the row header ... |
|--|---|
| ≤ 65,535 | 2 |
| > 65,535 | 8 |

Single-level column partitioning with no ADD option also consumes 2 bytes in the row header.

Partition assignment is based on how the partitioning expression is defined. The partitioning expressions for multilevel partitioning must be defined using only CASE_N or RANGE_N expressions in any combination. The functions CASE_N and RANGE_N are designed specifically to support simple partitioning expressions. See *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145. But, you can write any valid SQL expression as a partitioning expression for single-level partitioning, with the following exclusions:

- Comparison of CHARACTER or CHARACTER SET GRAPHIC data involving columns or expressions using the Kanji1 or KanjiSJIS server character sets.
- Any kind of user-defined function
- Aggregate functions
- Ordered analytical functions
- Built-in functions
- The RANDOM function
- The HASHAMP and HASHBAKAMP functions
HASHROW and HASHBUCKET *are* permitted.
- The system-derived PARTITION and PARTITION#L *n* columns

- Set operators
- Subqueries
- Columns having a BLOB, CLOB, or Geospatial data type.

You can base a partitioning expression for single-level partitioning on any of the following general forms.

- Direct partitioning on a numeric column
- Expressions based on 1 or more columns
- Expressions based on the CASE_N function
- Expressions based on the RANGE_N function

See *Teradata Vantage™ - Database Design*, B035-1094 for details on the various usage considerations for each of these partitioning strategies.

Partitioning expressions for multilevel partitioning can be based only on the following general forms in any combination.

- Expressions based on the CASE_N function (see *Teradata Vantage™ - Database Design*, B035-1094 for details).
- Expressions based on the RANGE_N function (see *Teradata Vantage™ - Database Design*, B035-1094 for details).

The following table presents the intended use of the CASE_N and RANGE_N functions for partitioning expressions.

| Use this function ... | To define a mapping ... |
|-----------------------|---|
| CASE_N | between conditions to INTEGER numbers. |
| RANGE_N | of ranges of INTEGER or DATE values to INTEGER numbers. |

The partitioning expressions you can define for partitioning a table have certain restrictions regarding the data types you can specify within them and with respect to the data type of the result of the function.

The following table summarizes these restrictions.

| Data Type | PARTITION BY | | |
|-----------------|--------------|--------|------------|
| | RANGE_N | CASE_N | Expression |
| ARRAY VARRAY | N | N | N |
| BIGINT | Y | X | I |
| BLOB | N | N | N |
| BYTE | X | X | X |
| BYTEINT | Y | X | I |
| CHARACTER | Y | X | I |

| Data Type | PARTITION BY | | |
|---|--------------|--------|------------|
| | RANGE_N | CASE_N | Expression |
| CLOB | N | N | N |
| DATE | Y | X | I |
| DECIMAL NUMERIC | X | X | I |
| NUMBER (exact form) | X | X | I |
| DOUBLE PRECISION FLOAT REAL | X | X | I |
| NUMBER (approximate form) | X | X | I |
| GRAPHIC | N | X | N |
| INTEGER | Y | X | Y |
| INTERVAL YEAR | X | X | I |
| INTERVAL YEAR TO MONTH | X | X | X |
| INTERVAL MONTH | X | X | I |
| INTERVAL DAY | X | X | I |
| INTERVAL DAY TO HOUR | X | X | X |
| INTERVAL DAY TO SECOND | X | X | X |
| INTERVAL SECOND | X | X | X |
| LONG VARCHAR | Y | X | I |
| LONG VARCHAR CHARACTER SET GRAPHIC | N | N | N |
| PERIOD The BEGIN and END bound functions are valid in a partitioning expression when they are defined on a valid PERIOD column and the result can be cast implicitly to a numeric data type. | N | X | N |
| SMALLINT | Y | X | I |
| TIME | X | X | X |
| TIME WITH TIME ZONE | X | X | X |
| TIMESTAMP | Y | X | X |
| TIMESTAMP WITH TIME ZONE | Y | X | X |
| UDT | N | N | N |

| Data Type | PARTITION BY | | |
|------------|--------------|--------|------------|
| | RANGE_N | CASE_N | Expression |
| VARBYTE | X | X | X |
| VARCHAR | Y | X | I |
| VARGRAPHIC | N | N | N |
| JSON | N | N | N |
| XML | N | N | N |

The following table explains the abbreviations used in the previous table.

| Key | |
|--------|--|
| Symbol | Definition |
| I | Valid for a partitioning expression. If the type is also the data type of the result, then it must be such that it can be cast to a valid INTEGER value. |
| N | Not valid for a partitioning expression. If the partitioning expression is defined using a CASE_N function, then this type is <i>not</i> valid for the CASE_N condition. |
| X | Valid for a partitioning expression, but cannot be the data type of the result of the expression. If the partitioning expression is defined using a CASE_N function, then this type <i>is</i> valid for the CASE_N condition. |
| Y | Valid for a partitioning expression and valid as the data type of the result of the partitioning expression. |

ADD Option

The ADD option reserves additional partition numbers for a partitioning level to enable adding partitions to a partitioning level at a later time using an ALTER TABLE statement. The following rules apply to the ADD clause for a row or column partition.

- If you specify an ADD clause for a partitioning level, the maximum number of partitions for that partitioning level is the number of defined partitions for the level plus the value of the constant specified in the ADD clause.

The maximum is 9,223,372,036,854,775,807.

- The maximum number of partitions for a row partitioning level is *at least* 2.

This error occurs when you define only 1 partition for a row partitioning level that also specifies ADD 0 or that does not specify an ADD option and the maximum is not increased to at least 2 in one of the following ways.

- For single-level partitioning, the maximum combined partition number is the same as the maximum partition number for this partitioning level. That value cannot exceed 9,223,372,036, 854,775,807.
- For 2-byte partitioning, the maximum number of partitions for the first level is increased to the largest value that does not cause the maximum combined partition number to exceed 65,535.
- For 8-byte partitioning, the maximum number of partitions for the first level is increased to the largest value that does not cause the maximum combined partition number to exceed 9,223,372,036,854, 775,807.
- For each row partitioning level in level order without an ADD clause where the level has 2-byte partitioning, the maximum number of partitions for the level is the largest value that does not cause the partitioning to be 8-byte partitioning.

Otherwise, the maximum number of partitions for the level is the largest value that does not cause the maximum number of combined partitions to exceed 9,223,372,036,854,775,807.

- You can specify ADD 0 for a partitioning level to specify that the maximum number of partitions for the level is the same as the number of defined partitions.

This is useful for a row partitioning level if you want a lower level that does not specify the ADD clause to receive any excess partitions.

This is useful for a column partitioning level if you want to override the default of ADD 10 so that other levels can have more partitions.

- If you do not specify an ADD clause for a column partitioning level and that level is the only partitioning level, the maximum number of partitions including the 2 for internal use is 65,534.
- If the following things are all true, the maximum number of partitions for the column partitioning level is the number of column partitions defined plus 10.
 - You do not specify an ADD clause for a column partitioning level
 - The table or join index also has row partitioning
 - At least one of the row partitioning levels does not specify an ADD clause

In this case, the default ADD clause for the column partitioning level is ADD 10.

- If the following things are all true, the maximum number of partitions for the column partitioning level is the largest value that does not cause the partitioning to be 8-byte partitioning.
 - You do not specify an ADD clause for a column partitioning level
 - The table or join index also has row partitioning
 - All of the row partitioning levels specify an ADD clause
 - Using the number of column partitions defined plus 10 as the maximum number of column partitions, the partitioning would be 2-byte partitioning

Otherwise, the maximum number of partitions for the column partitioning level is 9,223,372,036, 854,775,807.

Partitioning Expressions Using on a CASE_N or RANGE_N Character Column

You can use CASE_N functions to define a character partitioning expression that groups the rows of a table into partitions that are based on a particular condition. You can then evaluate the conditions to return the number of the first condition that evaluates to TRUE without encountering a condition that evaluates to UNKNOWN. You can also use various options to handle cases where a condition evaluates to UNKNOWN or when all conditions evaluate to FALSE.

You can use the following comparison operators and string functions within a CASE_N function to evaluate partitioning expression character strings using the current session collation if the expression is not part of a partitioning expression.

- =
- <
- >
- >=
- =<
- <>
- BETWEEN
- LIKE

If one of the partitioning expressions in a column partitioning uses a CASE_N function, then all of the comparisons use the session collation that was in effect when the table or join index was created or had its partitioning altered.

The rules for CASE_N expressions in a column partitioning created with a CREATE TABLE statement are the same as those for a non-character partitioning expression with the following exceptions.

- For single-level partitioning, if a partitioning expression consists only of a RANGE_N function with INTEGER data type, the total number of partitions defined (including all the range partitions, and the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN partitions if specified) must be less than or equal to 2,147,483,647.
- If you copy the primary index definition from a table to create another table using a CREATE TABLE ... AS ... WITH [NO] DATA statement, and the primary index of the source table is partitioned, the copied table definition is partitioned in the same way.

The database uses the collation used for the character partitioning of the source table for the definition of the target table.

The session collation does not need to be the same as the collation of the character partitioning for the source table.

If the collation for the source table is MULTINATIONAL or CHARSET_COLL, and the definition of the collation or collation character set has changed since the table was created, then using the CREATE TABLE ... AS statement returns an error to the requestor.

You must revalidate the table before it can be copied using the CREATE TABLE ... AS statement.

The rules for CREATE TABLE ... AS statements are otherwise the same as those for tables that do not have character partitioning.

- You can define a partitioning expression using a character or graphic comparison, but the partitioning expression cannot be defined on Kanji1 or KanjiSJIS columns or constant expressions.

A partitioning expression can specify the UPPERCASE qualifier and any of the following character functions.

- CHAR2HEXINT
- INDEX
- LOWER
- MINDEX
- POSITION
- TRANSLATE
- TRANSLATE_CHK
- TRIM
- UPPER
- VARGRAPHIC

Expressions and referenced columns in the partitioning expression must not have BLOB, CLOB, or Geospatial data types, nor can they be defined using any of the following functions and function types.

- External UDFs
- SQL UDFs
- Built-in functions, including the following:
 - Aggregate functions
 - Grouped-row OLAP functions
 - HASHAMP function
 - HASHBAKAMP function
 - RANDOM function
- ROWID keyword
- PARTITION or PARTITION[#Ln] columns
- Set operators
- Subqueries

A partitioning expression must be based on 1 or more columns in the table. Specifically, a partitioning expression must be a deterministic expression that is based on the columns within a row.

- The evaluation of the partitioning expression at any character partitioning level uses the session collation in effect when the table was created to determine the ordering of character data for comparison operators.

- The evaluation of a partitioning expression that is not based on a RANGE_N function at any character partitioning level uses the same case sensitivity rules that were in effect when the table was created to determine ordering of character data for comparison operators and string functions.

| IF a table is created in this session mode ... | THEN all comparisons with constant literals treat the literal as ... |
|--|--|
| ANSI | CASESPECIFIC unless the constant literals are explicitly cast to be NOT CASESPECIFIC. When an expression in a comparison is CASESPECIFIC, then the comparison is case sensitive; otherwise, it is case blind. |
| Teradata | NOT CASESPECIFIC by default. A comparison between a constant and a NOT CASESPECIFIC column or expression is treated as case blind unless the constant literals are explicitly cast to be CASESPECIFIC. |

See *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for more information on the rules for case sensitivity in character data comparisons.

- A partitioning expression at a given partitioning level should have the same case sensitivity in comparisons and string functions as what is commonly used in WHERE or ON clause conditions of DML requests involving the partitioning column. Otherwise, the character partitioning might not be eligible for optimizations that eliminate partitions, either statically or dynamically. For information about row partition elimination, see *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142. For a row partition to be statically eliminated, the conditional expression associated with that partition must be a combination of predicates that all have the same case sensitivity.
- The evaluation of a RANGE_N partitioning expression at any character partitioning level is independent of the default case sensitivity rules in effect for the session when the table was created, but only when the test value is an expression that does not contain any string literals without an explicit CAST.
- For a RANGE_N partitioning expression,

| IF a test value is ... | THEN its range boundaries are ... |
|------------------------|-----------------------------------|
| NOT CASESPECIFIC | NOT CASESPECIFIC |
| CASESPECIFIC | CASESPECIFIC |

- If the test value is an expression that contains NOT CASESPECIFIC columns or expressions and string literals with no explicit cast, then the test value behaves according to the following table.

| IF the partitioning is created in this session mode ... | THEN the test value ... |
|---|-------------------------|
| ANSI | is CASESPECIFIC |

| IF the partitioning is created in this session mode ... | THEN the test value ... |
|---|--|
| Teradata | for a RANGE_N expression is NOT CASESPECIFIC |

- A test value for a RANGE_N function must result in a BYTEINT, BIGINT, INTEGER, SMALLINT, DATE, TIMESTAMP(*n*), TIMESTAMP(*n*) WITH TIME ZONE, CHARACTER, or VARCHAR data type.
- The sum of the sizes of all constant literals referenced in all partitioning expressions in a partitioned primary index definition must be less than 64 KB. Otherwise, the CREATE TABLE statement returns an error to the requestor.
- If you use multiple character sets on your system, you should always create character partitioning columns using the Unicode server character set and store the partitioning column using the same server character set. Otherwise, a DML request against this table that can use a rowkey-based merge join might be restricted to direct rowkey-based merge joins between 2 partitioned tables that have identical partitioning.

Another reason to create your character partitioning columns using the Unicode server character set is to fully optimize dynamic row partition elimination. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

- To maximize the cases that can use character partitioning for optimization, you should create your partitioned tables in a Teradata mode session if the character partitioning expression set involves columns that are NOT CASESPECIFIC. All constant and non-constant character expressions in a comparison should be defined as NOT CASESPECIFIC.

If the partitioning columns are CASESPECIFIC, then the partitioning should either be created in an ANSI mode session, or you should define any non-constant character expressions that are specified in a comparison to be CASESPECIFIC.

The easiest way to guarantee that all comparisons are CASESPECIFIC is to make those comparisons between the CASESPECIFIC partitioning column (instead of an expression on the partitioning column) and a constant expression.

While all comparisons should be made using the same case sensitivity to enable static row partition elimination, the recommend best practice is to specify all partitioning to be case blind, which enables row partition elimination to occur when you specify either case blind or case sensitive WHERE clause conditions are present.

- The database might convert any string literals in partitioning expression comparison operations from the session client character set or explicitly specified character set, if a qualifier is present, of the CREATE TABLE statement, to the character set of the non-constant expression in the comparison. The database always performs this conversion for partitioning expressions based on the RANGE_N function.

For example, consider CASE_N (*t1.a* < 'cat'), where *t1.a* is the non-constant expression and 'cat' is the string literal. The characters specified in the string literal should exist in the client character set to ensure the proper translation of the non-constant expression to the server character set. You

should submit a SHOW TABLE statement and examine the SQL create text it returns to ensure that the system did not make any undesired translations.

This example shows the use of native Japanese ideographs in a character partitioning expression.

```
CREATE SET TABLE df2.t1c, Fallback, NO BEFORE JOURNAL,
      NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  c1 INTEGER,
  c2 CHARACTER(30) CHARACTER SET UNICODE NOT CASESPECIFIC,
  c3 CHARACTER(4) CHARACTER SET GRAPHIC CASESPECIFIC)
PRIMARY INDEX (c1)
PARTITION BY CASE_N(c2 BETWEEN '艶 '
      AND      '静 ',
  c2 = '偉 ',
  c2 BETWEEN '韓 '
      AND      '髻 ',
  NO CASE, UNKNOWN);
```

To create the same table using BTEQ or another client API that does not support client character sets such as KanjiEUC_OU natively, you must create the table using hexadecimal Unicode literals in place of the Japanese ideographs, like this.

```
CREATE SET TABLE df2.t1c, Fallback, NO BEFORE JOURNAL,
      NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  c1 INTEGER,
  c2 CHARACTER(30) CHARACTER SET UNICODE NOT CASESPECIFIC,
  c3 CHARACTER(4) CHARACTER SET GRAPHIC CASESPECIFIC)
PRIMARY INDEX (c1)
PARTITION BY CASE_N(c2 BETWEEN _UNICODE '9758'XC
      AND      _UNICODE '9759'XC,
  c2 =      _UNICODE '9797'XC,
  c2 BETWEEN _UNICODE '97D3'XC
      AND      _UNICODE '9AFB'XC,
  NO CASE, UNKNOWN);
```

It is your responsibility to ensure that the client character set is correct for the string literals you specify, and that the client character set is compatible with the server character set of the test value for RANGE_N expressions or non-constant expressions that are compared in CASE_N partitioning. The database does not do this for you.

Partitioning Expressions Using DATE or TIMESTAMP Built-In Functions

Vantage supports the BEGIN and END bound functions on Period columns in partitioning expressions anywhere that a DateTime expression is valid.

For information about how to optimize partitioning expressions based on updatable dates and updatable timestamps for reconciliation using ALTER TABLE TO CURRENT statements, see [Partitioning Expression Based on Updatable Current Date and Timestamp Expressions](#).

Follow these guidelines for creating the partitioning expression for tables that are to be queried using DATE conditions.

- Use RANGE_N with a DATE partitioning column. For example,

```
PARTITION BY RANGE_N(date_column BETWEEN DATE '...'
                        AND      DATE '...'
                        EACH INTERVAL 's' t)
```

- Use INTERVAL constants in the EACH clause, where *t* is DAY, MONTH, YEAR, or YEAR TO MONTH
- Use DATE constants for the ranges, for example DATE '2003-08-06'.

Do *not* use INTEGER or CHARACTER constants for dates.

DATE constants are a better choice for the following reasons.

- They are easier to read, and it is clear that the ranges are over dates.
- They provide less dependence on the FORMAT.
- It may seem intuitive to partition by a DATE column such as PARTITION BY salesdate.

This does *not* return a syntax error; in fact, it works correctly for a few very old dates and follows the rule of implicit conversion to get an INTEGER partition number.

The problem is that a table partitioned this way is not useful.

Instead, use RANGE_N and EACH INTERVAL '1' DAY.

- It might seem intuitive to specify something like PARTITION BY 7 to indicate that a date column in the primary index is to be partitioned by week, but this syntax returns a syntax error.

Instead, use RANGE_N and EACH INTERVAL '7' DAY

- Consider having only as many date ranges as you currently need plus a few extra for future dates. In doing so, be certain to balance the following concerns.
 - Limiting ranges to only those that are currently needed helps the Optimizer to cost plans better. It also allows for more efficient primary index access and joins if the partitioning column is not included in the primary index definition.
 - Define future ranges in such a way as to minimize the frequency of adding and dropping partitions using the ALTER TABLE statement (see [ALTER TABLE \(Basic Table Parameters\)](#)).

However, if you perform your partition adding and dropping task too infrequently, you might forget to do them when they are needed to provide accurate query support.

Partitioning Expression Based on Updatable Current Date and Timestamp Expressions

When the logic of a significant number of your DML requests against a table typically specify DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions, you should consider taking advantage of the functionality provided by updatable current dates and updatable current timestamps to partition the rows of the tables you most frequently access with such queries.

A partitioned primary index enables Vantage to partition the rows of a table or noncompressed join index in such a way that row subsets can be accessed efficiently without resorting to full-table scans. If the partitioning expression is defined using an updatable current date or updatable current timestamp, the partition that contains the most recent rows can be defined to be as narrow as possible to optimize efficient access to those rows. An additional benefit of an updatable current date or updatable current timestamp for a partitioning is that the partitioning expression can be designed in such a way that it might not need to be changed as a function of time.

To do this, you can specify the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions in the partitioning expression of a table or noncompressed join index and then periodically update the resolution of their values. This enables rows to be repartitioned on the newly resolved values of the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions at any time you determine that they require reconciliation. You can update the resolution of your partitioning scheme by submitting appropriate ALTER TABLE TO CURRENT statements.

To realize the optimal benefit from partitioning a table or join index using updatable current date or updatable current timestamp values, you must periodically reconcile their partitioning to a more recent date or timestamp value using ALTER TABLE TO CURRENT statements. See [ALTER TABLE TO CURRENT](#).

Although you can specify the DATE, CURRENT_DATE, and CURRENT_TIMESTAMP functions anywhere in a partitioning expression where a date or timestamp constant is valid, you should take special care to optimize the definitions of your partitions in such a way that they can be reconciled optimally using ALTER TABLE TO CURRENT statements.

Use care in designing your partitioning expressions to minimize the expense of reconciling the rows partitioned on newly resolved DATE, CURRENT_DATE, CURRENT_TIMESTAMP, or CURRENT_DATE constants using ALTER TABLE TO CURRENT statements. Reconciling the row can be expensive in terms of the time required to scan the table to locate the rows that need to be reconciled and the time required to move or delete rows.

If more than a small percentage of the rows in a table or noncompressed join index must be reconciled without being able to take advantage of the optimizations that are available to you, some other form of partitioning might be more advantageous for your applications. An example of such an optimization is full partition deletion of rows.

The various optimizations that are available to you, as well as some of the more obvious impediments to these optimizations that you can avoid, are documented in the following topics.

- [Optimal Reconciliation of CASE_N Partitioning Expressions Based on Updatable Current Date and Timestamp](#)
- [Optimal Reconciliation of RANGE_N PPI Expressions Based on Updatable Current Date and Timestamp](#)
- [Optimal Reconciliation of CASE Partitioning Expressions Based On Updatable Current Date and Timestamp](#)
- [Optimal Reconciliation of Partitioning Expressions Based On Updatable Current Date and Timestamp](#)

The following set of rules applies to using the DATE, CURRENT_DATE, and CURRENT_TIMESTAMP functions to define a partitioning expression for a partitioned primary index.

- You can specify the DATE and CURRENT_DATE functions anywhere in a partitioning expression that a date is valid.

You can specify a CURRENT_TIMESTAMP function anywhere in a partitioning expression that a timestamp is valid.

However, note that the result of a test value must not have the TIMESTAMP data type in a RANGE_N function.

- The report produced by a SHOW TABLE or SHOW JOIN INDEX statement returns the user-specified partitioning expression containing the user-specified *expression* with DATE, CURRENT_DATE, or CURRENT_TIMESTAMP and *not* the resolved date or timestamp value.

You can retrieve the last resolved DATE or CURRENT_DATE value, CURRENT_TIMESTAMP value, or both using one of the following system views.

- ResolvedDTSV
- ResolvedDTSVX

For more information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

- The report produced by a SHOW request (see [SHOW request](#)) returns the user-specified partitioning expression.

The report produced by a SHOW QUALIFIED request returns the last resolved date and timestamp for the DATE, CURRENT_DATE, and CURRENT_TIMESTAMP specified in the partitioning expression.

- The following statements are true for partitioning expressions that specify a RANGE_N function with a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function.

| If a partitioning expression is ... | THEN you cannot use an ALTER TABLE statement to ... |
|-------------------------------------|---|
| single-level | ADD or DROP ranges to the partitioning expression. |
| multilevel | ADD or DROP ranges to the level of the partitioning expression. |

Instead, you must use an ALTER TABLE TO CURRENT statement to ADD or DROP such ranges from a table. See [ALTER TABLE TO CURRENT](#).

- If you create a new table using the CREATE TABLE ... AS statement, the following rules apply.

- If the source table has a partitioning defined using the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions, and the new table is a copy of the source table and its index definitions, then Vantage copies the newly created table with the resolved date and timestamp from the source table.
- In all other cases, Vantage resolves any DATE, CURRENT_DATE, or CURRENT_TIMESTAMP values in the partitioning definition of the target table to the creation date, creation timestamp, or both, for the new table.

Optimal Reconciliation of CASE_N Partitioning Expressions Based on Updatable Current Date and Timestamp

The following rules and guidelines apply to specifying CASE_N partitioning expressions in a way that they can be reconciled optimally using ALTER TABLE TO CURRENT statements. See [ALTER TABLE TO CURRENT](#).

- You should specify DATE, CURRENT_DATE, or CURRENT_TIMESTAMP on one side of a CASE_N conditional expression only. The CASE_N specification can be optimal on either the LHS or the RHS of the expression, but you if you specify it on both side, the reconciliation cannot be optimized.

This optimization makes it possible for Vantage to skip some partitions during the reconciliation; however, if you specify a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function on both sides of a conditional expression, Vantage must scan all of the rows in the table to locate any rows that must be reconciled.

- Vantage does not scan a partition during reconciliation if it can determine that its rows will remain in their current partition after any DATE, CURRENT_DATE, or CURRENT_TIMESTAMP values have been reconciled.
- If you specify NO CASE in a CASE_N expression and more than 1 conditional expression specifies the DATE or CURRENT_DATE function, Vantage scans the NO CASE partition during reconciliation only when the prior conditional expressions specified with CURRENT_DATE or CURRENT_TIMESTAMP are not contiguous.

If you specify only 1 conditional expression using a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function, Vantage scans the NO CASE partition only when the partition corresponding to this conditional expression is skipped.

For example, the following expressions are all able to take advantage of this optimization.

- CASE_N (j ≥ CURRENT_DATE, NO CASE)

For this case, Vantage does not scan the rows in the second partition during reconciliation.

- CASE_N(j ≥ CURRENT_DATE /* Current quarter */,
j ≥ CURRENT_DATE - INTERVAL '3' MONTH
AND j < CURRENT_DATE /* previous quarter */,
NO CASE /* old quarters */)

For this case, Vantage does not scan the rows in the last partition, annotated as “old quarters,” during reconciliation because the expressions are all contiguous.

- If you specify NO CASE in the CASE_N expression and more than 1 conditional expression specifies CURRENT_DATE, Vantage scans the NO CASE partition during reconciliation.

If you specify only 1 conditional expression using a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function, Vantage scans the NO CASE partition during reconciliation only when the prior conditional expressions specified with DATE, CURRENT_DATE or CURRENT_TIMESTAMP are not contiguous.

For example, expressions such as the following provide such optimizations.

```
CASE_N(j >= CURRENT_DATE /* current quarter */
      j >= CURRENT_DATE = INTERVAL '3' MONTH
      AND j < CURRENT_DATE /* previous quarter */, NO CASE)
```

For this case, Vantage does not scan the rows in the second partition, labeled as “previous quarter,” during reconciliation.

```
CASE_N(j >= CURRENT_DATE /*current quarter*/,
      j >= CURRENT_DATE - INTERVAL '3' MONTH
      AND j < CURRENT_DATE /*previous quarter*/,
      NO CASE /*old quarters*/)
```

For this case, Vantage scans the NO CASE partition. Because there is a gap of 3 months between the second and third expressions, this is a non-contiguous case.

```
CASE_N(j >= CURRENT_DATE /*current quarter*/,
      j >= CURRENT_DATE - INTERVAL '3' MONTH
      AND j < CURRENT_DATE /*previous quarter*/,
      j < CURRENT_DATE-INTERVAL '3' MONTH /*old quarter*/,
      NO CASE)
```

For this case, Vantage scans partitions 1 and 2, but skips partitions 3 and 4 during reconciliation. The NO CASE partition is skipped because expressions 1, 2, and 3 are all contiguous.

- If you specify an UNKNOWN partition in a CASE_N expression, Vantage always scans it during reconciliation.

This also occurs if you specify a NO CASE OR UNKNOWN partition.

- Vantage does not scan any partitions defined with conditional expressions that do not specify either a DATE, CURRENT_DATE, or a CURRENT_TIMESTAMP function unless they are defined prior to any conditional expressions that specify a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function. Otherwise, they are scanned.

You should specify all conditional expressions that do not specify a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function before you define any conditional expressions that do specify a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function.

For example, the following expressions are all able to take advantage of this optimization.

- CASE_N(j ≥CURRENT_DATE /* current quarter */,
j ≥CURRENT_DATE - INTERVAL '3' MONTH AND j < CURRENT_DATE
/* previous quarter */, NO CASE)
- CASE_N(j ≥CURRENT_DATE /* current quarter */,
j ≥CURRENT_DATE-INTERVAL '3' MONTH AND j < CURRENT_DATE
/* previous quarter */,
j <CURRENT_DATE-INTERVAL '6' MONTH, /* old quarters */ NO CASE)

In this non-contiguous case the system scans NO CASE, and there is a gap of 3 months between 2nd and 3rd expression.

Optimal Reconciliation of RANGE_N PPI Expressions Based on Updatable Current Date and Timestamp

The following rules and guidelines apply to specifying RANGE_N partitioning expressions in a way that they can be reconciled optimally using ALTER TABLE TO CURRENT statements. See [ALTER TABLE TO CURRENT](#).

- A test expression in the RANGE_N function specifies the value used to determine the position in the specified range list.

If the test expression specifies either a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function, then Vantage scans all the rows during reconciliation. This is because the value of the expression changes as the value for DATE, CURRENT_DATE, or CURRENT_TIMESTAMP changes, so each row becomes a candidate to be moved to a different partition.

The test value *cannot* have a result data type of TIMESTAMP. Because of this, if you specify a CURRENT_TIMESTAMP function in the test value, you must employ it in a way that causes the result of the test value to be a value having the BYTEINT, SMALLINT, BYTEINT, or DATE data type.

- Assume that you specify a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function in the starting expression of the first range and then submit an ALTER TABLE TO CURRENT statement on the table.

If the starting expression of the first range that is resolved with a new DATE, CURRENT_DATE, or CURRENT_TIMESTAMP value falls on a partition boundary, then Vantage drops all the partitions preceding the matched partition.

This is equivalent to performing an ALTER TABLE statement that specifies a DROP RANGE clause.

- If you specify either a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function in an ending expression, then the following things are true about a NO RANGE partition.

| IF NO RANGE is ... | THEN Vantage ... |
|--------------------|---|
| specified | scans rows from the NO RANGE partition to partition them based on the newly resolved date. |
| not specified | does not scan any rows. The partitioning expression is modified to accommodate the changed range values. |

This is equivalent to performing an ALTER TABLE statement that specifies an ADD RANGE clause.

- If you specify a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function in both the starting expression and the ending expression, then both of the 2 preceding bullets apply.

Because Vantage must scan rows in the NO RANGE partition when you specify a NO RANGE partition, you should specify either a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function in the partitioning expression *without* defining a NO RANGE partition if possible.

- If a partitioning expression is defined with multiple ranges and you specify either a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function in the starting expression of the first range, the ending expression of the last range, or both, then an ALTER TABLE TO CURRENT reconciliation is optimized.

If you specify a DATE or CURRENT_DATE function in ranges other than the starting expression of the first range or the ending expression of the last range, then Vantage scans all rows during reconciliation and repartitions them as is necessary.

- If you specify either a DATE, CURRENT_DATE, or CURRENT_TIMESTAMP function in an ending expression, then the following things are true.
 - Specifying the DATE or CURRENT_DATE functions in RANGE_N conditions generally simplifies the ALTER TABLE logic, making it possible for you to submit ALTER TABLE TO CURRENT statements periodically instead of submitting ALTER TABLE statements to ADD and DROP ranges.
 - If you specify RANGE_N functions at levels greater than the first, you should specify DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions for both the starting expression of the first range and the ending Period expression of the last range.

You can further optimize reconciliations by dropping and adding the same number of partitions; otherwise, Vantage must scan all of the rows in the table.

- Specifying a DATE or CURRENT_DATE function in RANGE_N partitioning expressions often simplifies the ALTER TABLE logic for reconciling the values.

Because of this, you can periodically submit an ALTER TABLE TO CURRENT statement instead of specifying an ALTER TABLE statement to ADD or DROP ranges.

- If you specify RANGE_N functions in partitioning levels about the first level, both the starting expression of the first range and the ending expression of the last range should specify either a DATE, a CURRENT_DATE, or a CURRENT_TIMESTAMP function.

Reconciliation operations should drop and add the same number of partitions; otherwise, Vantage must scan all of the rows in the specified table or join index.

Optimal Reconciliation of CASE Partitioning Expressions Based On Updatable Current Date and Timestamp

The following case applies to optimized uses of CASE partitioning expressions in a partitioning expression.

If you specify either a DATE, CURRENT_DATE, or a CURRENT_TIMESTAMP function in a partitioning expression CASE expression, then Vantage must scan and reconciles all of the rows in the specified table or join index when you update their values using an ALTER TABLE TO CURRENT statement.

Because of this, the best practice is not to specify DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions in the CASE expressions you specify in a partitioning expression.

Optimal Reconciliation of Partitioning Expressions Based On Updatable Current Date and Timestamp

The following case applies to altering the partitioning expression of a populated table using an ALTER TABLE TO CURRENT statement.

Consider the following partitioning definition.

```
CREATE TABLE ppi_1 (
  i INTEGER,
  j DATE)
PRIMARY INDEX(i)
PARTITION BY CASE_N(j < DATE '2011-01-01',
                    j >= DATE '2011-01-01')
```

Assume that the current date is DATE '2011-01-01'. You cannot alter the partitioning definition for table *ppi_1* using an ordinary ALTER TABLE statement after *ppi_1* has been populated with rows, but you can alter its definition using an ALTER TABLE TO CURRENT statement if you define the table using a CURRENT_DATE function instead of specifying a simple date as the redefined table definition *ppi_2* demonstrates.

This redefinition of the partitioning expression for *ppi_1*, which replaces the DATE specification in its partitioning expression with a CURRENT_DATE specification, *can* be modified using an ALTER TABLE TO CURRENT statement.

```
CREATE TABLE ppi_2 (
  i INTEGER,
  j DATE)
PRIMARY INDEX(i)
```

```
PARTITION BY CASE_N(j < CURRENT_DATE,
                    j >= CURRENT_DATE)
```

For this example, the value of CURRENT_DATE resolves to DATE '2011-01-01'.

Partitioning Expression Columns and the Primary Index

For best practice, use these guidelines for the primary index and partitioning column set for a partitioned table. The primary index should:

- Distribute rows evenly across the AMPs.
- Facilitate direct row access.
- Be useful for satisfying join conditions.
- Be useful for aggregation.

For a unique primary index, you must include the partitioning column in the primary index, if the column is in the set of columns you want to be unique. Otherwise, the primary index can be nonunique and you can use a unique secondary index (USI) to enforce uniqueness on the primary index columns.

Loading Data into a Table with a Unique Secondary Index (USI)

Neither MultiLoad nor FastLoad can load rows into a table that has a USI. See *Teradata® MultiLoad Reference*, B035-2409 and *Teradata® FastLoad Reference*, B035-2411.

You can either use the Teradata Parallel Data Pump utility or you can use a batch load with error tracking. See *Teradata® Parallel Data Pump Reference*, B035-3021.

The method for performing a batch load is as follows:

1. Create a temporary transitional table with the same definition as the target table except for USIs, join indexes, hash indexes, referential constraints, or triggers.
2. Use either an INSERT ... SELECT or MERGE batch load with error tracking to move the bulk loaded rows into the target table.

For details about INSERT ... SELECT and MERGE, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146. For information about error tables, see [CREATE ERROR TABLE](#).

Planning Partitioned Primary Indexes

Although a primary index can have many partitions for access and joins, Optimizer costing can be affected negatively if there are too many unpopulated partitions.

Be aware that simply adding a partitioning column set to the primary index definition may not provide the best result. Instead, you should carefully plan your primary index and partitioning expression member columns in such a way that neither detracts from the other. In the ideal case, the primary index is defined identically with the partitioning column set in a manner that supports all of the possible uses equally well.

You cannot specify an EACH clause for character- or graphic-based test values in partitioning expression that is defined using a RANGE_N function.

If the primary index does not contain the entire partitioning column set, then you cannot define it to be unique. The following suggestions apply to this scenario.

- If the primary index columns must be an efficient access path, and there are many partitions, consider defining a USI or NUSI on the primary index column set.
- Consider defining fewer partitions when the table must also be accessed, joined, or aggregated on the primary index.

Query Conditions and Static Row Partition Elimination

The intent of the following guidelines is to indicate what to do, and what not to do, to maximize static row partition elimination to help optimize your queries.

As always, you need to verify that any technique you use produces the desired results.

- Avoiding specifying expressions or functions (excluding date-based built-in functions) on the partitioning column of a partitioned table.

The following examples show problematic table definitions and queries with rewritten definitions and queries for improved partitioning.

| Original Non-Optimal Table Definition and Queries | Rewritten Table Definitions and Queries |
|--|---|
| <pre>CREATE TABLE ... PARTITION BY x; SELECT ... WHERE x+1 IN (2,3);</pre> | <pre>CREATE TABLE ... PARTITION BY RANGE_N(x BETWEEN 1 AND 65533 EACH 1);</pre> <p>You can specify the maximum partition number anticipated for this 2-byte partitioning rather than 65,533 in this partition specification.</p> <pre>SELECT ... WHERE x IN (1,2);</pre> |
| <pre>CREATE TABLE ... PARTITION BY x+1; SELECT ... WHERE x IN (1,2);</pre> | <pre>CREATE TABLE ... PARTITION BY RANGE_N(x BETWEEN 0 AND 65532 EACH 1);</pre> <p>You can specify the maximum partition number anticipated for this 2-byte partitioning rather than 65,532 in this partition specification.</p> <pre>SELECT ... WHERE x IN (1,2);</pre> |

- Query predicates are optimally effective when you define the expression with constant conditions on the partitioning columns for the partitioned table.

The following predicates are all good examples of simple equality comparisons with constant conditions, where *d* is the partitioning column for the table.

- *d* = 10
- *d* >= 10 AND *d* <= 12
- *d* BETWEEN 10 AND 12
- *d* = 10+1
- *d* IN (20, 22, 24)
- *d* = 20 OR *d*=21

If the partitioning expression for the table is not defined using the RANGE_N function, query predicates are optimally effective when you specify them as equality conditions.

- Query predicates can be effective when they specify equality conditions with USING variables if the predicate specifies a single partition or the DATE or CURRENT_DATE built-in functions. For example, if the partitioning expression is date-based and the predicated condition on the partitioning column is defined using the CURRENT_DATE built-in function, then the Optimizer does static row partition elimination.

The Optimizer can also substitute USING request modifier values to further optimize the query plan. See Request Parsing in *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Multiple OR equality conditions on the same partitioning column do not permit row partition elimination.

If this is the case when you specify equality conditions on USING variables, consider either of the following alternatives.

- Use the UNION operator as a workaround.
- Substitute constants for the USING variables in the predicate conditions.

Instead of using parameterized queries, consider one of these alternatives.

- Use unparameterized queries.
- Use a preprocessor to substitute for the parameterized variables before you submit the query.

Join Conditions and Partitioned Primary Indexes

Follow these guidelines whenever possible when you are formulating join conditions for your queries against PPI tables.

- If possible, specify join conditions with an equality join on the primary index column set and partitioning column set.

This type of join condition uses an efficient, rowkey-based merge join.

- Consider adding the partitioning column for the PPI table to the other table in the join.
- For the case where the primary index column set does not contain the complete partitioning column set, specifying an equality join on the primary index column set, *but not on the partitioning column set*. It is better for there to be fewer partitions after any row partition elimination, and the fewer partitions that remain, the better.

- The Optimizer can specify a sliding window join when there is a small number of partitions; otherwise, the table might need to be spooled and sorted.
- Use RANGE_N to define fewer partitions for the table.
- Specify join conditions on the partitioning columns to influence the Optimizer to use row partition elimination to reduce the number of partitions involved in the join.
- Note that the Optimizer does not know whether a partition is unpopulated or populated, so it assumes that all defined partitions are populated with rows.

PARTITION statistics help the Optimizer to estimate the cost of join plans, but the join plan selected by the Optimizer cannot assume that partitions are unpopulated.

Effects of Dynamic Row Partition Elimination on Joins

The following guidelines apply to maximizing dynamic row partition elimination for your join queries (dynamic row partition elimination is not supported with merge join methods for multilevel partitioned tables. See *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142) against a partitioned table.

- Dynamic row partition elimination improves performance when a partitioned table and another table are joined by an equality condition on the partitioning column set of the partitioned table.

Note that this does not work with an inclusion product join, which the Optimizer specifies to work with IN (*subquery*) join conditions.

- Only those partitions that are required to answer a request are involved in the join.

When performing join steps, the system determines the qualifying partitions of the partitioned table dynamically based on the values of rows from the other table.

Instead of a product join against all the rows in the partitioned table, the Optimizer assigns a product join only for a row of the other table against a single partition instead of the entire table.

If there are 100 partitions in the partitioned table and only 5 are needed to answer a join request, the other 95 are not joined, and the system realizes a 95% resource saving for that join operation.

- Always collect statistics on the join columns.

The following table definitions, EXPLAIN request, and EXPLAIN report are used to demonstrate how dynamic row partition elimination can facilitate join operations where one of the tables in the join is a PPI table.

```
CREATE SET TABLE DB.sales (      -- Defines total day sales
  prodid  INTEGER,                -- for each product
  saledate DATE FORMAT 'YYYY-MM-DD',
  amount  DECIMAL(10,2))
PRIMARY INDEX (prodid, saledate)
PARTITION BY RANGE_N(saledate BETWEEN DATE '2004-01-01'
                      AND      DATE '2004-12-31'
                      EACH INTERVAL '1' MONTH );
```



```

CREATE SET TABLE DB.fiscal_month (      -- Defines the days in
  yr          SMALLINT NOT NULL,          -- in each fiscal month.
  mth         SMALLINT NOT NULL,          -- A fiscal month may
  dayofmth    DATE NOT NULL)              -- partially span up
PRIMARY INDEX (yr, mth);                  -- to 2 calendar months.

```

For example, fiscal month 9 in 2004 is from August 30 to September 26, or 28 days.

Assume a 2 AMP system and 10 product sales per day.

An EXPLAIN of the SELECT statement below includes a RETRIEVE step from DB.fiscal_month by way of the primary index “DB.fiscal_month.yr = 2004, DB.fiscal_month.mth = 9” followed by a SORT to partition by rowkey. Next, JOIN step to all partitions of DB.sales. using a product join, with a join condition of (“DB.sales.saledate = dayofmth”) enhanced by dynamic partition elimination.

```

SELECT yr, mth, SUM(amount)
  FROM DB.sales, DB.fiscal_month
 WHERE saledate = dayofmth
    AND   yr = 2004
    AND   mth = 9
 GROUP BY yr, mth;

```

Rules and Usage Notes for Partitioned Tables

The following guidelines and restrictions apply to all partitioned tables.

- When you are doing an analysis of whether a table should be partitioned or not, always weigh the costs of a given strategy set against its benefits carefully.

You should consider all of the following factors at minimum.

- The partitioning expression.

Consider all of the following factors when making your analysis of the partitioning expression.

- Would the proposed workloads against the table be better supported by a partitioning expression based on CASE_N, RANGE_N, or some other expression?
- Should the partitioning expression specify a NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN option?
- Should the table be partitioned on only 1 level or on multiple levels?

- The query workloads that will be accessing the partitioned table.

This factor must be examined at both the specific, or particular, level and at the general, overall level.

Among the factors you should consider are the following.

- Performance

- Does a nonpartitioned table perform better than a partitioned table for the given workload and for particularly critical queries?
- Is 1 partitioning strategy more high-performing than others?
- Do other indexes such as USIs, NUSIs, join indexes, or hash indexes improve query performance?
- Does a partitioning expression cause significant row or column partition elimination to occur or not?
- Access methods and predicate conditions
 - Primary index access, secondary index access, or something else?
 - Do typical queries specify an equality condition on primary index with the complete partitioning column set included?
 - Do typical queries specify a *non*-equality condition on the primary index?
- Join strategies
 - Do typical queries specify an equality condition on the primary index column set (and partitioning column set if they are not identical)?
 - Do typical queries specify an equality condition on the primary index column set but not on the partitioning column set?
 - Do typical query conditions support row partition elimination?
- Data maintenance
 - What are the relative effects of a partitioned table versus a nonpartitioned table with respect to the maintenance workload for the table?
 - If you must define a USI on the primary index to make it unique, how much additional maintenance is required to update the USI subtable?
- Frequency and ease of altering partitions
 - Is a process in place to ensure that ranges are added and dropped as necessary?
 - Does the partitioning expression permit you to add and drop ranges?
 - If the number of rows moved by dropping and adding ranges causes large numbers of rows to be moved, do you have a process in place to instead create a new table with the desired partitioning in place, then INSERT ... SELECT or MERGE the source table rows into the newly created target table with error logging?
 - If you want to delete rows when their partition is dropped, have you specified NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN partitions?
- Backup and restore strategy
- You can specify row partitioning for any of the following table types.
 - SET or MULTiset table
 - Base data table
 - Global temporary table
 - Volatile table

- Non-row-compressed join index
- You cannot specify row partitioning for any of the following table types.
 - Queue table
 - Derived table
 - Row-compressed join index
 - Hash index
 - Secondary index
- You can specify column partitioning for any of the following table types.
 - MULTiset table with no primary index
 - Base data table with no primary index
 - Non-row-compressed join index with no primary index
- You cannot specify column partitioning for any of the following table types.
 - SET table
 - Global temporary table
 - Global temporary trace table
 - Volatile table
 - Queue table
 - Derived table
 - Non-row-compressed join index
- You cannot specify a partitioned primary index for any of the following table types.
 - Nonpartitioned NoPI table
 - Column-partitioned table
 - Global temporary trace table
 - Row-compressed join index
 - Hash index
 - Secondary index
- You cannot define a partitioning expression using an identity column as a partitioning column.
- You can only define a PPI as UNIQUE if all the columns referenced in the partitioning expression are also referenced in the column list that defines the primary index.
- You cannot specify LOB columns in a partitioning expression.
- You cannot specify either external or SQL UDFs in a partitioning expression.
- You cannot specify PERIOD columns directly in a partitioning expression, but you *can* specify the BEGIN and END bound functions on Period columns in a partitioning expression.
- You can only define a PRIMARY KEY or UNIQUE constraint on the same set of columns as the primary index column list if both of the following conditions are also true.
 - Not all of the partitioning columns are included in the primary index column list.

- A USI is not explicitly defined on the same column set as the primary index. Any such PRIMARY KEY or UNIQUE constraint *implicitly* defines a USI on the same set of columns as those defining the primary index column list.
- You cannot define a USI on a table with a PPI if all the partitioning columns are also referenced in the primary index column list.

Instead, define the primary index as unique.

- A NUSI on a table with a PPI must be value-ordered if it is defined on the same set of columns as the primary index column list and all the partitioning columns are duplicated in the primary index column list.

Note that a *multicolumn* NUSI created with an ORDER BY clause counts as 2 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.

- Do not assume that NUSIs are needed for column-partitioned tables.

You should only add a NUSI to a column-partitioned table if it provides a benefit.

- Columns referenced in *partitioning_expression* must be drawn from the set of columns defined for the table on which the partitioning is defined for row-partitioned tables.

This is not true for column-partitioned tables, which do not have a primary index.

- If the data type for the result of *partitioning_expression* is not INTEGER, BIGINT, or CHARACTER, then the system implicitly casts the value to the INTEGER or BIGINT type, depending on its size.
- If you attempt to insert or update a row of a partitioned table, and the partitioning expression does not produce a result in the range 1 - 9,223,372,036,854,775,807 after casting to BIGINT if the result is not already typed as BIGINT, then the system returns an error for the insert or update operation.

If you want to develop a partitioning expression that permits all rows to be inserted or updated, you can use CASE expressions, options on the CASE_N and RANGE_N functions, and asterisks in the RANGE_N function to construct such an expression. See *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for more information about the use of CASE_N and RANGE_N functions.

- Because DATE values can be cast to INTEGER and BIGINT, they are valid values for partitioning expressions; however, only values between DATE '1900-01-01' and DATE '1906-12-31' have values between 1 and 9,223,372,036,854,775,807 when cast to BIGINT.

To compensate, you can adjust the partitioning expression similarly to the following example.

```
PARTITION BY d1 - 36890
```

where *d1* can then have values between DATE '2001-01-01' and DATE '2007-12-31' when cast to INTEGER values.

An even better way to handle dates is to use only the RANGE_N function when the partitioning column is of DATE type, as shown in the following example.

```
PARTITION BY RANGE_N(d1 BETWEEN DATE '2001-01-01'
                     AND DATE '2007-12-31')
                     EACH INTERVAL '1' DAY)
```

- If *partitioning_expression* is only a CASE_N function, then the number of conditions defined must be less than or equal to 214,748,647 (see the description of CASE_N in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145) to allow for the NO CASE [OR UNKNOWN] and UNKNOWN options.

This restriction applies even if the partitioning expression uses the CASE_N function in a subexpression.

Furthermore, the system enforces this restriction even if you do not specify additional options.

- If *partitioning_expression* is only a RANGE_N function, then the number of combined ranges defined for the function must be less than or equal to 9,223,372,036,854,775,805 to allow for the NO RANGE [OR UNKNOWN] and UNKNOWN options. When ranges are defined explicitly (*not* using an EACH clause), then other size limits such as those for table headers, request text, implied constraint checks, or EVL code, are likely to be reached before you exceed the limit of 65,535 ranges.

This restriction is enforced even if those options are *not* specified. This makes it possible to alter the index definition later to include the NO RANGE [OR UNKNOWN] or UNKNOWN options.

This rule does not apply if the partitioning expression includes the RANGE_N function in a subexpression.

See the description of RANGE_N in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145) for additional information and [Modifying Partitioning Using the ADD RANGE and DROP RANGE Options](#) for the implication of the NO RANGE option for removing partition ranges from a table using the ALTER TABLE statement.

- You cannot collect statistics on the system-derived PARTITION column for any of the following table types.
 - Join index (see [CREATE JOIN INDEX](#))
 - Global temporary table
 - Volatile table

See *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for more information about partitioned primary indexes.

See [Redefining the Primary, Primary AMP, or Partitioning for a Table](#) for information about how to use the ALTER TABLE statement to modify the primary index definition for a table.

Partitioning CHECK Constraints for Partitioned Tables With 2-Byte Partitioning

Vantage derives a table-level partitioning CHECK constraint from the partitioning expression for partitioned tables. The text for this constraint, which is derived for table constraints, partitioning constraints, and named constraints cannot exceed 16,000 characters.

The following diagrams show the two forms of this partitioning CHECK constraint derived for single-level partitioning. The forms differ depending on whether the partitioning expression has an INTEGER data type or not.

- The first form applies to single-level 2-byte partitioning expressions that do not have an INTEGER data type and are not defined only by a RANGE_N function.

In this case, the type of the expression must be cast to INTEGER.

```
CHECK (
  ( CAST ( partitioning_expression ) AS INTEGER )
) BETWEEN 1 AND max
```

- The second form applies to single-level 2-byte partitioning expressions that have an INTEGER data type.

```
CHECK ( partitioning_expression ) BETWEEN 1 AND max
```

partitioning_expression

Partition number returned by the single-level partitioning expression.

max

Maximum number of partitions defined by *partitioning_expression*.

If the partitioning expression is defined using something other than a RANGE_N or CASE_N function, the value of *max* is 65,535.

If the partitioning expression is defined using only a RANGE_N or CASE_N function, the value of *max* is the number of partitions defined by the partitioning expression.

See [Rules and Usage Notes for Partitioned Tables](#) and [Restrictions for Multilevel Partitioning](#) for information about the table-level partitioning CHECK constraints that Vantage creates for single-level and multilevel partitioned primary index tables and join indexes.

- The third form applies to multilevel 2-byte partitioning expressions.

The following diagram provides the form of this partitioning CHECK constraint that Vantage derives for a multilevel partitioned primary index with 2-byte partitioning.

```
CHECK (
  /* nn */ partitioning_expression_1 IS NOT NULL
  AND partitioning_expression_2 IS NOT NULL
  [ partitioning_expression_n IS NOT NULL ] [...]
)
```

nn

Number of levels, or number of partitioning expressions, in the multilevel partitioning. *nn* can range between 02 and 15, inclusive.

partitioning_expression_1

The partition number returned by the first multilevel partitioning expression.

partitioning_expression_2

The partition number returned by the second multilevel partitioning expression.

partitioning_expression_n

The partition number returned by the *n*th multilevel partitioning expression.

If the partitioning has 3 levels, there are 3 NOT NULL partitioning expressions in the implied constraint, if the partitioning has 10 levels, there are 10 NOT NULL partitioning expressions in the implied constraint, and so on.

For example, suppose you create the following 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_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);
```

The partitioning CHECK constraint SQL text that would be stored in *DBC.TableConstraints* for this multilevel partitioned primary index with 2-byte partitioning is as follows.

```
CHECK (/ *02*/ RANGE_N(o_custkey  BETWEEN 0
                      AND 49999
```

```

                                EACH 100)
IS NOT NULL
AND RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                                AND DATE '2006-12-31'
                                EACH INTERVAL '1' MONTH)
IS NOT NULL )

```

You could use the following SELECT statement to retrieve index constraint information for each of the multilevel partitioned objects.

```

SELECT *
FROM DBC.TableConstraints
WHERE ConstraintType = 'Q'
AND SUBSTRING(TableCheck FROM 1 FOR 13) >= 'CHECK (/02*/'
AND SUBSTRING(TableCheck FROM 1 FOR 13) <= 'CHECK (/15*/';

```

Partitioning CHECK Constraint for Partitioned Tables with 8-Byte Partitioning

Just as it does for a 2-byte partitioned primary index, Vantage derives a partitioning CHECK constraint for 8-byte partitioning from the partitioning expressions.

The text for this CHECK constraint that is derived for table constraints, partitioning constraints, and named constraints, cannot exceed 16,000 characters.

Vantage uses this form of partitioning CHECK constraint when any one of the following things is true about the partitioning expression for a table or join index.

- A partitioning expression for at least 1 level is defined only by a RANGE_N function with a BIGINT data type.
- At least 1 partitioning level in the partitioning expression specifies an ADD clause.
- The table or join index has 8-byte partitioning.
- The table or join index is column-partitioned.
- The number of partitions for at least 1 partitioning level other than level 1 of a populated table changes when the table is altered with an ALTER TABLE statement.
- The number of partitions for at least 1 partitioning level other than level 1 of an unpopulated table decreases when the table is altered with an ALTER TABLE statement.

The format for this partitioning CHECK constraint text for multilevel partitioning with 8-byte partitioning, which applies to both row-partitioned and column-partitioned tables and join indexes, is as follows.

```

CHECK ( /*nn bb cc*/ partitioning_constraint_1 [ AND partitioning_constraint_n ][...] )

```

nn

The number of partitioning levels.

For 2-byte partitioning, *nn* ranges between 01 and 15, inclusive.

For 8-byte partitioning, *nn* ranges between 01 and 62, inclusive.

bb

The number of bytes used to store the internal partition number in the row header.

For 2-byte partitioning, *bb* = 02.

For 8-byte partitioning, *bb* = 08.

cc

The column-partitioning level for a column-partitioned table.

If the table is not column-partitioned, *cc* = 00.

If the table is column-partitioned, the value of *cc* ranges between 01 and *nn*, inclusive, where *nn* is the number of partitioning levels for the table.

partitioning_constraint_i

The partitioning expression at partitioning level *i*.

i

i, the partitioning level of the partitioning expression for the table or join index, is in the range [1, *nn*]. Vantage does not store leading zeros for the value of *i*.

If the table or join index is row-partitioned at partitioning level *i*, the form of *partitioning_constraint_i* is as follows:

partitioning_constraint_i /**i d+a**/ IS NOT NULL

If the table or join index is column-partitioned at partitioning level *i*, the form of *partitioning_constraint_i* is as follows:

PARTITION#*Li* /**i d+a**/=1

d

The number of defined partitions for partitioning level *i*. For a column partitioning level, this value includes the 2 column partitions reserved for internal use. Vantage does not store leading zeros for the value of *d*.

a

The number of partitions that could be added to the partitioning level (which might be 0). If Vantage uses this form of partitioning CHECK constraint, or if the partitioning level is level 1, the value of *a* is an INTEGER number. Vantage does not store leading zeros for the value of *a*.

n

The number of the highest partitioning level defined for the partitioning expression. For example, if the partitioning expression for a table has 18 partitioning levels, then the value for *n* is 18, and the partitioning CHECK constraint documents 18 partitioning expressions.

Each of the individual partitioning constraints in the partitioning CHECK constraint corresponds to a level of partitioning in the order defined for the table.

See [ALTER TABLE \(Basic Table Parameters\)](#) for more information about the ALTER TABLE statement.

The following example demonstrates the form of partitioning CHECK constraint that Vantage uses for 8-byte partitioning. Assume you create the following 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_comment       VARCHAR(79))
NO PRIMARY INDEX
PARTITION BY (RANGE_N(o_custkey BETWEEN  0
                        AND 100000
                        EACH      1),
              COLUMN)
UNIQUE INDEX (o_orderkey);
```

The product of the maximum partition number of each partitioning level is 100,000 (100,000 * 1), which is greater than 65,535, so the table has 8-byte partitioning. The table-level partitioning CHECK constraint text for this 8-byte partitioning is as follows.

```
CHECK (/*02 08 02*/ RANGE_N(o_custkey BETWEEN  0
                        AND 100000
                        EACH      1
/*1 100001+485440633518572409*/
IS NOT NULL AND PARTITION#L2 /*2 9+10*/=1)
```

You can use the following SELECT statement to retrieve the level for the column partitioning for each of the objects that have column partitioning in the system.

```
SELECT DBaseId, TVMId, ColumnPartitioningLevel
      (TITLE 'Column Partitioning Level')
FROM DBC.TableConstraints
WHERE ConstraintType = 'Q'
```

```
AND    ColumnPartitioningLevel >= 1
ORDER BY DBaseId, TVMId;
```

See [Rules and Usage Notes for Partitioned Tables](#) and [Restrictions for Multilevel Partitioning](#) for information about the table-level partitioning CHECK constraints that Vantage creates for single-level and multilevel partitioning tables and join indexes with 2-byte partitioning.

Restrictions for Multilevel Partitioning

The following set of rules applies to multilevel partitioning in addition to the universal rules set listed in [Rules and Usage Notes for Partitioned Tables](#), all of which also apply to multilevel partitioning.

While a partitioning expression can have multiple column references, and a column can be referenced in more than 1 partitioning expression for the primary index, you should consider the usefulness of defining such a partitioning scheme. The most useful partitioning expressions are generally those that have only a single reference to a column that is not referenced in the other partitioning expressions of the primary index.

- If you specify more than 1 partitioning expression in the PARTITION BY clause, each such partitioning expression must consist of either a single RANGE_N or a single CASE_N function; otherwise, the system returns an error to the requestor.
- If you specify more than 1 partitioning expression in the PARTITION BY clause, the product of the number of partitions defined by each partitioning expression cannot be less than 4 nor can it exceed 9,223,372,036,854,775,807, inclusive. Otherwise, the system returns an error to the requestor. The minimum is four because a multilevel partitioning must, by definition, have at least 2 levels and each of those levels must have at least 2 partitions. The product is $2 \times 2 = 4$.

Additionally, each partitioning expression you specify must define *at least* 2 partitions. Otherwise, the system returns an error to the requestor.

This rule implies that the maximum number of partitioning expressions is 9,223,372,036,854,775,807.

This is because $2^{63} = 9,223,372,036,854,775,808$, which is larger than the maximum number of partitions that can be defined for a single table.

If you define more than 2 partitions at 1 or more levels, the number of partitioning expressions can be limited still further.

- A partitioning expression cannot contain the system-derived columns PARTITION#L1 through PARTITION#15, inclusive.

If this condition is not satisfied, the system returns an error to the requestor.

Combined Partitioning Expressions

This topic describes what combined partitioning expressions are and how Vantage uses them.

- Vantage derives a combined partitioning expression from the partitioning expressions defined for each individual row partitioning level and a column partition number of 1 for the column-partitioning level.

For n partitioning levels, where n is the number of partitioning levels defined, the combined partitioning expression is defined as follows.

$$\text{Combined partitioning expression} = \sum_{i=1}^{n-1} ((p_i - 1) \times dd_i) + p_n$$

where:

| Equation element | Description |
|------------------|--|
| p_i | Row partitioning expression at level i , numbering from left to right, or 1 for a column partitioning level. |
| dd_i | Constant value equal to the following product. $\prod_{j=i+1}^n d_j$ |
| d_j | Maximum partition number defined at level j . |

The parentheses delimiting the summation are not included in the combined partitioning expression. For example, this summation expands for 1-, 2-, 3-, 4-, and 5-level partitioning as the following combined partitioning expressions, respectively, where p_i and dd_i are defined as they were in the preceding definition for a combined partitioning expression.

| Partitioning Level | Expanded Summation |
|--------------------|---|
| 1 | p_1 |
| 2 | $(p_1-1)*dd_1+p_2$ |
| 3 | $(p_1-1)*dd_1+(p_2-1)*dd_2+p_3$ |
| 4 | $(p_1-1)*dd_1+(p_2-1)*dd_2+(p_3-1)*dd_3+p_4$ |
| 5 | $(p_1-1)*dd_1+(p_2-1)*dd_2+(p_3-1)*dd_3+(p_4-1)*dd_4+p_5$ |

The combined partitioning expression reduces to p_1 for single-level row partitioning, so there is no actual change in the usage rules for this case.

For column partitioning, the combined partition number for a specific column partition value of a table row can be derived from the combined partition number for this table row as follows:

$$\text{combined_part_number-specific_col_part_value_of_row} = \text{cpn} + (c - 1) * dd_i$$

where:

| Equation Element | Description |
|--|---|
| <i>combined_part_number-specific_col_part_value_of_row</i> | Combined partition number for a specific column partition value of a table row. |
| <i>cpn</i> | Combined partition for the row. |
| <i>c</i> | Column partition number for this column partition value within the row. |
| <i>dd_i</i> | Constant value equal to 1. |

This is the same as computing the combined partitioning expression for a row-partitioned table row except instead of using 1 for the column-partitioning level, this equation uses the column partition number corresponding to the specific column partition value of the table row.

While column partitioning can be defined at any level, it is recommended in most cases to put the column-partitioning level first before any row partitioning. Some considerations that might lead to assigning the column partitioning to a lower level in the partitioning hierarchy are potential improvements for cylinder migration and temperature-based block compression effectiveness for hot and cold data.

The combined partitioning expression defines how rows are ultimately partitioned on each AMP. The result of the combined partitioning expression for specific values of the partitioning column is referred to as the *combined partition number*.

- With multilevel partitioning, the number of partitions defined by the combined partitioning expression is likely to be large. If there is a large number of populated partitions, performance of primary index access, joins, and aggregations on the primary index might be degraded. Therefore, multilevel partitioning may not be an appropriate choice when these operations occur frequently without significant simultaneous row partition elimination. However, a large number of row partitions for the combined partitioning expression allows for finer row partition elimination. See [Determining the Partitioning Granularity](#).

Rules and Usage Notes for Multilevel Partitioning

The following guidelines and rules apply to various aspects of tables defined with multilevel partitioning.

- The order of partitioning expressions in the PARTITION BY clause does not affect the ability of the system to eliminate partitions, but does affect how effective the scan of partitions is. This should not be a concern if there are many rows (the word *many* in this case is defined to mean the rows span multiple data blocks) in each of the internal partitions.

Consider the following 2 cases.

- For a table with 65,535 combined partitions (65,535 partitions for the combined partitioning expression), 6.5535×10^9 rows per AMP, 100 byte rows, and 50 KB data blocks, each combined partition spans 200 to 201 data blocks (assuming rows are distributed evenly among the partitions). In this case, skipping over internal partitions should not incur too much overhead. All,

or nearly all, of the rows in the data blocks read should qualify, and at least 199 data blocks are skipped between each set of data blocks read.

- If the table only has 6.5535×10^6 rows per AMP, each combined partition has only about 0.2 data blocks. Assuming a worst case where only every other partition at the lowest level is eliminated, every data block must be read, and a full-table scan would be more efficient. The Optimizer does not cost the difference between these 2 choices. If it can do row partition elimination, it will, even if a full-table scan would be more efficient.

If there are 5 partitions at the lowest level, and 4 of the 5 partitions are eliminated, every data block must still be read.

If there are 6 partitions at the lowest level, and 5 of the 6 partitions are eliminated, some data blocks might be skipped, but probably not enough to overcome the overhead incurred, and this might be less efficient than a full-table scan.

With a large number of partitions at the lowest level as well as a large number of partitions being eliminated at the lowest level, enough additional data blocks might be skipped that the overhead incurred can be overcome, with the result possibly being more efficient than a full-table scan.

This case is *not* a good use of multilevel partitioning. Instead, you should consider an alternative partitioning that either has multiple data blocks for each internal partition or the internal partition is not populated. A more useful partitioning scheme is one that defines fewer levels of partitioning, or fewer partitions per level, with the lowest level having the most partitions so there are more rows per combined partition.

For example, if 1 level was partitioned in intervals of 1 day, changing the interval to 1 week or 1 month might be better.

Row partition elimination at lower levels in the row partition hierarchy requires more skipping, which could cause more I/O as well as increasing the CPU path length.

To obtain the best performance, you should place partitioning expressions that are more likely to evoke row partition elimination for common workloads at higher levels in the row partitioning hierarchy. Those partitioning expressions that are *not* likely to evoke row partition elimination should either be placed at lower levels in the hierarchy or eliminated altogether.

You should also consider placing the row partitioning expression with the highest number of row partitions at the lowest level in the partition hierarchy. Recall, however, that the ordering of row partitioning expressions *might* not be a significant factor if there are many data blocks per combined partition.

Note that if the number of partitions is expected to be altered for a RANGE_N partitioning expression, that partitioning expression must be specified at the highest level of the hierarchy.

You can use the following query to find the average number of rows per combined partition.

```
SELECT AVG(pc) FROM (
  SELECT COUNT(*) AS pc
  FROM t
```

```
GROUP BY PARTITION
) AS pt;
```

You can use the following query to find the average number of data blocks per combined partition.

```
USING (b FLOAT, r FLOAT)
SELECT (:r/:b) * AVG(pc) FROM (
  SELECT COUNT(*) AS pc
  FROM t
  GROUP BY PARTITION
) AS pt;
```

t

Table name.

r

Average block size.

b

Row size.

- Multilevel partitioning typically defines a large number of partitions for the combined partitioning expression. If there are a large number of populated partitions for the combined partitioning expression, performance of primary index access, joins, and aggregations on the primary index might be degraded.

As a result, multilevel partitioning might be an appropriate choice when these operations are rarely performed without also obtaining significant row partition elimination. Note, however, that a large number of partitions for the combined partitioning expression reduces the number of data blocks that must be processed when a small number of partitions needs to be accessed because of row partition elimination.

- If you attempt to insert or update a row of the table with a multilevel partitioning such that any one of the partitioning expressions for that row evaluates to null, the insert or update request (in Teradata session mode) or transaction (in ANSI session mode) aborts and the system returns an error message to the requestor.
- If you want the table to be defined in such a way that all valid rows can be inserted and so all its rows are updatable, you must construct each partitioning expression in such a way that a row with any value or null for its partitioning columns is assigned to some partition number such that the RANGE_N or CASE_N function does not return null.

You can use the NO RANGE [OR UNKNOWN] and UNKNOWN options and asterisks in the RANGE_N function and NO CASE [OR UNKNOWN] and UNKNOWN options in the CASE_N function to assist in constructing such partitioning expressions.

Note that you should not use these options to construct partitioning expressions that permit the inclusion of rows that should not be in the table. By having partitions defined for them, such rows can have a negative effect on query performance.

Use of these options and asterisks can also prevent the effective use of the ALTER TABLE statement to alter the partitioning of the table. See [ALTER TABLE \(Basic Table Parameters\)](#).

- Expression evaluation errors such as divide by zero can occur during evaluation of the partitioning expressions, causing ANSI mode requests or Teradata mode transactions to be rolled back.

You should construct your partitioning expressions to avoid such errors.

- Any excess partitions are reserved for level 1 and cannot be added to any other level. That is, the maximum number of partitions for level 1 for 2-byte partitioning is the following.

$$\text{Maximum number of partitions}_{\text{level 1}} = \left\lfloor \frac{65535}{\left(\prod_{i=2}^n d_i \right)} \right\rfloor$$

The maximum number of partitions for level 1 for 8-byte partitioning is the following.

$$\text{Maximum number of partitions}_{\text{level 1}} = \left\lfloor \frac{9223372036854775807}{\left(\prod_{i=2}^n d_i \right)} \right\rfloor$$

where:

| Equation element ... | Specifies |
|-------------------------|---|
| $\lfloor \quad \rfloor$ | the floor, or lower truncated value, of the expression. |
| d_i | the number of partitions defined at level i . |

For some cases, level 1 can be altered to have between 2 and the maximum number of partitions.

Purpose and Behavior of the NO RANGE and UNKNOWN Partitions

The system uses the UNKNOWN partition to assign rows whose partition number cannot be evaluated because the column value on which they are partitioned evaluates to null.

When you create a partitioned table using the RANGE_N function to specify its partitioning expression, but *without* defining a NO RANGE partition, any row that does not evaluate to a partition numbered from 1 through 9,223,372,036,854,775,805, inclusive, when its table definition is altered using an ALTER TABLE statement with a WITH INSERT [INTO] clause, is deleted from the source table and moved into the target save table specified by the WITH INSERT [INTO] clause.

When you create a partitioned table using the `RANGE_N` function to specify its partitioning expression *with* a `NO RANGE` partition, the rows for any dropped partition are moved into `NO RANGE`, even if you specify a `WITH DELETE` or `WITH INSERT [INTO]` clause in the `ALTER TABLE` statement that drops the partition. This is because by definition the `NO RANGE` partition contains all rows whose partition number does not evaluate to an explicitly defined partition range or to a valid partition in the range 1 to 9,223,372,036,854,775,805, inclusive.

You can see this with a partitioned table defined with a `NO RANGE` partition by using a `WITH INSERT [INTO]` or `WITH DELETE` clause specified by an `ALTER TABLE` statement. Either the data does not move into another table or it is not deleted from the specified table. Instead, it moves into the `NO RANGE` partition of the same table.

For example, consider the following PPI table definition.

```
CREATE SET TABLE ppi_salestable, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT (
  prod_code          CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
  sales_date         DATE FORMAT 'YYYY-MM-DD',
  agent_id           CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
  quantity_sold      INTEGER,
  prod_description   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 '2001-12-31' EACH INTERVAL '1' MONTH,
    '2002-01-01'(DATE)
  AND    '2002-12-31'(DATE) EACH INTERVAL '1' MONTH,
    '2003-01-01'(DATE)
  AND '2003-12-31'(DATE) EACH INTERVAL '1' MONTH,
  NO RANGE);
```

This table has 37 partitions: 1 for each of the months in the three years it is partitioned over and 1 for `NO RANGE`. Note that the partition numbers for the `NO RANGE` and `UNKNOWN` partitions are assigned internally to fixed partition numbers; therefore, they are not necessarily the highest numbered partitions for any given partitioned table.

You also create the following table to be used as the target table for an `ALTER TABLE` statement `WITH INSERT [INTO]` clause.

```
CREATE SET TABLE ppi_salestable1, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
```

```

CHECKSUM = DEFAULT (
  prod_code          CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
  sales_date         DATE FORMAT 'YYYY-MM-DD',
  agent_id           CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
  quantity_sold      INTEGER,
  prod_description   VARCHAR(50) CHARACTER SET LATIN NOT
                    CASESPECIFIC)
PRIMARY INDEX (prod_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);

```

Now insert some rows into the source table.

```

INSERT INTO ppi_salestable VALUES('PC2','2001-01-10','AG2',5,'PC');

INSERT INTO ppi_salestable VALUES('PC3','2001-03-10','AG2',5,'PC');

INSERT INTO ppi_salestable VALUES('PC4','2002-05-10','AG2',5,'PC');

INSERT INTO ppi_salestable VALUES('PC5','2003-07-10','AG2',5,'PC');

INSERT INTO ppi_salestable VALUES('PC5','2004-07-10','AG2',5,'PC');

```

Select all the columns from the table to verify the rows you had intended to insert actually were inserted.

```

SELECT PARTITION, prod_code, sales_date, agent_id, quantity_sold,
        prod_description
FROM ppi_salestable
ORDER BY 1;

PARTITION prod_code sales_date agent_id quantity_sold prod_description
-----
      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

```

You now drop the 12 partitions for the year 2001 with the intent of moving them into the table specified by the WITH INSERT INTO clause, *ppi_salestable1*.

```

ALTER TABLE ppi_salestable
  MODIFY PRIMARY INDEX (product_code, sales_date, agent_id)
  DROP RANGE BETWEEN DATE '2001-01-01'
    AND      DATE '2001-12-31'
    EACH INTERVAL '1' MONTH
  WITH INSERT INTO ppi_salestable1;

```

```

*** Table has been modified.
*** Total elapsed time was 1 second.

```

Having dropped the set of partitions for the year 2001, you perform a quick check to ensure that the rows in those partitions were moved out of *ppi_salestable* and into *ppi_salestable1*.

```

SELECT PARTITION, prod_code, sales_date, agent_id, quantity_sold,
       prod_description
FROM ppi_salestable
ORDER BY 1;

```

```

*** Query completed. 5 rows found. 6 columns returned.
*** Total elapsed time was 1 second.

```

| PARTITION | prod_code | sales_date | agent_id | quantity_sold | prod_description |
|-----------|------------|-------------------|------------|---------------|------------------|
| 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 |

Notice that the data did *not* move into *ppi_salestable1* as intended, but instead moves into the NO RANGE partition within the same table, *ppi_salestable* (see the entries marked in boldface). This is because of the presence of the NO RANGE partition in *ppi_salestable*. Because of that specification, any rows in partitions deleted from *ppi_salestable* or whose partition number evaluates to a value outside the inclusive range 1-9,223,372,036,854,775,805 are moved into the NO RANGE partition whether you define a WITH INSERT [INTO] or WITH DELETE clause or not. Therefore, if you want to use the functionality provided by these clauses, no *not* specify a NO RANGE partition for the source table.

See [Rules For Altering a Partitioning For a Table](#). Also see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for additional information about the RANGE_N and CASE_N functions and how the system uses them to determine partition numbers.

Determining the Partitioning Granularity

The key guideline for determining the optimum granularity for the partitions of a partitioned table is the nature of the workloads that most commonly access the partitioned table or join index. The higher the

number of partitions you define for a table or join index, the more likely an appropriate range query against the database object will perform more quickly, given that the partition granularity is such that the Optimizer can eliminate all but 1 partition.

On the other hand, it is generally best to avoid specifying too fine a partition granularity. For example if query workloads never access data at a granularity of less than one month, there is no benefit to be gained by defining partitions with a granularity of less than one month. Furthermore, unnecessarily fine partition granularity is likely to increase the maintenance load for a partitioned table, which can lead to degradation of overall system performance. In the end, even though too fine a partition granularity itself does not itself introduce performance degradations, the underlying maintenance on such a table *can* indirectly degrade performance.

Memory Limitations Related to Partitioned Tables

You might encounter several different memory limitations when working with partitioned tables. The following table addresses some of these limitations by documenting the relevant error message numbers and providing explanations or workarounds for them.

| Message Number | Problem | Possible Cause | Remedy |
|----------------|---|--|--|
| 3708 | Table header size limit exceeded (see <i>Teradata Vantage™ - Database Design</i> , B035-1094). The limit is 64 KB for a thin table header and 1 MB for a fat table header. | Too many columns in the table. | If possible, reduce the number of columns in the table. |
| | | Too many distinct values compressed in the table. | If possible, reduce the number of compressed values. |
| | | Partitioning expressions are too complex. | Simplify the partitioning expressions or reduce their number. If possible, use RANGE_N with EACH clauses instead of CASE_N. |
| 3710 | Parser memory size limit exceeded. The limit is variable and is determined by the value of the MaxParseTreeSegs parameter in DBS Control. | Current values for DBS Control parameters do not allocate enough parser memory to process the request. | Change the values for the following DBS Control parameters to the specified settings. You might find that your query workloads that require these values need to be increased still further. See <i>Teradata Vantage™ - Database Utilities</i> , B035-1102 for information about how to change the value for MaxParseTreeSegs in the DBS Control record. <ul style="list-style-type: none"> If the problem occurs with PPI tables that do not have hash or join indexes, then change the value for MaxParseTreeSegs to this value. <ul style="list-style-type: none"> 2,000 for byte-packed systems. 4,000 for byte-aligned systems. |

| Message Number | Problem | Possible Cause | Remedy |
|----------------|--|--|---|
| | | | <ul style="list-style-type: none"> If the problem occurs with PPI tables that have hash or join indexes, then change the value for MaxParseTreeSegs to this value and contact Teradata Support. 2,000 for byte-packed systems. 4,000 for byte-aligned systems. |
| | | Partitioning expressions are too complex. | Simplify the partitioning expressions or reduce their number. If possible, use RANGE_N with EACH clauses instead of CASE_N. |
| 3891 | Check if the partitioning CHECK constraint text size limit is exceeded. The partitioning CHECK constraint text is derived from the partitioning expression for the table or join index. | Text for the partitioning expressions in the PPI definition is too long. | Reduce the partitioning constraint text to 16,000 or fewer characters. 30 characters of the 16,000 character limit apply to the following constraint for single-level partitioning: CHECK (<i>partitioning_expression</i>) BETWEEN 1 AND 65,535. This defines the limit for the partitioning expression to be 16,000 - 30 = 15,970 characters. |
| 3930 | The dictionary cache is full. | Dictionary cache is too small. | Increase size of dictionary cache to 1 MB. Because the default size of the dictionary cache is 1 MB, the only reason you might need to do this is if the DBA has reduced the size of the dictionary cache to something less than its default. |

See Messages for additional information about these error messages.

The PPICacheThrP parameter of the DBS Control utility (for more information see *Teradata Vantage™ - Database Utilities*, B035-1102, *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142, and *Teradata Vantage™ - Database Design*, B035-1094) specifies the percentage value the system uses to calculate the cache threshold used in operations dealing with multiple partitions. The value is specified in units of 0.10 percent and can range between 0 and 500, inclusive.

PPICacheThrP also specifies the percentage value to use for calculating the number of memory segments that can be allocated to buffer the appending of column partition values to column partitions for column-partitioned tables.

The sum of the sizes of this memory minus some overhead divided by the size of a column partition context determines the number of available column partition contexts. If there are more column partitions in a target column-partitioned table than available column partition contexts, multiple passes over the source rows are required to process a set of column partitions where the number of column partitions in each set equals

the number of available column partition contexts. In this case, there is only one file context open, but each column partition context allocates buffers in memory.

The Optimizer also uses the DBS Control parameter PPICacheThrP to determine the number of available file contexts that can be used at a time to access a partitioned table.

| IF PPICacheThrP determines the number of available file contexts to be ... | THEN Vantage considers this many file contexts to be available ... |
|--|--|
| < 8 | 8 |
| > 256 | 256 |

Vantage uses the number of file contexts as the number of available column partition contexts for a column-partitioned table.

Note:

Vantage might associate a file context with each column partition context for some operations, and in other cases it might allocate a buffer with each column partition context.

Ideally, the number of column partition contexts should be at least equal to the number of column partitions that need to be accessed by a request. Otherwise, performance can degrade because not all of the needed column partitions can be read at one time.

Performance and memory usage can be impacted if PPICacheThrP is set too high, which can lead to memory thrashing or a system crash. At the same time, the benefits of partitioning can be lessened if the value for the DBS Control parameter PPICacheThrP is set unnecessarily low, causing performance to degrade significantly.

The default is expected to be applicable to most workloads, but you might need to adjust for the best balance.

The PPI Cache Threshold (PCT) is defined as follows for byte-packed format systems and byte-aligned format systems with file system cache per AMP values less than 100 MB.

$$PCT = \left(\sum \text{file_system_cache} \right) \times \left(\frac{PPICacheThrP}{1000} \right)$$

PCT is defined as follows for byte-aligned format systems with file system cache per AMP values greater than 100 MB.

$$PCT = \frac{100 \text{ MB} \times PPICacheThrP}{1000}$$

The default is 10, which represents 1.0 percent. *Never change this value without first performing a thorough analysis of the impact of the change on your query workloads.*

PPICacheThrP controls the memory usage of operations on partitioned tables. Larger values improve the performance of these operations, with the following exceptions:

- Data blocks are kept memory-resident.
If they must be swapped to disk, performance might begin to degrade.
- The number of contexts does not exceed the number of populated, non-eliminated partitions being processed.

In this case, increasing the value of PPICacheThrP does not improve performance because each partition has a context, and additional contexts would not be used.

Note:

For byte-aligned format systems only, the maximum partitioning cache that can be allocated is 100 MB regardless of the setting for PPICacheThrP.

The PCT value is used for the following operations on partitioned tables:

- Joins on a partitioned table.
- Aggregation of a partitioned table.

You should decrease PPICacheThrP if memory contention occurs during these types of operations.

Joins on a Partitioned Table

If a join is performed on a partitioned table or spool and the other table or spool is not partitioned, the maximum number of partitions processed at one time from the partitioned table or spool is equal to the following calculation.

$$\text{Maximum number of partitions processed} = \text{MAX}(\text{MIN}(\frac{\text{PCT}}{(\text{avg_data_block_size} + 12\text{K})}, 256), 8)$$

If the join is made on the primary index of 2 partitioned relations, then the maximum number of partitions processed at a time from the relations is calculated as follows.

$$\text{Maximum number of partitions processed} = \text{MAX}(\text{MIN}(f1, 256) + \text{MIN}(f2, 256), 16)$$

where:

$$\frac{(f1 \times db1) + (f2 \times db2)}{2} \leq \text{PCT}$$

For each partition that is being processed, 1 data block is always memory-resident.

The definitions for the variables in this equation are as follows.

| Equation element ... | Specifies the ... |
|----------------------|--|
| f1 | number of partitions to be processed at one time from the left relation in the join as determined by the Optimizer. |
| f2 | number of partitions to be processed at one time from the right relation in the join as determined by the Optimizer. |
| db1 | estimated average datablock size of the left relation in the join. |
| db2 | estimated average datablock size of the right relation in the join. |

Aggregation of a Partitioned Table

If the system aggregates on the primary index of a partitioned table, the maximum number of partitions processed at one time from the table is calculated as follows.

$$\text{Maximum number of partitions processed} = \text{MAX}(\text{MIN}(\frac{\text{PCT}}{(\text{estimated avg datablock size} + 12\text{K})}, 256), 8)$$

For each partition that is being processed, 1 data block is always memory-resident.

Restrictions and Limitations for Load Utilities and Partitioned Tables

Load utilities are supported on partitioned tables with the following restrictions and advisories.

- MultiLoad does not support loads into target tables having USIs. Many partitioned tables have USIs. An alternative is to FastLoad rows into an unpopulated staging table, then use either INSERT ... SELECT or MERGE with error logging to move the rows into the target table. See [CREATE ERROR TABLE](#) and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details.
You cannot use FastLoad to load rows into a column-partitioned table.
- MultiLoad IMPORT tasks require you to supply all values of the primary index column set and all values of the partitioning column set for deletes and updates.
- MultiLoad IMPORT tasks do not support updates of the partitioning column set.
- MultiLoad IMPORT tasks do not support primary index updates.
- FastLoad does not support loads into target tables having either USIs or NUSIs.
- FastLoad does not support loads into primary-indexed normalized target tables when a primary index column is also part of the ignore list for the NORMALIZE specification.
- MultiLoad and FastLoad do not support loading into target tables that are defined with either hash or join indexes.

- You should specify values for the partitioning column set when performing Teradata Parallel Data Pump deletes and updates to avoid lock contention problems that can degrade performance. For more information, see *Teradata® Parallel Data Pump Reference*, B035-3021.
- You should avoid updating primary index and partitioning columns with Teradata Parallel Data Pump to minimize performance degradation. For more information, see *Teradata® Parallel Data Pump Reference*, B035-3021.

Restrictions and Limitations for Archive/Recovery and Partitioned Tables

You cannot restore or copy an archive that contains partitioned tables to an earlier release of Vantage that does not support row partitioning or column partitioning.

If you restore or copy a partitioned table to another system, you should revalidate the table headers on the target system after the restore or copy operation completes.

Because many of the ALTER TABLE MODIFY operations change the version number of tables, you cannot perform the following operations after modifying primary index partitioning, uniqueness, or column set members.

- Cluster restores
- Single AMP restores
- Permanent journal rollforwards or rollbacks

If Archive/Recovery encounters such physical changes to a table during a rollforward or rollback operation, the operation stops and places an error message in the output listing.

The following table definition options and table definition-related operations performed by other statements do not invalidate future restorations of selected partitions for the targeted or referenced tables.

See [ALTER TABLE \(Basic Table Parameters\)](#) and [CREATE TABLE \(Table Options Clause\)](#) for more information about these options.

- Specifying table-level options that are not related to the semantic integrity of the table, including the following.
 - Fallback protection
 - Journaling attributes
 - Free space percentage
 - Data block size
- Specifying a partitioning expression. See [CREATE TABLE \(Index Definition Clause\)](#) for more information about this option.
- Specifying either column-level or table-level CHECK constraints. See [CREATE TABLE \(Column Definition Clause\)](#) for more information about these options.
- Collecting or dropping statistics on the table. See [COLLECT STATISTICS \(Optimizer Form\)](#) and [DROP STATISTICS \(Optimizer Form\)](#) for more information about collecting and dropping statistics.

- Adding comments about the table definition. See [COMMENT \(Comment Placing Form\)](#) for more information about adding comments about a table definition.
- Creating or dropping a hash index on the table. See [CREATE HASH INDEX](#) and [DROP HASH INDEX](#) for more information about creating and dropping hash indexes.
- Creating or dropping a join index on the table. See [CREATE JOIN INDEX](#) and [DROP JOIN INDEX](#) for more information about creating and dropping join indexes.
- Creating or dropping a secondary index on the table. See [CREATE INDEX](#) and [DROP INDEX](#) for more information about creating and dropping secondary indexes.
- Creating, replacing, or dropping a trigger on the table. See [CREATE TRIGGER/REPLACE TRIGGER](#) and [DROP MACRO](#) for more information about creating and dropping triggers.

When the target and source are different systems, you must repeat each of the previously listed operations on the affected tables of the target system to ensure that the 2 are kept in synchrony.

See [Rules for Retaining Eligibility for Restoring or Copying Selected Partitions](#) for information about modifications to a table definition that *do* make future restore or copy operations of selected partitions for targeted or referenced tables non-valid.

Secondary Indexes

You can define up to 32 secondary, hash, and join indexes (in any combination) for 1 table, and secondary indexes can be either unique (USI) or nonunique (NUSI). A multicolumn NUSI that specifies an ORDER BY clause counts as 2 consecutive indexes in this calculation.

This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.

UNIQUE and PRIMARY KEY constraints that do not define the primary index for a table also count against this total because they are defined internally as USIs for nontemporal tables. These constraints are defined internally as single-table join indexes, so any UNIQUE or PRIMARY KEY constraint counts equally against the maximum of 32 secondary, hash, or join indexes per table.

You cannot include BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, Period, or Geospatial columns in a secondary index column set.

You cannot include the system-derived PARTITION or PARTITION#L *n* columns in any secondary index definition.

You can include row-level security columns in a secondary index definition.

You can add secondary indexes to an existing table using the CREATE INDEX statement (see [CREATE INDEX](#)).

When a non-primary index uniqueness constraint is created for a table, Vantage implements it internally as a USI.

As a general guideline for decision support applications, whenever you define the primary index for a multiset table to be a NUPI, you should consider defining one of the following uniqueness constraints on its primary key or other alternate key to facilitate row access and joins.

- Unique secondary index
- UNIQUE NOT NULL constraint
- PRIMARY KEY constraint

You should always consider adding such constraints to your tables when they facilitate row access or joins. All manner of database constraints are often useful for query optimization, and the richer the constraint set specified for a database, the more opportunities exist to enhance query optimization.

The likely benefits of adding uniqueness constraints are not restricted to multiset NUPI tables. It is also true that if you create a SET NUPI table, the system performs duplicate row checks by default unless you place a uniqueness constraint on the table. Unique constraint enforcement is often less a less costly method of enforcing row uniqueness than system duplicate row checks.

Avoid defining a uniqueness constraint on the primary or alternate key of a multiset NUPI table just to enforce row uniqueness because MultiLoad and FastLoad do not support target tables with uniqueness constraints except those on the primary index. You can avoid the MultiLoad and FastLoad problems associated with indexes and triggers by using one of them (generally FastLoad) to load rows into a staging table that is otherwise identical with the target table, but has no constraints, triggers, or indexes defined on it. After loading the data into that table, you can then use either INSERT ... SELECT or MERGE with error logging to move it into the target table that has the desired constraints and indexes defined on it. See [CREATE ERROR TABLE](#) and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details.

Furthermore, USIs impose a performance cost because their index subtables must be maintained by the system each time the column set in the base table they reference is updated.

ALWAYS GENERATED ... NO CYCLE identity columns can be a better choice for the task of enforcing row uniqueness in multiset NUPI tables because they are supported by both MultiLoad and FastLoad. However, identity columns cannot be used to facilitate row access or joins.

You can also decide not to allow the system to enforce row uniqueness. Instead, you can use either of the following approaches.

- Enforce row uniqueness with application code.
- Avoid excluding duplicate rows by any arbitrary method, then perform periodic validation queries against your tables to detect duplicate rows.

This method assumes two things.

- Your application code tolerates duplicate rows.
- You know how to identify duplicate rows and how to eliminate any that are found in your tables.

There are several reasons to advise against approaches to duplicate row management that avoid using declarative constraints in order not to guarantee database integrity.

For example, if you define any non-primary index uniqueness constraints on a table, you must drop them all before you use MultiLoad or FastLoad to load rows into that table, then recreate them after the load operation completes.

If the MultiLoad or FastLoad operation loads any duplicate rows into the table, then you cannot recreate a uniqueness constraint on the table. You must first detect the duplicates and then remove them from the table.

Application-based methods of duplicate row management make several assumptions that are difficult to enforce. These assumptions include.

- The application code used to detect and reject duplicate rows is implemented identically across all applications in the enterprise.
- The application code used to detect and reject duplicate rows, even if universally implemented, is correct for all situations.
- All updates to the database are made by means of those applications. This assumption neglects the possibility of ad hoc interactive SQL updates that bypass the application-based duplicate row detection and rejection code.

USIs are generally *not* recommended for tables in workloads that support OLTP applications.

Each secondary index has an associated secondary index subtable. Each row in a secondary index subtable contains the indexed column value and 1 or more row IDs that point to the base table data rows containing that value (see *Teradata Vantage™ - Database Design*, B035-1094 for details of USI and NUPI subtable row layouts).

| Rows for this type of secondary index ... | Are stored in a subtable on ... |
|---|--|
| USI | <p>a different AMP from the rows they reference.</p> <p>This is generally, but not universally, true. It is possible, though unlikely, for a USI row to be stored on the same AMP as the base table row it references. For performance planning, it is best to think of secondary index access as always being a 2-AMP operation.</p> <p>An exception to this is the case where a USI is defined on the same column set as a partitioned primary index, but not all of the partitioning columns are in the primary index column set.</p> |
| NUSI | the same AMP as the rows they reference. |

Secondary indexes can greatly reduce table search time for so-called set selections (this term, which refers to selecting multiple rows, is technically a misnomer because a set can also contain 0 or 1 elements). However, update operations (including delete, insert, and update) on tables with secondary indexes require more I/O to process because of the required transient journaling, the possibility that secondary index subtables must be updated (if the indexed column set was updated), fallback table updates, and so on.

USI subtables are hashed and are very efficient for selecting a single row or a small number of rows.

NUSI bit mapping provides efficient retrieval when complex conditional expressions are applied to very large tables. Several weakly selective NUSIs can be overlapped and bit mapped to achieve greater selectivity.

The ALL option for NUSIs ignores the assigned case specificity for a column. This property enables a NUSI to do the following.

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

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 should always test the performance effects of a secondary index before implementing it.

For more information about secondary indexes, see *Teradata Vantage™ - Database Design*, B035-1094.

PRIMARY KEY and UNIQUE Constraints Versus Primary Indexes

You can define the primary index for a table using either a PRIMARY KEY or a UNIQUE constraints as the default primary index in a CREATE TABLE statement.

The following bullets list the rules for defining primary keys and UNIQUE constraints with respect to primary indexes:

- A table can have at most 1 primary key and need not have a primary index.
- If a table has a primary index, it can have only 1.
- You cannot define a primary index and a PRIMARY KEY or UNIQUE constraint on the same column set.

You can still define a relationship for referential integrity by referencing the UPI of a table even if no primary key is defined explicitly for that table because it is always valid to define a referential integrity relationship with any alternate key.

- If both a primary index and primary key are specified in a CREATE TABLE statement, then the primary index is the hashing index and the primary key is mapped to a unique secondary index by default.
- If a primary key is specified in a CREATE TABLE statement, but a primary index is not, then the system maps the primary key to a UPI by default.
- If neither primary index nor primary key is specified in a CREATE TABLE statement, then the system defines the first column that has a UNIQUE constraint as the UPI by default.
- If there is no PRIMARY INDEX, PRIMARY KEY, or UNIQUE constraint defined in a CREATE TABLE statement, and the PrimaryIndexDefault parameter is set to either D or P, then Vantage defines the first index-eligible column defined for the table to be its primary index.

The system defines this index as a NUPI by default except for the case of a single column table defined with the SET (no duplicate rows permitted) option, in which case the system defines it as a UPI.

- If there is no PRIMARY INDEX, PRIMARY KEY constraint, or UNIQUE constraint defined in a CREATE TABLE statement, and the PrimaryIndexDefault parameter is set to N, then Vantage creates the table with *no* primary index.
- Columns defined with either of the following constraints *cannot* be defined to be nullable.
 - PRIMARY KEY

- UNIQUE
- Columns defined with any of the following constraints can be defined as nullable:
 - PRIMARY INDEX
 - UNIQUE PRIMARY INDEX
 - INDEX
 - UNIQUE INDEX

You should declare the column set that constitutes these index types to be NOT NULL unless there is a compelling reason not to.

- You cannot define a PRIMARY KEY or UNIQUE constraint with the same column set as a secondary index defined on the same table.
- You cannot define a UNIQUE constraint explicitly on the same columns as a PRIMARY KEY constraint.

Nullable Columns Are Valid for Unique Indexes

While columns defined with PRIMARY KEY or UNIQUE constraints not only cannot be null, but must also be explicitly defined as NOT NULL, nullable columns *are* valid for UPI and NUPI column definitions. This is not good database design practice, and you should only permit nullable columns in index definitions under extraordinary conditions.

Note that these semantics demand that nulls in unique index columns must be treated as if they are equal to one another even though SQL nulls only represent an *absence* of value, which means they can neither be equated nor compared in any other way. Also note that all null primary index rows hash to the same AMP, so even a UPI can produce row distribution skew if you allow it to be null. For a description of other inconsistencies in the semantics of SQL nulls, see *Teradata Vantage™ - Database Design*, B035-1094.

| IF an index declared as UNIQUE or UNIQUE PRIMARY consists of ... | THEN ... |
|--|--|
| a single column | only 1 row can exist in the table with a null for that column. |
| multiple columns | the table cannot have more than 1 row in which all the values are identical because duplicate rows are not permitted when uniqueness is defined. |

See *Teradata Vantage™ - Database Design*, B035-1094 for additional information about nulls in database design, their potential for causing problems in various situations, and suggestions about how to avoid those problems.

Related Information

For information and usage suggestions for PPI, see *Partitioned Primary Index Usage Orange Book*, 541-0003869.

For information and usage suggestions for nonpartitioned NoPI tables see *No Primary Index (NoPI) Table User Guide Orange Book*, 541-0007565.

For important usage information about column-partitioned tables and join indexes, see:

- *Increased Partition Limit and Other Partitioning Enhancements Orange Book*, 541-0009027
- *Teradata® Columnar Primer Orange Book*, TDN0009884

See CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for information about the syntax used to create row-partitioned tables, nonpartitioned NoPI tables, and column-partitioned tables.

See [ALTER TABLE \(Basic Table Parameters\)](#) and ALTER TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for information about how to modify table indexes.

CREATE TABLE Global and Temporary

These topics provide supplemental usage information about the CREATE TABLE statement.

For CREATE TABLE syntax information and examples, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE TABLE (Global Temporary/Volatile Table Preservation Clause)

About Deleting or Preserving Global Temporary and Volatile Table Contents

This clause instructs the database whether to delete or preserve the contents of a global temporary or volatile table when a transaction completes.

DELETE is the default.

This option is valid for global temporary and volatile tables only.

Also see [Global Temporary Tables](#) and [Volatile Tables](#).

ON COMMIT DELETE/PRESERVE ROWS

The following table explains how session mode affects the timing of the ON COMMIT action.

| In this session mode ... | The ON COMMIT action is applied at this time ... |
|--------------------------|---|
| ANSI | when the application issues a COMMIT statement. |
| Teradata | <ul style="list-style-type: none"> The end of each request (implicit transaction). When the ET statement is processed (explicit transactions). <p>If you are using nested BT statements, then ON COMMIT is applied when the ET for the outermost BT is processed.</p> |

Related Information

For in-depth information about partitioned primary indexes, including usage suggestions, see *Partitioned Primary Index Usage Orange Book*, 541-0003869.

For information about the syntax used to create global temporary and volatile tables, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

For information about modifying the definition of a global temporary or volatile table, see [ALTER TABLE \(Basic Table Parameters\)](#) and ALTER TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE TABLE AS

These topics provide supplemental usage information about the CREATE TABLE statement.

For CREATE TABLE syntax information and examples, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE TABLE (AS Clause)

Required Privileges

The privileges required to execute CREATE TABLE with an AS clause are identical to those for a standard CREATE TABLE statement.

About Copying Table Definitions

You can create work or test tables that are identical to, or based on, base tables in a database. For example, you can use work tables with the same structure as base tables to speed up processing because it is faster to insert rows into a unpopulated work table and later merge the updates into a persistent base table than it is to update the base table directly one update at a time.

You can create test tables that are identical to some or all of the base tables used by live applications for prototyping new applications on a test system.

The AS clause option of CREATE TABLE provides a simple method for copying some or all definitions from an existing table to a new table.

The AS ... WITH NO DATA clause copies only the source table or query expression definitions to the new table, while the AS ... WITH DATA clause copies both the source table definitions and its data to the new table.

You can also append an AND STATISTICS option to a WITH [NO] DATA clause to copy any collected source table statistics or empty histograms to a target table. If you specify WITH NO DATA AND STATISTICS, the system sets up the appropriate statistical histograms in the dictionary, but does not populate them with source table statistics. This is referred to as *zeroed statistics*.

Copying Table Definitions

The AS clause copies definitions for an existing table or query expression to a new table.

The following table describes the various AS options.

| Use this option ... | To copy from an existing source table or query expression to a new target table ... |
|------------------------------------|--|
| AS ...WITH NO DATA | the specified column definitions only. |
| AS ... WITH DATA | the specified column definitions <i>and</i> the data associated with those columns. Note that you cannot create a global temporary table using the WITH DATA option. |
| AS ... WITH NO DATA AND STATISTICS | the specified column definitions and their statistical histograms. Data is not copied. |
| AS ... WITH DATA AND STATISTICS | the specified column definitions and their statistical histograms. Data is copied. Use count information is copied when USECOUNT is enabled for the target database. See BEGIN QUERY LOGGING . |

To copy only a subset of the source table and, optionally, its statistics, specify a subquery that selects only those columns desired in the target table. You can also use the subquery option to specify column expressions based on columns in the existing table to define column modifications for the target table. See [Using Subqueries To Customize An AS Clause](#) for further information.

Target Table Kind Defaults

If the source is a base table and you do not specify an explicit table kind, the table kind of the target table defaults to the source table kind.

Note that if you use the abbreviated CT ... AS syntax, you cannot specify a table kind. Because of this, the system automatically copies the table kind of the source table to the target table when you use the CT ... AS syntax.

If the source is a subquery, the system creates the target table using the default table kind for the session mode unless you specify an explicit table kind in the CREATE TABLE statement, as indicated in the following table.

| IF you copy to a target table using a subquery while in this session mode ... | THEN the target table has this table kind unless you explicitly specify something different ... |
|---|---|
| ANSI | MULTISET |
| Teradata | SET |

For example, if you copy a definition to a target table in ANSI session mode using a subquery, then the table kind of the target table defaults to MULTISET unless you explicitly specify the SET table kind in your CREATE TABLE ... AS ... statement.

Exclusions from CREATE TABLE ... AS Syntax

Column definitions *cannot* contain either of the following attributes.

- Data types

Note that column definitions can contain valid data type *attributes*.
- Referential integrity constraints

Columns cannot be defined as identity columns.

General Usage Rules

The following general rules apply to table definitions copied using an AS clause in CREATE TABLE.

- When you use a subquery as your source definition, if you do not explicitly specify a column set to define the primary index, primary key, or a unique constraint for the target table, then the first column defined for the table is selected to be its *nonunique* primary index by default (see [About Primary-Indexing, Row-Partitioning, Column-Partitioning, NoPI Tables, and Secondary Indexes](#)).
- When you specify a subquery as your source definition, a column that maps to an expression assumes the data type of the expression result.
- When you specify a subquery as your source definition and also specify the WITH DATA AND STATISTICS option, a special set of rules applies. See [Specific Rules For AS ... WITH DATA AND STATISTICS That Uses A Subquery To Define The Source Table Column And Index Set](#).
- When you specify an existing table (not as defined by a derived table subquery) as the source table definition in the AS clause, then the new table always assumes the following table-level characteristics of the source table.
 - Column structures, including columns with UDT, Period, ARRAY, and VARRAY types.
 - Fallback options
 - Journaling options (except when the new table is a global temporary or volatile table, in which case permanent journaling is *not* copied).
 - All defined indexes except hash and join indexes.

Note that if new indexes are defined for the target table, then that table does *not* assume any existing indexes.
- When you specify an existing table (that is not specified as a subquery) as the source table definition in the AS clause and also specify the WITH DATA AND STATISTICS option, a special set of rules applies.
- If a partitioned primary index definition is copied to a new target table and no index definitions are specified for the target table, then that table is partitioned in the same way as the source table.
- If a partitioned primary index definition is copied to a new target table, no index definitions are specified, and the CREATE TABLE statement for the target table renames the partitioning columns, then those columns are also renamed in the partitioning expression for the target table.
- Column definitions can contain any of the following attributes.
 - Column names

- Default values
- NOT NULL constraints
- CHECK constraints
- UNIQUE constraints
- Columns in the target table are in the same left-to-right order as they are defined in the source table or source subquery select list.

Usage Rules for Normalized Tables

With the exception of the following list, the rules for creating a normalized table using a CREATE TABLE statement and creating a normalized table using a CREATE TABLE ... AS statement are the same. For a complete list of those rules, see [Rules and Restrictions for the NORMALIZE Option](#).

Following is a list of exceptions to the rules that are specific to CREATE TABLE ... AS statements.

- You cannot specify the NORMALIZE option for the target table in a CREATE TABLE ... AS statement.
- A target table created with a CREATE TABLE ... AS statement has the same NORMALIZE definition as the source table if you do not declare any columns.
- If you create the target table using a subquery, the table does not inherit the NORMALIZE specification of the source table.

Attributes Not Copied To The Target Table

The following list describes attributes that are not copied to a target table from the source table or query expression using an AS clause with CREATE TABLE.

- The following source table attributes are not copied to the target table because they affect multiple tables.
 - REFERENCES constraints
 - Triggers
- When an existing source table (not a subquery) is specified in the AS clause and indexes are specified in the Index Definition clause, then source table indexes are *not* copied to the target table.
- When a subquery is specified in the AS clause, indexes from tables specified in the subquery are not copied. Any indexes you want for the target table must be specified explicitly. If you do not specify an explicit primary index, the system chooses a default primary index.
- When the source definition is taken from a query expression, the system does not carry the table kind over to the target table definition (see [Target Table Kind Defaults](#)).

Using Subqueries To Customize An AS Clause

Subqueries are a powerful means for customizing the definition and data for a target table by using a query expression to select source columns for the target table definition.

If you use a subquery to copy the source table columns and, optionally, their collected statistics, the system does not copy their attributes from the source table to the target table unless you also specify 1 or more column descriptors, in which case it copies only the column attributes you specify explicitly in the subquery to the target table.

If you do not specify any column descriptors, the system copies only the data types (and, optionally, the collected statistics) of the selected source columns or the resultant data types of any columns used in a subquery expression to the target table. It does not copy any attributes such as FORMAT, DEFAULT, COMPRESS, and so on unless you specify them explicitly.

A column descriptor is a target table column name followed by a specification of 1 or more column attributes. If you specify column descriptors in the definition for the target table, then the column attributes they specify replace any contradictory attributes of the selected source column.

The system also copies any source column attributes from the selected columns that are not specified explicitly in a subquery column descriptor to the target table. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for an example.

Because the source was derived from a subquery rather than from a base table definition:

- The table-level attributes of table t are not copied to table t1.
- The indexes defined for table t are not copied to table t1.
- Because no explicit primary index was specified, the system selected the first valid column in t1, column a, to be its nonunique primary index.

There are several workarounds for the limitations of the subquery form of CREATE TABLE ... AS.

- Use the subquery form, but specify the indexes and other table column descriptors explicitly for the target table.
- Use the non-subquery form of CREATE TABLE ... AS.
- Use the subquery form, then use ALTER TABLE. To add attributes to the new table, see [ALTER TABLE \(Basic Table Parameters\)](#).
- Either use the WITH DATA option for CREATE TABLE ... AS (see [Rules For AS ... WITH NO DATA That Uses A Subquery To Define The Source](#)) or create a new table, then populate it with the data from the original table using an INSERT ... SELECT statement. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Following are rules for using of subqueries expressed in an AS clause.

- Most valid subquery can be specified in the AS clause.
- Any restrictions on valid subqueries also apply for the subquery forms of the AS clause. For example, the following restrictions apply.
 - You cannot specify an ORDER BY clause.
 - The WHERE, HAVING, QUALIFY, or ON clause of any condition specified within a recursive view definition cannot specify a SAMPLE clause within a predicate subquery.
 - The following table lists the rules for AS and NAMED clauses.

| Options Specified | Description |
|---|---|
| Both AS and NAMED clauses are specified | The AS clause takes precedence and the column names for the target table are derived from the names it specifies. |
| No column names are specified | Names specified for expressions are used to name the derived columns of the target table. |

- The following restrictions apply to subqueries.

| Subquery Includes | Description |
|-------------------|--|
| Column names | Column names for the new table default to the existing names of the selected columns if you do not specify different names for them. |
| Join expression | Duplicate column names from the joined tables must be differentiated by fully qualified names. |

- The following copy traits are defined for the subquery forms of the AS clause.
 - Column-level attributes are not copied, except for data type.
 - Table-level attributes, including indexes, are not copied.
 - The following user or database default attributes are used for the target table unless you specify explicit values for them in the CREATE TABLE statement.

The standard user and database defaults are defined in the following table.

| Table Option | Permanent Table Defaults | Global Temporary Table Defaults | Volatile Table Defaults |
|--|--------------------------|---------------------------------|-------------------------|
| Fallback Note: You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback. | None | None | None |
| Permanent journaling | No Before No After | Not permitted | Not permitted |
| Transaction journaling | None | Log | Log |
| Referential integrity constraint | None | Not permitted | Not permitted |
| CHECK or BETWEEN constraints | None | None | Not permitted |
| COMPRESS column | None | None | Not permitted |
| DEFAULT and TITLE clauses | None | None | Not permitted |

| Table Option | Permanent Table Defaults | Global Temporary Table Defaults | Volatile Table Defaults |
|---------------------|--------------------------|---------------------------------|--------------------------|
| Named Index | None | None | Not permitted |
| FREESPACE or DBSIZE | User or Database default | User or Database default | User or Database default |
| ON COMMIT | Not applicable | Delete | Delete |

Exclusive AS ... WITH DATA Clause Rules

The following list of rules applies only to an AS ... WITH DATA clause. These rules do *not* affect an AS ... WITH NO DATA clause.

- You cannot create global temporary tables using the AS ... WITH DATA clause.
- You cannot specify referential integrity constraints for an AS ... WITH DATA clause because a forward reference on a table that does not exist might be defined, and such a definition blocks any inserts until you create the referenced table.

Instead, use the ALTER TABLE statement (see [ALTER TABLE \(Basic Table Parameters\)](#)) to add referential integrity constraints *after* you create the new table.

- An error that prevents a subquery result from being stored in the new table aborts the creation of the new table.

For example, you code a CREATE TABLE ... AS ... WITH DATA statement and specify a column constraint such as CHECK or NOT NULL. If a constraint causes a row in the subquery result to be rejected, then the table creation aborts and the system returns a constraint violation error to the requestor.

Comparing PARTITION Statistics Copied With COLLECT STATISTICS ... FROM *source_table* to PARTITION Statistics Copied With CREATE TABLE AS ... WITH DATA AND STATISTICS

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 statement are identical and the data is the same in both tables.

If you use a CREATE TABLE ... AS ... WITH DATA AND STATISTICS statement 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 statement 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 2 tables might appear to be identical from comparing their definitions using the output of SHOW TABLE statements on the tables.

All data in the columns whose statistics are copied using a COLLECT STATISTICS ... FROM *source_table* must be identical in both the source and target tables.

When there is a mismatch between the source and target tables, the request does not abort or return an error. Instead, it returns 1 of 2 possible warnings, depending on the nature of the mismatch.

| Warning message | Description |
|---|--|
| Statistics are not available in the source table | <ul style="list-style-type: none"> No statistics have been collected on the source table or No statistics have been collected in the source table on the specified columns in the target table. <p>This warning can be returned when statistics have been collected on the source table, but cannot be copied to the target table for some reason.</p> |
| Only some of the statistics were copied to the target table | <p>Not all of the statistics from the source table could be copied to the target table. The database does not inform you which statistics were or were not copied.</p> |

You do not obtain the same results from a COLLECT STATISTICS ... FROM *source_table* statement and a CREATE TABLE ... AS ... WITH DATA AND STATISTICS statement if the internal partition numbers are different. While the database copies the same statistics using both methods, in the COLLECT STATISTICS ... FROM *source_table* case, there are cases where the statistics are not valid for the data even though 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.

General Rules For CREATE TABLE AS ... WITH DATA AND STATISTICS

The following list of rules applies only to an AS ... WITH DATA AND STATISTICS clause. These rules do *not* affect an AS ... WITH NO DATA clause. See [General Rules For CREATE TABLE AS ... WITH NO DATA](#).

- If there are no columns or indexes in the target table for which statistics are eligible to be copied, the system returns a warning message to the requestor.
- If you specify an explicit index definition for the target table, then the system does not copy PARTITION statistics from the source table to the target table.

This is true for both single-column PARTITION statistics and for composite statistics on a column set that includes the system-derived PARTITION column.

- If no statistics have been collected on the specified source table column or index sets, the system ignores the AND STATISTICS option and returns a warning message to the requestor.
- If only a subset of the statistics from the source table are eligible to be copied to the columns and indexes of the target table, the system returns a warning message to the requestor.

- If the number of multicolumn statistics you specify to be copied to the target table exceeds the maximum number of multicolumn statistics allowed (the maximum number of multicolumn statistics that can be collected and maintained for a table is 32), then the system copies multicolumn statistics only up to the limit, does not copy the remainder of the multicolumn statistics to the target table, and reports a warning message to the requestor.
- If all columns in a MULTiset source table are nonunique, and if the target table is a SET table, then the system does not copy statistics to the target table. This is because of the possible violation of the rule of equal cardinalities in the source and target tables: if there are duplicate rows in the source table, the system eliminates them before copying to the target table, resulting in unequal cardinalities between the 2 tables.
- If the source table has *at least* 1 nonunique column and the target table is a SET table, then the system copies the statistics from the source table to the target table if all other rules are also obeyed.
- If you specify the NOT CASESPECIFIC attribute for any column in the target table definition, and it does not match the corresponding source table column attribute specification, the system does not copy the statistics for that column or index set because of the possible violation of the rule about not modifying the data in the target table.
- If all the columns in the source table are nonunique and you specify the NOT CASESPECIFIC attribute for any column in the target table definition that does not match the corresponding source table column attribute definition, then the system does not copy the statistics for that column or index because of the possible violation of the rule of equal cardinalities in the source and target tables.
- If you specify the UPPERCASE attribute for a column in the target table definition that does not match with the corresponding source table column attribute definition in the source table, then the system does not copy statistics for any column or index because of the possible violation of the rule about not modifying the data in the target table.
- Vantage copies single-column *index* statistics from the source table to the target table as single-column *column* statistics in the target table if no target table index is defined on the corresponding column set in the target table.
- Vantage copies single-column *column* statistics from the source table to the target table as single-column *index* statistics in the target table if an index is defined on that column in the target table.
- Vantage copies composite *index* statistics from the source table to the target table as multicolumn statistics if the source table index definition is not defined for the target table.
- Vantage copies multicolumn statistics from the source table to the target table as composite *index* statistics if the target has an index defined on that column set.

Specific Rules For CREATE TABLE AS ... WITH DATA AND STATISTICS That Does *Not* Use a Subquery To Define The Source Table Column And Index Set

If a CREATE TABLE ... AS statement does not specify a subquery to define the source table column and index sets to be copied, the following rules apply in addition to those stated in [General Rules For CREATE TABLE AS ... WITH DATA AND STATISTICS](#).

- The system copies all the available statistics on the source table for single columns to the target table.
- The system copies all the available composite statistics on the source table columns to the target table except for any multicolumn statistics that include the system-derived PARTITION column.
- If no index definitions are specified for the target table, the system copies all available statistics on the source table indexes to the target table. The term *zeroed statistics* refers to the condition in which the synopsis data structures, or histograms (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142), for the statistics on a column set or index have been constructed, but no statistics have been collected on the column set or index. A common example of this state is when you create a new table and then collect statistics on it when it does not yet contain any data rows.
- The following rules apply to copying PARTITION statistics from the source table to the target table.

| IF you ... | THEN the system ... |
|---|--|
| define any explicit indexes for the target table | does not copy any statistics that include the system-derived PARTITION column from the source table to the target table. |
| do not define any explicit indexes for the target table | copies any available single-column statistics on the system-derived PARTITION column or multicolumn statistics that include the system-derived PARTITION column from the source table to corresponding column set in the target table. |

Specific Rules For AS ... WITH DATA AND STATISTICS That Uses A Subquery To Define The Source Table Column And Index Set

If a CREATE TABLE ... AS statement specifies a subquery to define the source table column and index sets to be copied, the following rules apply in addition to those stated in [General Rules For CREATE TABLE AS ... WITH DATA AND STATISTICS](#).

- The system does not copy statistics if you specify multiple table definitions in a subquery because multiple table specifications are joins, which change the data for the target table. This violates the rule that source and target table cardinalities must be identical. It is possible to resolve to a single source table when join elimination is done for multiple tables specified in a query. The system also does not copy statistics in this situation, as the rule specifies.
- The system does not copy statistics if none of the columns specified in the subquery of the CREATE TABLE ... AS statement has a uniqueness constraint and the target is defined to be a SET table because this violates the rule that source and target table cardinalities must be identical.

However, if any column in the subquery is defined with a uniqueness constraint, then the system copies the statistics provided that all other eligibility requirements are met.

This is a specific restatement of the general rule stated in bullet number 7 under the topic [General Rules For CREATE TABLE AS ... WITH DATA AND STATISTICS](#).

- The system does not copy statistics for the system-derived column PARTITION to the target table when the source table is specified in a subquery. This is true both for single-column

PARTITION statistics and for multicolumn statistics where the column set includes the system-derived PARTITION column.

- The system does not copy statistics to target table columns when the source relation in a subquery is a complex view or complex derived table.
- The system does not copy statistics for any column or index defined for the target table if only a proper subset of the data is copied from the base table because this violates the rule that source and target table cardinalities must be identical.

For example, the system copies only partial data when you specify any of the following.

- Join conditions
- A DISTINCT operator
- A TOP *n* operator
- Aggregate functions
- OLAP functions
- A WHERE clause
- A GROUP BY clause
- A HAVING clause
- A QUALIFY clause
- Vantage does not copy statistics for any column or index defined for the target table if the source table data is modified in the target table definition.

Similarly, Vantage does not copy statistics for a composite index if the source table index data is changed when it is copied to the target table because this violates the rule about modifying target table data. An example of this would be the case where any of the index columns is specified with expressions in the subquery.

For example, data on a single column or index is modified if you specify a CASE expression or arithmetic expression in its target table definition.

- The system does not copy multicolumn statistics and composite index statistics if the left-to-right order of the columns specified in the select list are different from that of the left-to-right order of the source table columns. This is because the index or multicolumn values in the statistics are recorded in the field-id order of the columns in the set, so the values do not apply in the target table if the essential ordering of the underlying columns is not preserved.
- The system does not copy multicolumn statistics unless all of the component columns of the composite index or multicolumn set are copied to the target table.
- The system does not copy single column and single-column index statistics from a source table unless that column is also copied to the target table.

General Rules For CREATE TABLE AS ... WITH NO DATA

The following list of rules applies only to an AS ... WITH NO DATA clause. (see for the set of general rules that applies to the AS ... WITH DATA AND STATISTICS clause) [General Rules For CREATE TABLE AS ... WITH DATA AND STATISTICS](#)

- If there are no columns or indexes in the target table for which statistics are eligible to be copied, the system returns a warning message to the requestor.
- If you specify an explicit index definition for the target table, then the system does not copy PARTITION statistics from the source table to the target table.

This is true for both single-column PARTITION statistics and for composite statistics on a column set that includes the system-derived PARTITION column.

- If no statistics have been collected on the specified source table column or index sets, the system ignores the AND STATISTICS option and returns a warning message to the requestor.
- If only a subset of the statistics from the source table are eligible to be copied to the columns and indexes of the target table, the system returns a warning message to the requestor.
- If the number of multicolumn statistics you specify to be copied to the target table exceeds the maximum number of multicolumn statistics allowed (the maximum number of multicolumn statistics that can be collected and maintained for a table is 32), then the system copies multicolumn statistics only up to the limit, returns an error message to the requestor, and does not copy the remainder of the multicolumn statistics to the target table.
- If you specify WITH NO DATA for a volatile source table, the system returns an error.
- If all columns in a MULTISET source table are nonunique, and if the target table is a SET table, then the system copies zeroed statistics to the target table.
- The database copies zeroed statistics from all eligible source table columns and indexes if you specify the NO DATA options for permanent data tables, and global temporary tables.

The system also copies the current timestamp value for those source table statistics to the appropriate statistical histograms for the target table.

- The database copies zeroed statistics when you specify a NOT CASESPECIFIC or UPPERCASE attribute for any column in the target table definition whether it matches the corresponding source table column attribute specification or not.
- The database copies zeroed statistics for an index if you specify an index definition for the target table and it matches an existing index or multicolumn definition for the source table.
- The database copies eligible composite index statistics from the source table as zeroed multicolumn statistics to the target table if the index on which they were collected is not defined for the target table.
- The database copies eligible composite index statistics from the target table to the corresponding index on the target table when such an index is defined.
- The database copies eligible multicolumn statistics on the source table as zeroed composite index statistics to the target table if the target table has an index defined on the corresponding columns.
- If the target table *has* an index defined on the corresponding columns, the system copies eligible statistics on the source table to the target table as described by the following table.

| THIS type of source table statistics ... | IS copied to the target table as this type of statistics ... |
|--|--|
| single-column | single-column. |
| single-column index | single-column index. |

| THIS type of source table statistics ... | IS copied to the target table as this type of statistics ... |
|--|--|
| single-column PARTITION | single-column PARTITION. |
| multicolumn | multicolumn. |
| composite index | composite index. |
| composite PARTITION | composite PARTITION. |

- If the target table does *not* have an index defined on the corresponding columns, the system copies zeroed statistics on the source table to the target table as described by the following table.

| THIS type of source table statistics ... | IS copied to the target table as this type of zeroed statistics ... |
|--|---|
| single-column | single-column. |
| single-column index | single-column index. |
| single-column PARTITION | single-column. |
| multicolumn | multicolumn. |
| composite index | composite index. |
| composite PARTITION | multicolumn. |

- The database copies single-column statistics on the source table as zeroed single-column statistics to the target table only if that column is eligible for its statistics to be copied.

Rules For CREATE TABLE AS ... WITH NO DATA That Does *Not* Use a Subquery To Define The Source

In addition to the set of general rules provided in [General Rules For CREATE TABLE AS ... WITH NO DATA](#), the following list of specific rules applies only to an AS ... WITH NO DATA clause that does not specify a subquery.

- Vantage copies all the available single-column statistics on source table columns as zeroed statistics.
- Vantage copies all available multicolumn statistics except those multicolumn statistics that include the system-derived column PARTITION as multicolumn zeroed statistics.
- Vantage copies all available statistics from source table indexes to the target table as zeroed index statistics when you do not define any indexes for the target table.
- Vantage copies all available statistics on the system-derived PARTITION column and any multicolumn statistics that include the system-derived PARTITION columns in their column set as single-column and multicolumn zeroed statistics, respectively, from source table column sets to the target table as zeroed index statistics when you do not define any indexes for the target table.

Rules For AS ... WITH NO DATA That Uses A Subquery To Define The Source

In addition to the set of general rules provided in [General Rules For CREATE TABLE AS ... WITH NO DATA](#), the following list of specific rules applies only to an AS ... WITH NO DATA clause that *does* specify a subquery to define the source table for the copy operation.

- Vantage copies zeroed statistics if a set of the columns specified in the subquery do not have uniqueness constraints, and the target table is a SET table.
- Vantage copies zeroed statistics if any of the following clauses, operators, conditions, or function types are specified in the definition of the target table.
 - WHERE clause
 - GROUP BY clause
 - HAVING clause
 - QUALIFY clause
 - DISTINCT operator
 - TOP *n* operator
 - Inner join
 - Outer join
 - Aggregate function
 - OLAP function
- Vantage copies zeroed multicolumn statistics and zeroed index statistics even if the relative order of the index columns specified in the select list is different from that of the index column order in the source table.
- Vantage does *not* copy zeroed statistics if the subquery references more than 1 table.
- Vantage does *not* copy zeroed statistics for a column or index if that column or index is modified in the target table definition because this violates the rule against copying statistics when a column or index set is modified in any way in the target table definition.
- Vantage does *not* copy zeroed statistics for a composite index if any member of the index column set is modified in the target table definition because this violates the rule against copying statistics when a column or index set is modified in any way in the target table definition.

An example case of a modification to a column or index is specifying a CASE expression, arithmetic expression, or any conversion attribute.

- Vantage does *not* copy zeroed statistics, whether on a single column or on a composite column set, for the system-derived PARTITION column.
- Vantage does *not* copy zeroed statistics for target table columns that are formulated from complex views or complex derived tables in a subquery.
- Vantage does *not* copy multicolumn zeroed statistics or composite index zeroed statistics if any of the component columns of the multicolumn statistics or composite index statistics are not copied to the target table.

- Vantage does *not* copy single-column zeroed statistics or single-column index zeroed statistics if the column or single-column index is not copied to the target table.

Misleading Similarities of WITH DATA and WITH NO DATA Clauses

Because of the rules governing what is or is not copied to a target table during a CREATE TABLE ... AS operation, some statements that appear to be equivalent syntactically are not equivalent semantically.

This topic presents several scenarios to highlight some equivalent and non-equivalent coding.

Equivalence Set 1

The following CREATE TABLE ... AS ... WITH NO DATA statement is equivalent to the CREATE TABLE ... AS ... WITH DATA statement that follows it because both use the same subquery to define the columns in *target_table*.

Note that the second statement is coded with a WHERE clause that always tests FALSE. This clause ensures that only the column definitions for the subquery, and not its data, are copied to *target_table*.

```
CREATE TABLE target_table AS
  (SELECT *
   FROM source_table
  )
WITH NO DATA;
CREATE TABLE target_table AS
  (SELECT *
   FROM source_table
   WHERE 1=2
  )
WITH DATA;
```

Equivalence Set 2

The following CREATE TABLE ... AS ... WITH DATA statement is equivalent to the paired CREATE TABLE ... AS ... WITH NO DATA statement and INSERT ... SELECT statements that follow because the CREATE TABLE ... AS ... WITH NO DATA statement copies the exact table definition from *source_table* and the INSERT ... SELECT then populates *target_table* with all of the data from *source_table*.

```
CREATE TABLE target_table AS (
  SELECT *
  FROM source_table)
WITH DATA;
CREATE TABLE target_table AS source_table
WITH NO DATA;
```



```
INSERT target_table
  SELECT *
  FROM source_table;
```

Non-Equivalence Set 1

The first CREATE TABLE ... AS ... WITH NO DATA statement is *not* equivalent to the second because the subquery in the second statement uses the attribute defaults defined by the table options clause. You must name attributes explicitly to define them differently than their defaults. See [Using Subqueries To Customize An AS Clause](#).

```
CREATE TABLE target_table AS source_table
WITH NO DATA;
```

If source_table is MULTISET and you issue this CREATE TABLE statement in Teradata session mode, the table kind of target_table is also MULTISET.

```
CREATE TABLE target_table AS (
  SELECT *
  FROM source_table)
WITH NO DATA;
```

Under the same conditions cited for the first CREATE TABLE statement, this statement instead produces a table kind of SET for target_table because it uses a subquery for its source rather than a base table.

Non-Equivalence Set 2

This result parallels Non-Equivalence Set 1.

The first CREATE TABLE ... AS ... WITH DATA statement is *not* equivalent to the second because the subquery in the second statement uses defaults for attributes defined by the table options clause. You must name attributes explicitly to define them differently than their defaults (see [Using Subqueries To Customize An AS Clause](#)).

```
CREATE TABLE target_table AS source_table
WITH DATA;
CREATE TABLE target_table AS (
  SELECT *
  FROM source_table)
WITH DATA;
```

Related Information

For information about the FROM TABLE option, which is similar to the WITH DATA AND STATISTICS option for CREATE TABLE AS, see [COLLECT STATISTICS \(Optimizer Form\)](#).

For information about using CREATE TABLE (AS Clause) with ALTER TABLE to modify existing tables, see [ALTER TABLE \(Basic Table Parameters\)](#).

In *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144:

- For information about the syntax used to copy a table definition and its data and statistics, see CREATE TABLE.
- For syntax information related to using CREATE TABLE (AS Clause) to modify existing tables, see ALTER TABLE.

CREATE TABLE Queue

These topics provide supplemental usage information about the CREATE TABLE statement.

For CREATE TABLE syntax information and examples, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE TABLE (Queue Table Form)

Information About Table Characteristics Not Specific To Queue Tables

The information presented for this statement is specific to queue tables.

QUEUE Keyword

Each queue table you define must stipulate the keyword QUEUE as 1 of the CREATE TABLE options following the table name; otherwise, the table you define does not have the properties associated with queue tables and you cannot use consume mode when you select from it (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).

QITS Column

The first column defined for any queue table must be a Queue Insertion Time Stamp (QITS) column. Each queue table has only one QITS column, and it must be defined exactly as indicated with the following attributes.

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

where *QITS_column_name* indicates the name you specify for the QITS column.

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 is user-, not system-defined; however, the only significant option you have when defining the QITS column is its name.

A QITS column cannot be defined as any of the following:

- UNIQUE
- PRIMARY KEY
- Unique secondary index
- Identity column

A QITS column can be the NUPI for a table, but you should avoid following that practice. If you do not define an explicit primary index, primary key, or uniquely constrained column in the table, the QITS column becomes its primary index by default because it is the first column defined for the table.

You should select the primary index column set that best optimizes the performance of browse mode select operations on the table, just as you would for a non-queue table.

You might find it useful to define additional queue management columns for functions such as message identification or queue sequencing for your queue tables.

Function of Queue Tables

A queue table is a special database object: a persistent table used to handle queue-oriented data, such as event processing and asynchronous data loading applications, with subsequent complex processing of the buffered data load. The properties of queue tables are similar to those of ordinary base tables, with the additional unique property of behaving like an asynchronous first-in-first-out (FIFO) queue.

Row ordering in a queue table is not guaranteed to be truly FIFO for the following reasons.

- The system clocks on MPP system nodes are not synchronized.
- The QITS value for a row might be user-supplied or updated, either of which could change its position in the queue.
- A transaction rollback restores rows with their original QITS value, which might be an earlier value than rows that have already been consumed.
- Insert operations within the same multistatement request might be assigned the same QITS value.
- If Teradata Workload Manager is enabled, 1 or more rules might defer consume mode operations from running. As a general rule, you should not create rules that affect SELECT AND CONSUME operations because such workload restrictions can easily lead to queue table rows being processed in an order that differs significantly from a “true” FIFO ordering.

You can think of a queue table as a regular table that also has a memory-resident cache associated with it that tracks the FIFO queue ordering of its rows (see [Queue Table Cache](#)). Additionally, consumed rows are retrieved and deleted from the database simultaneously, which ensures that no row can be processed more than once.

Because most, if not all, rows for a given queue table are memory-resident on a PE, they are processed similarly to primary index operations made against non-queue tables, which are single-AMP operations applied with a row-hash WRITE lock on the row.

An ideal queue table has the following characteristics.

- Low cardinality (implying that its rows are consumed at roughly the same rate as they are inserted).
- Infrequent UPDATE operations to its rows.
- Infrequent DELETE operations on its rows.

Unlike standard persistent table definitions, a queue table definition must always contain a user-defined insertion timestamp (QITS) as the first column of the table. The QITS column contains the time at which the row was inserted into the queue table, which the system then uses to approximate FIFO ordering. Even

though the QITS value might not be unique, because timestamp values can repeat or be updated, the database always ensures the uniqueness of the row in the queue.

Queue tables provide several advantages that enable them to handle event-related data. Among the benefits provided by queue tables to database application developers are the following.

- Enabling superior implementation of queue-oriented applications, such as message- or workflow-based models.
- Enabling internally- or externally-generated queries to wait actively for the appearance of event data in a queue without having to poll the database periodically to detect the presence of event-related data.
- Supporting asynchronous loading and processing program structures by which applications can insert event-related data into queue tables, which can then buffer that data until it can be processed by user-developed procedures that perform either complex analysis themselves or that insert the data into other processing queues for later analysis.
- Supporting the ability to develop interfaces between the database management system and third party message-based Enterprise Application Integration (EAI) applications. For example, you can write external adapters to insert data into queue tables by invoking stored procedures, macros, or SQL INSERT statements. Similarly, you can write external adapters to extract data from queue tables for insertion into third party EAI application queues.

The following applications are some of the obvious ways that queue tables support enhanced methods of building applications to analyze event-related data.

- Event alerts.

When an event is detected by application code running in the database management system (such as a stored procedure), it can, for example, insert data from that incident into an event queue table by means of a trigger. An external event-processing application could then extract events from the database by submitting a `SELECT AND CONSUME TOP 1` statement, which then waits for data to be inserted into a queue table. When data arrives at the queue, the waiting `SELECT AND CONSUME TOP 1` statement returns a result to the external application, which then processes the data further.

The external application might then loop and submit another `SELECT AND CONSUME TOP 1` statement to wait for further event data to be inserted into the queue table. This functionality eliminates the need for the polling loops required by applications, based on non-queue tables, that must blindly and repeatedly submit `SELECT` statements while waiting for an event to occur.

- Data loading with asynchronous processing.

A typical example of such data loading coupled with asynchronous processing might be data being loaded into a queue table from an external messaging source.

Each individual piece of data might require significant processing. Rather than processing the incoming data immediately upon its arrival in the database, the system can instead buffer it in a queue table. The newly inserted data might then be processed by a group of stored procedures coded with `SELECT AND CONSUME TOP 1` statements that wait for data to be inserted into the queue table.

These stored procedure groups would share the labor of processing the queue table data, with each piece of arriving event data being parceled to an available stored procedure for processing.

This technique smooths resource consumption in response to varying data loading inflow rates. By controlling the number of stored procedures that are available to process queue table data, you can manage the maximum rate of resource consumption as well as the maximum throughput rate.

For example, during a short epoch having a high inflow rate, the system can initially buffer the data in a queue table. When the inflow rate later falls below a user-determined critical level, waiting stored procedures can then process the buffered queue table data.

- Schedule changes.

When transportation carriers need to make an unanticipated schedule change, additional entities, such as booking and call center systems, need to be notified to handle the implications of the change.

For example, upon notification of the schedule change, the booking system can begin to rebook all affected transactions.

The front line for notifying and rebooking customers is the call center. Call centers work from a queue of customers, in this case a queue of customers who need to be notified of the schedule change. Calls can be prioritized by applying business rules that consider a number of factors, such as whether the change involves a long or short haul, how significant the change is, the length of the delay caused by the schedule change, and how valuable the customer is to the business.

The input of each of these systems to the analysis can be modeled as a queue table. For example, messages in 1 queue table might represent schedule changes. Stored procedures reading that queue table could insert messages into the queue tables for the booking and call center systems, and so on. Call center agents could then be assigned to customers who need to be notified of a scheduling change by an application that consumes data from the call center queue table.

You can create error tables to track batch insert and update errors on a queue table. See [CREATE ERROR TABLE](#) for further information about error tables. Also see INSERT/INSERT ... SELECT and MERGE in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for information about how to specify that you want to track batch insert and update errors for those DML requests.

Queue Table Cache

The database maintains a cache that holds row information for a number of unconsumed rows for each queue table. The system creates this queue table cache, which resides in its own Dispatcher partition, during system startup. There can be a queue table cache task on each PE in your system.

Cache row entries are shared by all queue tables that hash to that PE. Each queue table row entry is a pair of QITS and rowID values for each row to be consumed from the queue, sorted in QITS value order. The entry for a given queue table exists only in 1 queue table cache on the system. The determination of which system PE is assigned to an active queue table is made within the Dispatcher (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142) partition by hashing its table ID using the following algorithm.

$$\text{queue_table_cache_PE_number} = \text{table_ID}[1] \text{MOD}(\text{number_system_PEs})$$

where:

| Syntax element ... | Specifies the ... |
|---|---|
| <i>queue_table_cache_ PE_number</i> | position number within the configuration map array of the PE containing the queue table cache to which the given queue table entry is assigned. |
| tableID[1] | numeric value of the second word of the double word tableID value (where the first word is notated as tableID[0]). |
| MOD | modulo function. |
| <i>number_system_PEs</i> | number of PEs in the system configuration. |

For example, suppose a system has 6 online PEs and the value for tableID[1]=34.

$$\text{queue_table_cache_PE_number} = 34 \text{ MOD}(6) = 4$$

This value points to the fifth position in the system-maintained configuration map array of online PEs, which begins its numbering at position 0. As a result of the calculation, the system assigns entries for this queue table to the cache on the fifth online PE listed in the system configuration map. If that PE goes offline, then the system reassigns the queue table to the first online PE in the configuration map.

During startup, the database allocates 64KB to the queue table cache on each PE and increases its size dynamically in 64KB increments to a maximum of 1MB per PE as required. The system initializes all the fields in the cache during startup and allocates space for 100 table entries. As queue tables are activated, they populate these slots beginning with the lowest numbered slot and proceeding upward from there. When the maximum cache size is reached, the system flushes it, either partially or entirely, depending on the number of bytes that must be inserted into the cache.

Assume the following definitions.

TotalRow Count = the number of unconsumed rows in a queue table on the AMPs.

RowCount = the number of rows in the queue table cache on a PE.

Each queue table cache entry has a maximum of approximately 2,000 row entries. As with other system maxima, this number refers to an isolated case. The actual limit on the number of row entries is determined by the size of the queue table cache. For example, 1 table can have 10 row entries, another can have 100, another can have 1,000, and so on until the limit of 1 MB is reached, at which point the system begins to flush entries from the cache. But no individual table can have more than approximately 2,000 row entries in the cache at a time.

When the maximum is reached, the system increments only the TotalRowCount value and does not add a new row entry to the cache.

| IF the value for ... | THEN ... |
|------------------------------------|--|
| TotalRowCount > RowCount | there are more rows in the queue table than there are rows in the queue table cache. |
| RowCount = 0 AND TotalRowCount > 0 | the system triggers a row collection operation to repopulate the queue table cache. |

Note that there are three events that can trigger a partial or full purge of row entries for a table from the cache without their being consumed.

- Exceeding the maximum 100 queue table entry slots.

This event causes a partial purge of row entries for the subject table. The system purges only as many cached row entries as are necessary to make room for the new entries. Note that this event can occur even if the total cache size is less than 1 MB.

- Exceeding the maximum 1 MB (approximately 20,000 row entries) queue table cache size limit.

This event causes a partial purge of row entries for the subject table. The system purges only as many cached row entries as are necessary to make room for the new entries. Note that this event can occur even if fewer than 100 queue table entry slots have been used.

- An UPDATE, DELETE, or MERGE operation on a queue table.

This event category purges all entries for the subject table from the cache.

Insertion of a table entry into 1 of the 100 table entry slots in the cache is triggered by either of the following events.

- An insert operation into the table.
- A consume operation from the table.

When the cache reaches its maximum limits of either 100 queue table entries or 20,000 total row entries for that PE, the next incoming request triggers the flush.

The process is outlined in the following table.

1. The system attempts to flush any queue table entries marked as spoiled.
 - If there are no spoiled queue table entries, then the database checks to see if there are any pending in-queue requests or if it is in the process of collecting rows.

If the most recent entry has pending in-queue requests or is in the process of collecting rows, then the database continues the search for an entry to flush beginning with the next lowest entry in the list.

If the most recent entry has no pending in-queue requests or is in the process of collecting rows, then the database flushes it from the cache.

Because it is possible to control the ordering of queue entries by inserting rows with user-selected QITS values in place of system-determined default QITS values, the entries that are flushed might not actually be those most recently added to the queue. See [Ordering Queue Table Rows](#).
 - If there are spoiled queue table entries, then the database flushes them.
2. If no qualifying entry can be found, the system returns an error to the requestor.

When the queue table cache reaches its maximum size of 1 MB (approximately 20,000 row entries), the next incoming request triggers a flush.

The process is outlined in the following sequence of events.

1. The system attempts to flush any queue table entries marked as spoiled.

| IF there are ... | THEN the database ... |
|--------------------------------|--|
| no spoiled queue table entries | <p>does the following.</p> <p>Checks each entry to determine if there are any pending in-queue requests or if it is in the process of collecting rows.</p> <p>If so, the entry is not eligible to be purged from the cache.</p> <p>Retrieves the cardinality of each qualifying queue table entry.</p> <p>Entries are ranked by their cardinality, with the largest entry being targeted for the first purge attempts.</p> <p>Computes the quantity of space required to insert the incoming request into the cache.</p> <p>Beginning with the most recently added rows in the first qualifying entry, based on their timestamp value, purges enough rows to reclaim the space needed to insert the incoming request into the queue table cache.</p> <p>Because it is possible to control the ordering of queue entries by means of inserting rows with user-selected QITS values in place of system-determined default QITS values (see Ordering Queue Table Rows), the entries that are flushed might not actually be those most recently added to the queue.</p> <p>This process continues until enough cache space is freed to accommodate the incoming request.</p> |
| spoiled queue table entries | flushes them. |

- If no qualifying entry can be found, the system aborts the transaction and returns an error to the requestor.

Identifying Queue Tables

You can use the *DBC.TablesV* view to identify and locate queue tables.

The following query reports the names and containing databases for all the queue tables in the system.

```
SELECT databasename, tablename
FROM DBC.tablesV
WHERE queueflag = 'y';
```

See *Teradata Vantage™ - Data Dictionary*, B035-1092 for more information about the *DBC.TablesV* view.

SQL Operations on Queue Tables

SQL interfaces provide support for FIFO push, pop, and peek operations on queue tables as indicated in the following table.

| THIS FIFO operation ... | Is supported by this SQL statement ... |
|-------------------------|--|
| push | INSERT. |

| THIS FIFO operation ... | Is supported by this SQL statement ... |
|-------------------------|--|
| pop | SELECT AND CONSUME TOP 1. |
| peek | SELECT. |

The different operative modes performed by the two forms of the SELECT statement are referred to as consume and browse modes, respectively. Note that you cannot use either the PERCENT or WITH TIES options with consume mode SELECT operations on a queue table. Additionally, you must always specify 1, and only 1, as the value for *n* in the TOP *n* specification. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

| This form of SELECT ... | Performs this function on a queue table ... |
|--------------------------|---|
| SELECT AND CONSUME TOP 1 | consume. |
| SELECT | browse. |

Note that consume mode rows are FIFO-ordered unless otherwise explicitly ordered (see [Ordering Queue Table Rows](#)) and browse mode rows use the standard SQL ordering mechanisms such as an ORDER BY clause.

Vantage supports the following DML statements for embedded SQL processing of queue tables.

- DELETE
- INSERT
- MERGE
- SELECT
- SELECT AND CONSUME
- UPDATE

Note that SELECT AND CONSUME is *not* supported for positioned cursors. You cannot use consume mode SELECT statements in ANSI session mode, because all ANSI mode cursors are, by default, positioned cursors. See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 and *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446 for further information.

Vantage provides queue table DML support for all the following forms of embedded SQL.

- Static
- Dynamic
- Static cursor
- Dynamic cursor

As a general rule, you should not perform DELETE, MERGE, and UPDATE statements frequently on a queue table because these operations spoil the FIFO cache for the entries for that table. As a result, Vantage must perform a full-table scan the next time it accesses the FIFO cache to rebuild it.

Instead, you should reserve these statements for exception handling.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 and *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for more detailed information about using these statements with queue tables.

Because queue tables are persistent database objects in the database, the basic Teradata features for transactions are provided for them. Like other database transactions, queue table transactions have the ACID properties of atomicity, consistency, isolation, and durability. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for further information about Teradata transactions.

Consume Mode Transactions

A consume mode transaction enters a delayed, or waiting, select state whenever a SELECT AND CONSUME operation fails to find rows in the specified queue table. An INSERT operation to that queue table wakes up the transaction.

When you retrieve a row in consume mode, the system then deletes it from the table.

A typical consume mode process might include the following.

1. A requestor issues a SELECT AND CONSUME TOP 1 statement against an unpopulated queue table.
2. The request accesses the FIFO cache and finds no rows.
3. The system puts the transaction containing the unfulfilled request into a delayed state until the status of the queue table changes.

No more than 24 transactions per PE can concurrently be in a delayed state. When this number is exceeded, the system returns an error to the requestor.

4. A requestor issues an INSERT operation on the queue table previously specified by the delayed transaction.
5. The act of inserting a row set into the queue table sends a message to the delayed transaction to awaken it. Delayed transactions can also be awakened in either of the following ways.
 - Through direct intervention by issuing an ABORT statement against the transaction
 - Through indirect intervention by dropping the queue table while a delayed transaction is pending against it.
6. The awakened transaction performs a SELECT AND CONSUME TOP 1 operation against the newly populated queue table.

Note that a SELECT AND CONSUME statement can consume any rows inserted into a queue table within the same transaction.

7. After the select request has been satisfied, the system consumes (deletes) the selected row set from the queue table.

You should position your SELECT AND CONSUME TOP 1 statements as early as possible within a transaction to avoid conflicts with other database resources. This is to minimize the likelihood of a situation where the SELECT AND CONSUME TOP 1 statement would enter a delayed state while holding locks on resources that might be needed by other requests.

The only lock assignment available for a SELECT AND CONSUME TOP 1 operation is a severity of WRITE applied at the row hash level.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for further information about how the INSERT, SELECT, and SELECT AND CONSUME statements process event processing-related data.

See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for further information about management of queue table transactions by the database.

Performance Issues for Queue Tables

The following list of issues applies to the performance-related aspects of queue tables.

- The general purpose of queue tables is to facilitate the ability of some applications to interface with asynchronous tasks that occur outside the database.

The expected behavior for a queue table is that the producers of queue table rows are working at approximately the same speed as their consumers. Rows are consumed as quickly as they are inserted into the table.

An application that would typically populate a queue table with millions of rows before consuming any of them would not see any performance gain over using a non-queue table, so is not a good candidate for using the feature.

Asynchronous events *within* the database can also use queue tables. For example, suppose you have defined a trigger on an inventory table that checks for the quantity of an item falling below some threshold value. When the threshold is reached, the trigger fires and inserts a row into a queue table that is then processed by an application that reorders the low quantity item.

- The performance of a consume mode SELECT operation is very similar to that of selecting a single row from a non-queue table using a USI.

The performance is more expensive than selecting a single row from a non-queue table using a primary index.

- You should position your SELECT AND CONSUME TOP 1 statements as early as possible within a transaction to avoid conflicts with other database resources. This is to minimize the likelihood of a situation where the SELECT AND CONSUME TOP 1 statement would enter a delayed state while holding locks on resources that might be needed by other requests.
- You should avoid the following programming practices when issuing SELECT AND CONSUME statements.
 - Coding *any* SELECT AND CONSUME statements within explicit transactions that might exert and hold a large number of locks on database objects.
 - Coding large numbers of SELECT AND CONSUME statements within a transaction, especially if there are also DELETE and UPDATE operations made on the same queue table as SELECT AND CONSUME statements.

Each time the system performs a DELETE, MERGE, or UPDATE operation on a queue table, the FIFO cache for that table is spoiled. The next INSERT or SELECT AND CONSUME

statement performed on the table initiates a full-table scan to rebuild the FIFO cache, which has a performance impact.

- When the number of queue table rows on a single PE frequently exceeds the size of the queue table cache, which is approximately 2,000 row entries per table or 20,000 row entries per PE (the definitive limit is 1 MB of entries), then the system must perform full-table scans more frequently to refresh the cache. Such states also increase the likelihood of errors related to lock-oriented transaction aborts and queue table cache overflow being returned to the requestor.
- An INSERT operation into a queue table does not affect response time when the system is not CPU-bound.
- An INSERT operation into a queue table is more expensive than an insert into a non-queue table because it forces an update of the FIFO cache on the affected PE.
- You should restrict DELETE, MERGE, or UPDATE operations on queue tables to exceptional conditions because of the negative effect on performance. The critical factor is not how many such operations you code, but how frequently those operations are performed. You can ignore this admonition if, for example, you run an application that performs many DELETE, MERGE, or UPDATE operations only under rarely occurring, exceptional conditions.

Otherwise, because of the likely performance deficits that result, you should code DELETE, MERGE, and UPDATE operations only sparingly, and these should never be frequently performed operations.

UPDATE, MERGE, and DELETE operations on a queue table are more costly than the same operations performed against a non-queue table because each such operation forces a full-table scan to rebuild the FIFO cache on the affected PE.

- Vantage uses the Teradata dynamic workload management software to manage all deferred requests (transactions in a delayed state) against a queue table. The Teradata dynamic workload management client application software does not need to be enabled to be used by the system to manage delayed consume mode requests

As a result, you should optimize your respective use of the 2 features because a large number of deferred requests against a queue table can have a negative effect on the ability of the Teradata dynamic workload management software to manage not just delayed consume mode queries, but *all* queries, optimally.

- The queue table FIFO cache on each PE supports a maximum of 100 queue tables.

When the number of active queue tables in the cache exceeds 100, the system performs full-table scans on all the tables in the cache and initiates a purge of the cache by taking one of the following actions, beginning the purge by first attempting the cache purge using the first method listed, then proceeding to the second if the first is either not applicable or exhausted.

Swap out a queue table that has been spoiled. For example, if a queue table has had a delete operation performed against it, it is a candidate for purge from the FIFO cache.

Purge an inactive queue table from the FIFO cache.

See [Queue Table Cache](#) for details.

- The queue table FIFO cache on each PE supports approximately 20,000 queue table row entries.

When the number of row entries in a the FIFO queue table cache exceeds the limit of roughly 20,000 row entries, the system purges the most recent entries from the longest queue to make room.

Because it is possible to control the ordering of queue entries by means of inserting rows with user-selected QITS values in place of system-determined default QITS values (see [Ordering Queue Table Rows](#)), the entries that are flushed might not actually be those most recently added to the queue.

For more information, see [Queue Table Cache](#).

- Queue tables that hash to the same PE FIFO cache share the same pool of row entries.
- The ideal number of active queue tables per PE is 1.

To optimize the distribution of your queue tables across the PEs, consider creating them all at the same time.

- If you use a client application that is enabled for iterated request processing to insert rows into a queue table, take care to keep the number of records packed into the USING data buffer sent with the SQL request low.

For example, if you use the BTEQ .IMPORT command to insert rows into a queue table, keep the packing density of SQL operations per transaction less than 5 using the BTEQ .SET PACK command.

See *Basic Teradata® Query Reference*, B035-2414. This ensures optimal performance by minimizing deadlocks and the retryable errors they can cause. Setting the .SET PACK command argument to a value greater than 1 enables iterated request processing, which invokes several restrictions on the DML USING clause. For information about these restrictions, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

By keeping the number of SQL operations per transaction low, you minimize the number of row hash-level WRITE locks placed on the table during the insert operations performed by the .IMPORT command. See [Queue Table Cache](#).

If you know that you are not going to perform any update or delete operations on the queue table while inserting rows into it, and if the table has been consumed or inserted into since the last reset of the queue table cache, then packing density is not an issue.

Restrictions on Queue Tables

There are a number of restrictions on queue tables and their use, as detailed in the following list.

- Queue tables cannot be temporal tables.
- The table definition DDL for a queue table must always specify the keyword QUEUE immediately following the specification for SET or MULTiset (see [QUEUE Keyword](#)).
- The first column defined for a queue table must be a user-defined QITS column (see [QITS Column](#)).
- You cannot drop the QITS column from a queue table (see [ALTER TABLE \(Basic Table Parameters\)](#)).
- You cannot define the QITS column to be any of the following.
 - UNIQUE PRIMARY INDEX
 - UNIQUE
 - PRIMARY KEY

- Unique secondary index
- Identity column

For more information about the uses of, and restrictions on, identity columns, see [Identity Columns](#) and [Rules for Specifying Identity Columns](#).

- You cannot define a queue table as either a nonpartitioned NoPI table or as a column-partitioned table. See [Nonpartitioned NoPI Tables](#) and [Column-Partitioned Tables](#).
- You cannot define a queue table with either of the following referential integrity clauses.
 - FOREIGN KEY ... REFERENCES
 - REFERENCES
- You cannot reference queue table columns from a FOREIGN KEY ... REFERENCES or REFERENCES clause in the definition of a different, non-queue, table.
- All other column- and table-level constraint clauses are valid within a queue table definition with the exception of UNIQUE and PRIMARY KEY constraints not being valid for the QITS column.
- You cannot define any column of a queue table with a BLOB, CLOB, Geospatial, JSON, XML, ARRAY, or VARRAY data type. For information about BLOB, CLOB, ARRAY, and VARRAY data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143. For information about Geospatial data types, see *Teradata Vantage™ - Geospatial Data Types*, B035-1181. For information about the JSON data type, see *Teradata Vantage™ - JSON Data Type*, B035-1150. For information about the XML data type, see *Teradata Vantage™ - XML Data Type*, B035-1140.
- Queue tables cannot also be global temporary tables.
- Queue tables cannot also be volatile tables.
- You cannot specify permanent journals for queue tables.
- Because queue tables cannot be global temporary or volatile tables, the following restrictions also apply to them.
 - You cannot log their change images to a transaction journal.
You cannot specify the LOG option for queue tables. See [LOG and NO LOG](#).
 - You cannot preserve their contents after a transaction completes.
You cannot specify an ON COMMIT clause for queue tables. See [ON COMMIT DELETE/PRESERVE ROWS](#).
- You cannot use the copy table syntax to copy the definition of a queue table. Neither the source, including a table referenced in a subquery, nor the target table of a CREATE TABLE AS clause can be a queue table. See [CREATE TABLE \(AS Clause\)](#).
- You cannot define queue tables with a partitioned primary index.
- Queue tables are not supported for the following load and export utilities.
 - MultiLoad
 - FastLoad
 - FastExport

- If you change the *tdlocaledef.txt* file and issue a `tpareset` command, the new format string settings affect only those tables that are created *after* the reset. Existing table columns continue to use the extant format string in *DBC.TVFields* unless you submit an `ALTER TABLE` statement to change it. See [ALTER TABLE \(Basic Table Parameters\)](#).

Queue Table-Related Restrictions on Other SQL Statements

The following queue table-related restrictions apply to other SQL statements.

| You cannot ... | In this statement ... |
|-----------------------------------|--|
| reference a queue table | <ul style="list-style-type: none"> CREATE HASH INDEX CREATE JOIN INDEX CREATE VIEW (both forms) |
| create a trigger on a queue table | CREATE TRIGGER |

Browsing a Queue Table

To browse a queue table is to select from it as you would a non-queue table.

The following list of queries provides some peeks at a queue without consuming any rows.

- You can determine the queue depth of a queue table by using the `COUNT(*)` aggregate function, as follows.

```
SELECT COUNT(*)
FROM myqueue;
```

If the depth is zero, then the system places a consume mode `SELECT` statement into a delayed state.

- You can peek at the next row set that would be consumed from the queue table using the following query.

```
SELECT *
FROM shoppingcart
WHERE qits = (SELECT MIN(qits)
              FROM shoppingcart);
```

- The following query selects only the next ten rows to be consumed.

```
SELECT TOP 10 *
FROM myqueue
ORDER BY QITS;
```

- The following query, a browse mode `SELECT` statement, returns the entire queue in FIFO order.


```
SELECT *
FROM myqueue
ORDER BY myqueue_qits;
```

- The duration of a queue table measures how old the top queue table row is.

To determine the duration of a queue table, you can execute the following statement:

```
SELECT TOP 1 CURRENT_TIMESTAMP, qits, (CURRENT_TIMESTAMP - qits)      DAY(4)
TO SECOND AS queue_duration
FROM shoppingcart
ORDER BY qits;
```

This query generates a report similar to the following:

```
*** Query completed. One row found. 3 columns returned.
Current Timestamp (6)  2004-05-19 14:30:01.420000+00:00
Q Insertion Time      2004-05-18 08:44:19.460000
queue_duration        1 05:45:41.960000
```

Rules for Consuming Rows From a Queue

The following rules and recommendations apply to consuming rows from a queue table (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for a complete description of the rules and limits for consuming queue table rows).

- To consume rows from a queue table, specify the **AND CONSUME TOP 1** keywords in your **SELECT** statement. 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.

The following example consumes a row from a queue table named *shopping_cart*.

```
SELECT AND CONSUME TOP 1 *
FROM shopping_cart;
```

- You can also consume rows from stored procedures and embedded SQL applications that use the **INTO** clause to assign the values from a row to host or local variables.
- You should place any action taken based on consuming a row from a queue table in the same transaction as the consume mode **SELECT** operation on that same queue table. This ensures that both the row consumption and the action taken on that queue table are committed together, so no row or action for that queue table is lost.

If no action is to be taken, then you should isolate any **SELECT AND CONSUME** statement as the only request in a transaction. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

- A multistatement request should contain only 1 consume mode SELECT statement.
- A trigger can call a stored procedure that consumes a row from a queue table.

Restrictions on Consume Mode SELECT Statements

The following restrictions apply to SELECT AND CONSUME TOP 1 statements. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

- You cannot join queue tables with other tables in a SELECT AND CONSUME statement.

The workaround for this is to perform an INSERT ... SELECT operation to copy the rows from the queue table you want to join with another table into a non-queue table, then join that non-queue table with the other table.

For example:

```
DELETE ALL FROM tempTbl;
INSERT INTO tempTbl
SELECT AND CONSUME TOP 1 * FROM qTbl;
SELECT c.name, t.customerId, t.orderId
FROM tempTbl AS t, customer AS c
WHERE t.customerId = c.customerId;
```

- You cannot specify a WHERE clause in a SELECT AND CONSUME statement.
- You cannot specify aggregate or ordered analytic functions in a SELECT AND CONSUME statement.
- The mandatory TOP *n* clause cannot be specified for any case other than *n*=1.
- SELECT AND CONSUME statements do not support the PERCENT or WITH TIES options for the TOP *n* clause. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

You cannot specify a SELECT AND CONSUME statement in a:

- Subquery
- Search condition
- Logical predicate
- Set operator

Ordering Queue Table Rows

You may need to alter the default FIFO ordering of a queue table before consuming them. For example, in a particular application, some queue table rows might have a higher priority and must be processed immediately, while others are less important. You can change the order of rows in the queue by altering their QITS value using normal SQL INSERT and UPDATE statements.

You can decrease the QITS value for a row by moving it toward the front of the queue, closer to consumption. Conversely, by increasing the QITS value for a row, you can move it toward the rear of the queue, further away from consumption.

You can use the various peek capabilities available to you in browse mode (see [Browsing a Queue Table](#)) to determine how the existing QITS values are distributed in the queue.

To create an artificial queue position for a new row, use the SQL INSERT statement.

Rather than letting the system create the QITS value for a row by default, you instead insert it into the queue table with a QITS value that you specify. In the following example, several rows have already been inserted into a queue table named *order_QT* on May 25th 2004. The example request inserts a new row into the queue ahead of the existing rows by supplying a false earlier QITS value for the INSERT.

```
INSERT INTO order_QT (qits, qty, item, description, price)
VALUES ('2004-05-25 00:00:00', 1, 'RLLB1', '1 doz BASEBALLS',
      25.00);
```

To create an artificial queue position for a row that already exists in a queue table, use the SQL UPDATE statement.

You can either increment or decrement the QITS value for a row in 1 of 2 ways.

- By changing the existing value of an INTERVAL constant.
- By assigning a specific value.

The following example updates the QITS value for the row with a queue ID number of 29 by decrementing the QITS timestamp value by an INTERVAL constant.

```
UPDATE order_QT
SET qits = qits - INTERVAL '2' HOUR
WHERE qsn = 29;
```

The following example updates the QITS value for the same row with a specific timestamp value.

```
UPDATE order_QT
SET qits = '2004-05-25 00:00:00'
WHERE qsn = 29;
```

You can also use the Upsert form of the UPDATE statement or the MERGE statement to order queue table rows.

The following example uses the Upsert form of the UPDATE statement to increment by 3 hours the QITS timestamp value for an existing row in the *order_QT* table having an order number value of BZ22905789. If no such row exists, the system inserts a new row into the table using the values specified in the VALUES clause of the conditional INSERT statement.

```
UPDATE order_QT
SET qits = qits + INTERVAL '3' HOUR
WHERE order_num = 'BZ22905789'
ELSE
```

```
INSERT INTO order_QT (qits, order_num, qty, item, description,
                      price)
VALUES ('2004-05-26 00:00:00', 'BZ22905789', 1, 'RCG11',
       'Catcher Glove', 55.00);
```

Note that update operations on a queue table should only be part of the exception handling logic of an application and *not* a common occurrence. When you modify a queue table by means of an update operation, its run time cache is purged, which causes the system to perform a full-table scan of the table to rebuild that cache. The queue table run time cache is limited to approximately 20,000 row entries per PE (the exact limit is 1 MB of row entries per PE). Additional categorical limits are the following: approximately 2,000 row entries per PE and exactly 100 queue table entries per PE.

Updating rows in an unpopulated queue table has no effect on any transaction in a delayed state waiting for rows to appear in that unpopulated queue table.

Populating a Queue Table

You populate a queue tables in the same way you populate non-queue tables: with INSERT statements, INSERT ... SELECT statements, or using a load utility like Teradata Parallel Data Pump or Teradata Parallel Transporter (using the STREAM and INSERT operators) that generates those statements.

You can use a simple INSERT ... SELECT statement to populate a queue table from either a queue- or a non-queue table. You can also use an INSERT ... SELECT AND CONSUME statement to populate a queue table with rows from another queue table.

Do *not* insert the QITS value, because the system inserts it by default using the value for CURRENT_TIMESTAMP.

Similarly, you do not insert the QSN value if your queue table uses a system-generated identity column.

The source table for an INSERT ... SELECT statement can be either another queue table or a non-queue table.

The following example ANSI session mode transaction uses an INSERT statement and multiple INSERT ... SELECT requests to insert several rows into a queue table named *shopping_cart*.

```
INSERT INTO shopping_cart (order_num, product, quantity)
VALUES ('I07219100', 'dozen baseballs' , 1);

INSERT INTO shopping_cart (order_num, product, quantity)
  SELECT order_num, product, quantity
  FROM mail_orders;

INSERT INTO shopping_cart (order_num, product, quantity)
  SELECT order_num, product, quantity
  FROM backorder_tbl
  WHERE order_num = 'I00200314';
```

```
COMMIT;
```

Considerations for Populating an Unpopulated Queue Table in a Delayed State

One or more sessions could be in a delayed state waiting for rows to be inserted at the time you populate an unpopulated queue table. Without regard to fairness, Vantage reinstates 1 or more of those delayed sessions, up to the number of rows inserted into the table, to enable them to continue their consume mode requests.

| Session Mode | Use the following after any INSERT statements to make those rows visible to any delayed sessions |
|--------------|--|
| ANSI | COMMIT |
| Teradata | <ul style="list-style-type: none"> • SEMICOLON character (implicit transaction) • END TRANSACTION statement terminated with a SEMICOLON character (explicit transaction) |

Using Multistatement Requests To Populate Queue Tables

You can also use multistatement requests to populate queue tables.

In the following example, three rows are inserted into a queue table named *shopping_cart* using 1 multistatement request.

```
INSERT INTO shopping_cart (order_num, product, quantity)
VALUES ('AX3725079002', 'Youth Tee', 2)
;INSERT INTO shopping_cart (order_num, product, quantity)
VALUES ('AX3725079002', 'Youth Shorts', 4)
;INSERT INTO shopping_cart (order_num, product, quantity)
VALUES ('AX3725079002', 'Youth Acc', 1);
```

While this is a fast method of populating a queue table, it does not guarantee that the rows are consumed in the original FIFO order. This is because if you do not supply a QITS value for the individual rows, each INSERT statement assigns the same timestamp to the QITS column because the system calculates the default value for CURRENT_TIMESTAMP only once for the entire multistatement request, and it then inserts that value into the QITS column of each row inserted by that request.

You *can* maintain row-specific QITS uniqueness and FIFO order for multistatement INSERT statements while also preserving the default CURRENT_TIMESTAMP value by adding a unique interval value to the default current timestamp value.

The following example shows how to do this, incrementing each statement in the request by an arbitrary INTERVAL SECOND value of 0.001 seconds.

```
INSERT INTO shoppingcart (qits, ordernum, product, quantity)
VALUES (CURRENT_TIMESTAMP, 'AX3725079002', 'Youth Tee', 2)
;INSERT INTO shoppingcart (qits, ordernum, product, quantity)
VALUES (INTERVAL '0.001' SECOND + CURRENT_TIMESTAMP,
        'AX3725079002', 'Youth Shorts', 4)
;INSERT INTO shoppingcart (qits, ordernum, product, quantity)
VALUES (INTERVAL '0.002' SECOND + CURRENT_TIMESTAMP,
        'AX3725079002', 'Youth Acc', 1);
```

Related Information

See CREATE TABLE (Queue Table Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for the syntax used to create queue tables.

CREATE TRANSFORM - CREATE VIEW

These topics provide supplemental usage information about selected SQL DDL statements alphabetically from CREATE TRANSFORM through CREATE VIEW.

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE TRANSFORM and REPLACE TRANSFORM

How REPLACE TRANSFORM Differs From CREATE TRANSFORM

For REPLACE TRANSFORM:

- If the specified transform group exists, the system replaces it with the new definition.
- If the specified transform group does not exist and the associated UDT does not have a defined transform group, the system creates the specified transform group.

Function of UDT Transforms

The CREATE TRANSFORM statement associates a transform group with a UDT. You do not need to create a transform group for any Period or Geospatial data types.

There must be a transform group for each UDT you develop. Note that the system defines a transform group for you when you create a new distinct UDT by default, but you must explicitly define a transform group for all new structured UDTs.

A UDT must have its transform functionality completely defined to be used as the column type of any table, otherwise an attempt to create or alter a table to include a column having the UDT type aborts and the system returns an error to the requestor.

The transform group of a UDT performs the following mappings:

- From a particular database UDT to the server form of a particular predefined client data type.
A routine that performs this transform is referred to as a fromsql transform routine.
- From a predefined client data type to a particular UDT in the database.
A routine that performs this transform is referred to as a tosql transform routine.

Such a pair of transform mappings is called a transform group. A transform group defines a single tosql/fromsql pair. There must be one, and only one, transform group per UDT.

You can perform a valid CREATE TRANSFORM request that associates only a tosql routine or only a fromsql routine with a UDT, but you must define complete transform functionality for a UDT before you can specify it as the column data type of any table. If you do not, the system aborts the ALTER TABLE or CREATE TABLE request and returns an error to the requestor.

| FOR this type of UDT ... | THE transform group functionality ... |
|---|--|
| one-dimensional and multidimensional ARRAY and VARRAY | <p>is generated automatically by the database.</p> <p>The format of the transformed output in a VARCHAR string is a string of each array element value, referred to as the transformed value string, separated by a comma and delimited by parentheses as indicated below. Assuming the array has <i>n</i> elements:</p> <p>(<element_1>,<element_2>, ... <element_n>)</p> <p>The following rules apply.</p> |

| FOR this type of UDT ... | THE transform group functionality ... |
|--------------------------|---|
| | <ul style="list-style-type: none"> • BLOB, CLOB, and Geospatial element types are not supported. • If the element type is not CHARACTER, VARCHAR, or a UDT with a CHARACTER or VARCHAR attribute, the definition of the array constructor/transformed value string is VARCHAR(64000) CHARACTER SET LATIN. If the size of the transformed value string is greater than 64k when the array type is being created, the request aborts and the database returns an error to the requestor. • If the element type is CHARACTER, VARCHAR, or a UDT with a CHARACTER or VARCHAR attribute, the definition of the array transformed value string is VARCHAR(<i>max</i>) of the same character set as its element type. For the VARCHAR case, <i>max</i> signifies the largest numeric value possible for the size of a VARCHAR in the current character set. For a Latin character set, this is 64 KB. If the size of the transformed value string is greater than the value of <i>max</i> when the ARRAY type is being created, the request aborts and the database returns an error to the requestor. • The system does not report any uninitialized elements. If an element value has been initialized, then all of the elements before it will have also been initialized either to null or to a value. • Overflow avoidance. The size of an array transformed value string must be within the size limit of a VARCHAR, CHARACTER, and CLOB data types, particularly VARCHAR data type). If the size of an array transformed value string is bigger than the size limit of a VARCHAR variable when the ARRAY type is being created, the database aborts the request and returns an error to the requestor. • A null element must be indicated by a NULL literal. If the server character set is other than Latin, the null element is indicated by the corresponding encoding of a NULL literal in the server character set. The same applies on a null structured UDT value or a structured UDT with a null attribute. |
| | <ul style="list-style-type: none"> • The database ignores any pad, new line, or tab characters whether they precede the comma or follow it. The same is true if such characters precede the last APOSTROPHE character or follow the first APOSTROPHE character. If the element type is CHARACTER, VARCHAR, or UDT with CHARACTER or VARCHAR attributes, and if ARRAY/VARRAY elements contain many embedded APOSTROPHE characters, this requires an extra APOSTROPHE character to distinguish the embedded APOSTROPHE character. This limits the total size of the transform string that can be output when selecting the ARRAY/VARRAY column because it counts the embedded APOSTROPHE characters. If your ARRAY/VARRAY elements contain embedded APOSTROPHE characters, they are also be output when the ARRAY/VARRAY column is selected in transforms ON mode, using the fromsql transform. In the worst case, a string with all APOSTROPHE characters embedded, the max transform string is reduced by half. Therefore, if the total size of the transform string that could be generated for an ARRAY/VARRAY type exceeds the maximum row size, the CREATE TYPE request for that ARRAY/VARRAY type aborts and the database returns an error to the requestor. |
| distinct | is generated automatically by the database. |

| FOR this type of UDT ... | THE transform group functionality ... |
|--------------------------|---|
| | <p>If the system-generated transform group semantics is adequate, you need not define explicit <code>tosql</code> and <code>fromsql</code> transform functionality using the <code>CREATE TRANSFORM</code> statement.</p> <p>If your applications require different or richer transform semantics, then you can specify explicit transform group functionality using <code>CREATE TRANSFORM</code>.</p> |
| structured | must be defined explicitly using the <code>CREATE TRANSFORM</code> statement. |

The system invokes transforms implicitly whenever UDT data is imported to or exported from the database.

The system does not permit you to specify a transform group for the UDT parameter set of external routines. Instead, when a process invokes an external routine, the system passes a UDT *handle* to the external routine argument rather than passing the UDT value to it directly. The external routine can use this UDT handle to get or set the value of a UDT argument by means of a set of library functions provided by Teradata. For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The predefined type that is specified in the transform group routines of a UDT is referred to as the external type of the UDT. The external type is always in Teradata server format, meaning that the normal client format-to-database format transformations continue to take place in addition to UDT-specific `tosql` and `fromsql` operations.

For a `tosql` operation, this means the following actions occur in the sequence indicated.

1. The system transforms the external type of the UDT from its client format to its database format.
2. The system performs the `tosql` routine.

For a `fromsql` operation, this means the following actions occur in the sequence indicated:

1. The system performs the `fromsql` routine to transform the UDT to its external type database form.
2. The system transforms the external type from its database format to its client format.

Creating Parallel Cast and Transform Functionality

Best practices suggest that you should provide parallel functionality for the transform group functionality and the equivalent cast functionality.

The following table explains what this means more explicitly:

| Cast Functionality | Equivalent Transform Functionality |
|---|---|
| External predefined data type to UDT cast | External predefined data type to UDT <i>tosql</i> transform |
| UDT to external predefined data type cast | UDT to external predefined data type <i>fromsql</i> transform |

This redundant functionality is particularly recommended if you want to use a UDT in the same way you would use any other data type, especially with respect to load utilities and the `USING` row descriptor.

The simplest way to accomplish these equivalencies is to reference the same external routines in both the CREATE TRANSFORM and CREATE CAST statements for the same UDT.

The implications of this best practice differ for distinct and structured UDTs as follows:

| FOR this kind of UDT ... | You must do the following work to create parallel casting and transform functionality... |
|--------------------------|--|
| distinct | none if you plan to use the system-generated casts and transforms. However, if you decide to write your own external cast and transform routines for the UDT, you must create the parallel cast and transform functionality explicitly using the CREATE CAST and CREATE TRANSFORM statements, respectively. |
| structured | create the necessary casts and transforms explicitly, using the same external routines for both. |

See [CREATE CAST and REPLACE CAST](#) for information about how to create casting functionality for a UDT.

Rules For tosql Transform Routines

A tosql routine transforms a predefined data type to a UDT.

The tosql transform routines are generally used by utilities such as FastLoad to convert incoming client data directly into its corresponding UDT data type in the database.

The rules for tosql routines are as follows:

- The tosql transform routine can *only* be a UDF.
methods are not valid.
- The tosql UDF transform routine must be defined as DETERMINISTIC.
- The tosql UDF transform routine must have one and only one declared parameter, which must be a predefined data type.
The RESULT data type must be the UDT.
- The tosql UDF transform routine must be contained within the *SYSUDTLIB* database.
- If you declare both a tosql transform routine and a fromsql routine, then the declared parameter data type of the tosql routine must be identical to the RESULT data type of the fromsql routine.

Valid tosql Transform Functionality

The following table documents the valid inputs for tosql functionality.

| Source Input | Source Data Type of Routine Definition | Target Data Type of Routine Definition | Target Output |
|---------------------|--|--|---------------|
| BYTEINT SMALLINT | BYTEINT | UDT | UDT |

| Source Input | Source Data Type of Routine Definition | Target Data Type of Routine Definition | Target Output |
|---|--|--|---------------|
| INTEGER BIGINT DECIMAL NUMERIC REAL FLOAT DOUBLE PRECISION DATE CHARACTER VARCHAR | SMALLINT | | |
| | INTEGER | | |
| | BIGINT | | |
| | DECIMAL NUMERIC | | |
| | REAL FLOAT DOUBLE PRECISION | | |
| BYTE VARBYTE BLOB | BYTE | | |
| | VARBYTE | | |
| | BLOB | | |
| CHARACTER VARCHAR CLOB BYTEINT SMALLINT INTEGER BIGINT DECIMAL NUMERIC REAL FLOAT DOUBLE PRECISION DATE | CHARACTER | | |
| | VARCHAR CHARACTER VARYING | | |
| CHARACTER VARCHAR CLOB | CLOB | | |
| DATE BYTEINT SMALLINT INTEGER BIGINT DECIMAL NUMERIC REAL FLOAT DOUBLE PRECISION CHARACTER VARCHAR | DATE | UDT | UDT |
| TIME TIME WITH TIME ZONE CHARACTER VARCHAR | TIME | | |
| | TIME WITH TIME ZONE | | |

| Source Input | Source Data Type of Routine Definition | Target Data Type of Routine Definition | Target Output |
|--|--|--|---------------|
| | TIMESTAMP | | |
| | TIMESTAMP WITH TIME ZONE | | |
| | | | |
| INTERVAL YEAR INTERVAL YEAR TO MONTH INTERVAL MONTH | INTERVAL YEAR | | |
| | INTERVAL YEAR TO MONTH | | |
| | INTERVAL MONTH | | |
| INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL SECOND | INTERVAL DAY | | |
| | INTERVAL DAY TO HOUR | | |
| | INTERVAL DAY TO MINUTE | | |
| | INTERVAL DAY TO SECOND | | |
| | INTERVAL HOUR | | |
| | INTERVAL HOUR TO MINUTE | | |
| | INTERVAL HOUR TO SECOND | | |
| | INTERVAL MINUTE | | |
| | INTERVAL MINUTE TO SECOND | | |
| | INTERVAL SECOND | | |

Rules For fromsql Transform Routines

A fromsql routine transforms a UDT to a predefined data type.

The fromsql transforms are commonly used both utilities and by some commonly invoked SQL statements.

The following example shows an operation that causes implicit invocation of the fromsql transform routine.

Suppose a client application executes a SELECT statement whose column list contains a UDT column or expression. The following sequence of events occurs.

1. The system implicitly invokes the fromsql transform routine of the defined transform group for that UDT to map the UDT value to a predefined data type value.
2. The client application receives the predefined type value.

For example, consider the following example:

```
SELECT udt_column
FROM table1;
```

The fromsql transform routine converts the values of *udt_column* into the corresponding predefined data type values before returning them to the client application. If the UDT does not have a defined transform or if its transform group does not have a defined fromsql routine, the request aborts and the system returns an error to the requestor.

The rules for fromsql routines are as follows:

- The fromsql transform routine can be either a method or a UDF.
- The fromsql transform routine must be defined as DETERMINISTIC.
- If the fromsql transform routine is a UDF, then *function_name* or *specific_function_name* must specify a UDF contained within the *SYSUDTLIB* database.
- The fromsql transform routine must have the following declared parameter list and result data type:

| IF the fromsql transform routine is a ... | THEN the ... |
|---|--|
| UDF | UDF must have one declared parameter whose type is the UDT. The RESULT data type must be a predefined data type. |
| method | method cannot have a declared parameter. In this case, the UDT instance is an implicit parameter. The RESULT data type must be a predefined data type. |

- If you specify both tosql and fromsql transform routines, then the declared parameter type of the tosql routine must be the same as the RESULT data type for the fromsql routine.

Valid fromsql Transform Functionality

The following table documents the valid targets for fromsql functionality.

| Source Input | Source Data Type of Routine Definition | Target Data Type of Routine Definition | Target Output |
|--------------|--|--|---|
| UDT | UDT | BYTEINT | BYTEINT SMALLINT INTEGER BIGINT DECIMAL |
| | | SMALLINT | |
| | | INTEGER | |
| | | BIGINT | |
| | | DECIMAL | NUMERIC REAL FLOAT DOUBLE PRECISION CHARACTER |
| | | NUMERIC | |
| | | REAL | |

| Source Input | Source Data Type of Routine Definition | Target Data Type of Routine Definition | Target Output |
|--------------|--|--|--------------------------|
| | | FLOAT | VARCHAR |
| | | DOUBLE PRECISION | DATE |
| | | BYTE | BYTE |
| | | VARBYTE | VARBYTE |
| | | BLOB | BLOB |
| | | CHARACTER | CHARACTER |
| | | VARCHAR | VARCHAR |
| | | CHARACTER VARYING | C |
| | | | LOB |
| | | | BYTEINT |
| | | | SMALLINT |
| | | | INTEGER |
| | | | BIGINT |
| | | | DECIMAL |
| | | | NUMERIC |
| | | | REAL |
| | | | FLOAT |
| | | | DOUBLE PRECISION |
| | | | DATE |
| | | | TIME |
| | | | TIME WITH TIME ZONE |
| | | | TIMESTAMP |
| | | | TIMESTAMP WITH TIME ZONE |
| | | CLOB | CLOB |
| | | | CHARACTER |
| | | | VARCHAR |
| | | DATE | DATE |
| | | | BYTEINT |
| | | | SMALLINT |
| | | | INTEGER |

| Source Input | Source Data Type of Routine Definition | Target Data Type of Routine Definition | Target Output |
|--------------|--|--|---------------------------|
| | | | BIGINT |
| | | | DECIMAL |
| | | | NUMERIC |
| | | | REAL |
| | | | FLOAT |
| | | | DOUBLE PRECISION |
| | | | CHARACTER |
| | | | VARCHAR |
| | | TIME | TIME |
| | | TIME WITH TIME ZONE | TIME WITH TIME ZONE |
| | | TIMESTAMP | TIMESTAMP |
| | | TIMESTAMP WITH TIME ZONE | TIMESTAMP WITH TIME ZONE |
| | | INTERVAL YEAR | INTERVAL YEAR |
| | | INTERVAL YEAR TO MONTH | INTERVAL YEAR TO MONTH |
| | | INTERVAL MONTH | INTERVAL MONTH |
| | | INTERVAL DAY | INTERVAL DAY |
| | | INTERVAL DAY TO HOUR | INTERVAL DAY TO HOUR |
| | | INTERVAL DAY TO MINUTE | INTERVAL DAY TO MINUTE |
| | | INTERVAL DAY TO SECOND | INTERVAL DAY TO SECOND |
| | | INTERVAL HOUR | INTERVAL HOUR |
| | | INTERVAL HOUR TO MINUTE | INTERVAL HOUR TO MINUTE |
| | | INTERVAL HOUR TO SECOND | INTERVAL HOUR TO SECOND |
| | | INTERVAL MINUTE | INTERVAL MINUTE |
| | | INTERVAL MINUTE TO SECOND | INTERVAL MINUTE TO SECOND |
| | | INTERVAL SECOND | INTERVAL SECOND |

Transform Requirements for Teradata Utilities

Some Teradata utilities require you to define transforms to permit them to process UDT data. The following list explains what those requirements are:

- Database utilities
None of the Teradata platform-based utilities have transform requirements.
- Teradata Tools and Utilities

The following table explains the transform requirements for various Teradata Tools and Utilities:

| Utility | Transform Requirements |
|---|--|
| FastExport | fromsql |
| FastLoad | tosql |
| MultiLoad | None |
| Teradata Parallel Data Pump (TPump) | None |
| Teradata Parallel Transporter: <ul style="list-style-type: none"> UPDATE LOAD EXPORT | <ul style="list-style-type: none"> tosql fromsql |
| Teradata Parallel Transporter STREAMS | None |

Miscellaneous UDT Information Related To Client Applications and Transforms

Because the entire purpose of transform groups is to pass UDTs between the client and the database platform transparently, this section describes some of the important points relative to how client software deals with UDTs. To be accurate, UDTs are not passed to the client. They are transformed into a predefined type and data having that predefined type is passed to the client.

This information is by no means comprehensive. To get the full picture of how various client utilities deal with UDTs, consult the appropriate Teradata Tools and Utilities user documentation.

The first thing to understand is that the Teradata Tools and Utilities are not UDT-aware. Client software does not see UDT values, but values having a predefined data type representing a UDT on the Teradata platform. How a platform UDT is transformed to a client predefined type is entirely at the discretion of the transform developer.

Because of this, the following things are true about defining input record layouts (such as the FIELD command in a MultiLoad or TPump record layout):

- For distinct UDTs, you must specify the underlying predefined (intrinsic) data. The database handles all conversions to and from the UDT by means of transform groups. The Teradata Tools and Utilities do not know that a value will eventually reside in, or that it came from, a distinct UDT.
- For structured UDTs, you have two options:

- Specify the client representation of the UDT as a single field with a single predefined data type.

For example, suppose you have a structured UDT called *circle* that is constructed from 3 attributes, each having the predefined FLOAT data type. The attributes are as follows:

- The x coordinate of the center of the circle.
- The y coordinate of the center of the circle.
- The radius of the circle.

Pursuing the first option, you would combine the attributes into a single field and represent the structured UDT on the client as `circle BYTE(24)`.

The database handles the conversion to and from the external and internal UDT representations by means of user-defined transforms.

- Specify the client representation of the UDT as 3 separate fields.

Again using the circle example, you would specify 3 separate fields, each having the predefined FLOAT data type, for example:

- `x_coord` FLOAT
- `y_coord` FLOAT
- `radius` FLOAT

You would then construct the structured circle UDT on the Teradata platform side using the NEW expression in the VALUES clause of the INSERT, or the SET clause of the UPDATE. For more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

For example, for circle UDT, the syntax is the following:

```
NEW circle(:x,:y,:r)
```

See the appropriate Teradata Tools and Utilities documents for details.

CREATE TRANSFORM/REPLACE TRANSFORM And System-Generated Constructor UDFs

Note that creating or replacing a transform also causes the system-generated UDF constructor function for the UDT to be recompiled invisibly (invisible in that the system does not return any compilation messages unless the compilation fails for some reason, in which case the system returns an appropriate error message to the requestor).

CREATE TRIGGER/ REPLACE TRIGGER

Teradata Extensions to ANSI Triggers

Vantage triggers provide the following Teradata extensions to the ANSI SQL:2011 specification for CREATE TRIGGER.

- ORDER clause
- ENABLED/DISABLED option
- OLD_NEW_TABLE transition table

Function of REPLACE TRIGGER Requests

REPLACE TRIGGER executes as a DROP TRIGGER request followed by a CREATE TRIGGER request, except for the handling of the privileges granted to the original trigger. The database retains all of the privileges that were granted directly on the original trigger for the replacement trigger.

If the specified trigger does not exist, the REPLACE TRIGGER statement creates it. In this case, the REPLACE statement has the same effect as a CREATE TRIGGER statement.

If an error occurs during the replacement of the trigger, the existing trigger is not dropped and its definition remains as it was before the replacement attempt was made.

REPLACE TRIGGER does not change the timestamp generated when the trigger was originally created. For information about changing trigger timestamps, see ALTER TRIGGER in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Trigger Terminology

The following table defines terms associated with the creation and use of an SQL trigger:

| Term | Description |
|--------------------|---|
| DISABLED | An optional element of a trigger definition that defines the current state of the trigger. A disabled trigger is inactive until it is enabled, but its definition remains in the data dictionary. |
| ENABLED | An optional element of a trigger definition that defines the current state of the trigger. An enabled trigger is an active database object. |
| ORDER clause | An optional clause that assigns an integer value to determine the order of execution of the trigger when multiple triggers having the same trigger action time and trigger event are defined on the same table. |
| REFERENCING clause | An optional clause that allows the WHEN condition and triggered actions of a trigger to reference the set of rows in the transition table set. See REFERENCING Clause . |

| Term | Description |
|-------------------------|--|
| Subject table | The table with which a trigger is associated and on which its triggering event takes place. |
| Transition row | A row in the transition table. |
| Transition table | A temporary table that contains either old or new values, or both, for rows modified by a triggering statement. See Transition Tables for more information about transition tables. |
| Trigger action time | A specification that indicates when triggered SQL statements perform in relation to the triggering event. A triggered statement performs either BEFORE or AFTER a triggering event. |
| Trigger granularity | A specification that indicates whether the trigger performs for each row or for each statement modified by the triggering event. Trigger granularity is expressed as the trigger type, either FOR EACH STATEMENT or FOR EACH ROW. FOR EACH STATEMENT is the default granularity for all triggers. |
| Triggered action | The main action part of the trigger definition, consisting of the following elements: <ul style="list-style-type: none"> • An optional trigger granularity clause • An optional WHEN condition • A triggered SQL statement |
| Triggered SQL statement | A set of SQL statements fired by the triggering event. A triggered SQL statement can modify one or more tables, including the subject table. |
| Triggering event | The DML statement whose execution fires the trigger. This is a DELETE, INSERT, or UPDATE operation or a variation of these operations. An UPDATE operation can have a list of columns associated with it. |
| WHEN <i>condition</i> | A Boolean search condition that defines different things for different trigger types. <ul style="list-style-type: none"> • For a Statement trigger, the WHEN condition determines whether the trigger executes. • For a Row trigger, the WHEN condition determines the set of changed rows for which the trigger executes. |

Transition Tables

A transition table is a dynamically created table that you can reference using a correlation name.

The following table indicates the type of rows that are contained by a transition table depending on the triggering statement that caused the transition table to be created.

| Triggering Statement | Generates a Transition Table with this Row Set |
|----------------------|--|
| DELETE | Old rows only. |
| INSERT | New rows only. |

| Triggering Statement | Generates a Transition Table with this Row Set |
|--|--|
| <ul style="list-style-type: none"> • UPDATE • MERGE UPDATE | Old and new rows. |

The following table documents the rows that can be referenced in a transition table depending on trigger granularity.

| Trigger Granularity | References Rows for this Transition Table |
|---------------------|--|
| Row | <ul style="list-style-type: none"> • NEW [ROW] • NEW TABLE • NEW_TABLE |
| Statement | <ul style="list-style-type: none"> • OLD [ROW] • OLD TABLE • OLD_TABLE • OLD_NEW_TABLE <p>OLD_NEW_TABLE is valid only for AFTER UPDATE triggers.</p> |

REFERENCING Clause

The REFERENCING clause allows the WHEN condition and triggered actions of a trigger to reference the set of rows in the transition table set. This indirect access to the transition table rows is useful for making comparisons between OLD and NEW rows in the subject table or for use in the triggered action.

The references are to transient and virtual tables, which can include values from the subject table either before (OLD ROW or OLD TABLE), after (NEW ROW or NEW TABLE), or before and after (OLD_NEW_TABLE) the data-changing statement. Only AFTER UPDATE statement triggers can produce OLD_NEW_TABLE transition tables.

These are distinct from direct references to the subject table. Triggers cannot make direct comparisons between before and after rows of a subject table.

A reference to the subject table in a triggered SQL statement is called an outer reference. Outer references occur when the triggered SQL statement of a WHEN clause or the WHEN clause itself refers to a column or row in the subject table.

Rules For Using the REFERENCING Clause

The following rules apply to the REFERENCING clause:

- The scope for the correlation names in the clause is the entire trigger.
For this reason, the names cannot duplicate names of any tables or views specified in the trigger definition.
- REFERENCING correlation names cannot be repeated in the clause specification.

- After the **REFERENCING** keyword, you can specify the following five options in any order and combination; however, each option can be specified only once:

- **OLD** or **OLD ROW**

You can only specify **OLD** or **OLD ROW** for row triggers.

- **NEW** or **NEW ROW**

You can only specify **OLD** or **OLD ROW** for row triggers.

- **OLD TABLE** or **OLD_TABLE**
- **NEW TABLE** or **NEW_TABLE**
- **OLD_NEW_TABLE**

Note that you can only specify **OLD_NEW_TABLE** for **AFTER UPDATE** triggers.

Any of the five transition table types can contain UDT columns.

- Triggers can reference any **OLD [ROW]**, **NEW [ROW]**, **OLD TABLE**, **NEW TABLE**, or **OLD_NEW_TABLE** subject to the following limitations:

| THIS type of trigger ... | References ... |
|--------------------------------|---|
| AFTER statement trigger | only transition tables. |
| AFTER row trigger | both transition rows and transition tables. |
| BEFORE row trigger | only transition rows. |

- You can access data from a transition table using the specified **OLD TABLE**, **OLD_TABLE**, **NEW TABLE**, **NEW_TABLE**, and **OLD_NEW_TABLE** correlation names.

Row triggers can also access values from single rows in the transition table for which the trigger is currently being processed using the **OLD [ROW]** and **NEW [ROW]** correlation names. You cannot specify **OLD**, **OLD ROW**, **NEW**, or **NEW ROW** for statement triggers.

Cascaded row triggers *cannot* access transition table data (see [Cascading and Recursion of Triggers](#)).

- Triggered action statement subqueries can select data from the old and new tables using the **OLD TABLE**, **OLD_TABLE**, **NEW TABLE**, and **NEW_TABLE** correlation names, but they cannot select data using the **OLD [ROW]** or **NEW [ROW]** correlation names.
- The behavior of **OLD TABLE**, **OLD_TABLE**, **NEW TABLE**, **NEW_TABLE**, and **OLD_NEW_TABLE** references in the **REFERENCING** clause is as follows:

| THIS transition table type ... | Contains the entire set of transition rows ... |
|--------------------------------|---|
| OLD TABLE | before any UPDATE s or DELETE s are made on the subject table. |
| NEW TABLE | after any UPDATE s or INSERT s are made on the subject table, including default values, constraint checks, and so on. |
| OLD_NEW_TABLE | both before and after any UPDATE s are made on the subject table. |

The following example demonstrates the usefulness of OLD_NEW_TABLE for AFTER UPDATE triggers. Suppose you have the following very simple *inventory* table:

```
CREATE TABLE inventory (
  prod_num  INTEGER,
  avail_qty INTEGER)
UNIQUE PRIMARY INDEX (product_num);
```

The rows in *inventory* initially look like this:

| Inventory | |
|-----------|-----------|
| prod_num | avail_qty |
| 101 | 100 |
| 201 | 50 |
| 301 | 150 |

Suppose you have defined an AFTER UPDATE trigger that is fired when the following triggering statement is executed:

```
UPDATE inventory
SET avail_qty = avail_qty - 50
WHERE prod_num IN (101, 301);
```

Because of the WHERE clause condition in this request, rows for *prod_num* 201 are not updated. Depending on how the trigger is defined, the following three transition tables are possible when the trigger fires:

| OLD_TABLE | | NEW_TABLE | | OLD_NEW_TABLE | | | |
|-----------|-----------|-----------|-----------|---------------|-----------|-----------|-----------|
| | | | | Old_Value | | New_Value | |
| prod_num | avail_qty | prod_num | avail_qty | prod_num | avail_qty | prod_num | avail_qty |
| 101 | 100 | 101 | 50 | 101 | 100 | 101 | 50 |
| 301 | 150 | 301 | 100 | 301 | 150 | 301 | 100 |

Given these transition tables, suppose you want to know the before and after update values for the rows updated by this trigger. To do this, you specify the following SELECT request as part of the FOR EACH STATEMENT WHEN clause in the trigger definition. The request returns the following result table:

```
SELECT *
FROM OldTab, NewTab;
```

| OldTab | | NewTab | |
|----------|-----------|----------|-----------|
| prod_num | avail_qty | prod_num | avail_qty |
| 101 | 100 | 101 | 50 |
| 101 | 100 | 301 | 100 |
| 301 | 150 | 101 | 50 |
| 301 | 150 | 301 | 100 |

This is not the desired output because it contains nonsense rows, shaded in the table, that were not updated by the trigger. The rows make no sense because they represent *prod_num* values that were either changed from 101 to 301 or from 301 to 101, and the UPDATE request only changed *avail_qty* values because of its SET clause condition *avail_qty* = *avail_qty* - 50.

To filter those rows, you must add a WHERE clause condition to the SELECT request that filters any rows with changed *prod_num* values from the output, as follows:

```
SELECT *
FROM OldTab, NewTab
WHERE OldTab.prod_num = NewTab.prod_num;
```

After you add this condition, the request returns the following result table:

| OldTab | | NewTab | |
|----------|-----------|----------|-----------|
| prod_num | avail_qty | prod_num | avail_qty |
| 101 | 100 | 101 | 50 |
| 301 | 150 | 301 | 100 |

To achieve the desired result, Vantage had to take the following steps:

1. Join *OldTab* to *NewTab*.
2. Filter the rows retrieved by the join operation with a WHERE clause predicate to eliminate undesired rows from the result table.

You can avoid this extra processing by creating your AFTER UPDATE triggers to reference only OLD_NEW_TABLE, which eliminates both the join operation and the predicate filtering required to select the desired rows from the rows created by joining OLD_TABLE and NEW_TABLE as you must do if you create the trigger using those two transition tables instead of OLD_NEW_TABLE.

The following WHEN clause SELECT request returns only the desired rows without need of either a join or a WHERE clause predicate. The rows that were filtered by the predicate *OldTab.prod_num* =

`NewTab.prod_num` in the previous request do not need to be filtered in this request because there is no join operation to create them artificially in the first place.

```
SELECT *
FROM OldNewTab;
```

This request returns the identical results table as the filtered request that joined *OldTab* to *NewTab*:

| OldTab | | NewTab | |
|----------|-----------|----------|-----------|
| prod_num | avail_qty | prod_num | avail_qty |
| 101 | 100 | 101 | 50 |
| 301 | 150 | 301 | 100 |

The following table summarizes the rules for using transition tables.

| REFERENCING Clause Specification | Usage Information |
|--|---|
| OLD [ROW] AS <i>old_transition_variable_name</i> | <p>You can use <i>old_transition_variable_name</i> to reference columns in a transition row before it was modified by a triggering event. This specification applies to the following trigger types only.</p> <ul style="list-style-type: none"> ◦ DELETE ◦ MERGE UPDATE ◦ UPDATE |
| NEW [ROW] AS <i>new_transition_variable_name</i> | <p>You can use <i>new_transition_variable_name</i> to reference columns in a transition row after it has been modified by a triggering event. This specification applies to the following trigger types only.</p> <ul style="list-style-type: none"> ◦ INSERT ◦ MERGE INSERT ◦ MERGE UPDATE ◦ UPDATE |
| OLD_TABLE AS <i>old_table_name</i> or OLD TABLE AS <i>old_table_name</i> | <p>You can use <i>old_table_name</i> to reference a table before it was modified by a triggering event. This specification applies to the following trigger types only.</p> <ul style="list-style-type: none"> ◦ DELETE ◦ MERGE UPDATE ◦ UPDATE <p>For example, after a delete from <i>subj_tab</i>, the following trigger counts the number of rows in <i>old_tab</i> and then inserts the count into <i>trig_count_tab</i>.</p> <pre>CREATE TRIGGER trig1 AFTER DELETE ON subj_tab REFERENCING OLD_TABLE AS old_tab FOR EACH STATEMENT</pre> |

| REFERENCING Clause Specification | Usage Information |
|---|---|
| | <pre>(INSERT trig_count_tab SELECT COUNT(*) FROM old_tab);</pre> |
| <p>NEW_TABLE AS <i>new_table_name</i> or NEW TABLE AS <i>new_table_name</i></p> | <p>You can use <i>new_table_name</i> to reference a table after it was modified by a triggering event. This specification applies to the following trigger types only.</p> <ul style="list-style-type: none"> ◦ INSERT ◦ MERGE INSERT ◦ MERGE UPDATE ◦ UPDATE <p>For example, after an update to <i>subj_tab</i>, the following trigger counts the number of rows in <i>new_tab</i> and then inserts the count into <i>trig_count_tab</i>.</p> <pre>CREATE TRIGGER trig1 AFTER UPDATE ON subj_tab REFERENCING NEW_TABLE AS new_tab FOR EACH STATEMENT (INSERT trig_count_tab SELECT COUNT(*) FROM new_tab);</pre> |
| <p>OLD_NEW_TABLE AS <i>old_new_table_name</i></p> | <p>You can use <i>old_new_table_name</i> to reference a table both before and after it has been modified by a triggering event without having to join the NEW_TABLE and OLD_TABLE transition tables or having to filter the results of that join. This specification applies to the following trigger type only.</p> <ul style="list-style-type: none"> ◦ AFTER UPDATE <p>For example, after any update to <i>subj_tab</i>, the following trigger selects all four columns from any rows in <i>old_new_tab</i> that satisfy the condition <i>oldc2</i> > <i>newc2</i> and then inserts them into <i>trig_tab</i>:</p> <pre>CREATE TRIGGER trig1 AFTER UPDATE ON subj_tab REFERENCING OLD_NEW_TABLE AS old_new_tab (oldc1, oldc2, newc1, newc2) FOR EACH STATEMENT (INSERT trig_tab SELECT oldc1, newc1, oldc2, newc2 FROM old_new_tab WHERE oldc2 > newc2;);</pre> |

- OLD TABLE and NEW TABLE cannot be updated because they are not persistent. However, when referenced in a query, the old, new, and old_new transition tables are treated as if they were base tables. As a result, a request such as the following joins the OLD TABLE and NEW TABLE transition tables:

```
SELECT *
FROM old_table_alias, new_table_alias;
```

Because of the required join operation for this query, you should always specify an OLD_NEW_TABLE transition table for your AFTER UPDATE statement triggers to achieve better performance.

Row and Statement Triggers

Triggers are of two mutually exclusive types: row and statement. You cannot combine row and statement operations within a single trigger definition.

Both row and statement triggers can call stored procedures (see [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#) and [CREATE PROCEDURE and REPLACE PROCEDURE \(SQL Form\)](#)).

The following table summarizes the key differences between row and statement triggers.

| A row trigger ... | A statement trigger ... |
|--|--|
| fires once for each row changed by the triggering statement. The process of firing a trigger includes testing WHEN conditions for their truth value and not firing if the specified condition evaluates to FALSE. There is no limit on the number of times a row trigger fires. | fires only once. Statement triggers cannot access changed rows. |
| does not fire if no row is modified by the triggering statement. | fires even if no row is modified by the triggering statement. |
| can be either a BEFORE or AFTER trigger. | can only be an AFTER trigger. |
| can access any OLD values, NEW values; or any OLD TABLE, OLD_TABLE, NEW TABLE, or NEW_TABLE transition tables that make sense by means of the REFERENCING clause. For example, it makes no sense to perform an INSERT with an OLD [ROW] transition variable or an OLD TABLE transition table because there can be no pre-existing variable or table (see the next table for a complete list of valid and non-valid transition variables and tables). See the table beginning on the following page for details of which of these transitions are valid for a given triggered action statement. | cannot use the REFERENCING options of OLD and NEW transition table names, only the OLD TABLE, OLD_TABLE, NEW TABLE, and NEW_TABLE transition table names. See the table beginning on the following page for details of which of these transitions are valid for a given triggered action statement. |
| cannot specify OLD_NEW_TABLE transition tables. There is no analog of OLD_NEW_TABLE transition tables for row triggers. | can specify OLD_NEW_TABLE transition tables. |

If your application can be written using either a row trigger or a statement trigger, you can almost always achieve better performance using the statement trigger because it fires only once, while the equivalent row trigger fires once for each *row* that the triggering statement updates.

The following table summarizes the valid combinations of trigger types, trigger activation times, update operations, and transition variables and tables:

| Trigger Type | Activation Time | Update Operation | Valid Transitions |
|--------------|-----------------|---|--|
| ROW | BEFORE | DELETE | OLD [ROW] |
| | | INSERT | NEW [ROW] |
| | | MERGE INSERT | NEW [ROW] |
| | | MERGE UPDATE | <ul style="list-style-type: none"> • OLD [ROW] • NEW [ROW] |
| | | UPDATE | <ul style="list-style-type: none"> • OLD [ROW] • NEW [ROW] |
| | AFTER | DELETE | <ul style="list-style-type: none"> • OLD [ROW] • OLD_TABLE • OLD TABLE |
| | | INSERT | <ul style="list-style-type: none"> • NEW [ROW] • NEW_TABLE • NEW TABLE |
| | | MERGE INSERT | <ul style="list-style-type: none"> • NEW [ROW] • NEW_TABLE • NEW TABLE |
| | | MERGE UPDATE | <ul style="list-style-type: none"> • NEW [ROW] • NEW_TABLE • NEW TABLE • OLD [ROW] • OLD_TABLE • OLD TABLE |
| | | UPDATE | <ul style="list-style-type: none"> • NEW [ROW] • NEW_TABLE • NEW TABLE • OLD [ROW] • OLD_TABLE • OLD TABLE |
| STATEMENT | BEFORE | Not applicable. BEFORE STATEMENT triggers are not defined by the ANSI SQL:2011 standard, nor are they supported by Teradata. See Unsupported Trigger Features . | |
| | AFTER | DELETE | <ul style="list-style-type: none"> • OLD_TABLE • OLD TABLE |
| | | INSERT | <ul style="list-style-type: none"> • NEW_TABLE |

| Trigger Type | Activation Time | Update Operation | Valid Transitions |
|--------------|-----------------|------------------|---|
| | | | <ul style="list-style-type: none"> • NEW TABLE |
| | | MERGE INSERT | <ul style="list-style-type: none"> • NEW_TABLE • NEW TABLE |
| | | MERGE UPDATE | <ul style="list-style-type: none"> • NEW_TABLE • NEW TABLE • OLD_TABLE • OLD TABLE |
| | | UPDATE | <ul style="list-style-type: none"> • NEW_TABLE • NEW TABLE • OLD_TABLE • OLD TABLE • OLD_NEW_TABLE |

Triggered Action Statements

Triggered action statements can be one or more of the SQL statements summarized in the following table:

| Trigger Type | Valid Statements | |
|--------------|---|--|
| AFTER | <ul style="list-style-type: none"> • ABORT • CALL • DELETE • INSERT • INSERT ... SELECT <p>This applies only to standard INSERT ... SELECT statements. INSERT ... SELECT AND CONSUME statements are <i>not</i> valid as triggered action statements.</p> | <ul style="list-style-type: none"> • ROLLBACK • Atomic Upsert • UPDATE • EXEC of macros containing any of the valid statements |
| BEFORE | <ul style="list-style-type: none"> • ABORT • EXEC of macros containing no data-changing statements | <ul style="list-style-type: none"> • ROLLBACK • SET clause (INSERT and UPDATE row triggers only). |

You can also execute UDFs and call stored procedures from within triggered action statements. Any valid triggered action statement can contain UDT expressions.

The general rules for the use of these statements in SQL also apply to their use as triggered action statements, with the following differences:

- No CLIV2 response parcels are generated by the database after execution of the triggered action statements.

- The triggered action statements report messages to the requesting user only for aborts and failures.
- The execution of the triggered action statements is atomic, meaning that the SQL transaction cannot be explicitly terminated.

If the execution of such atomic SQL statement is unsuccessful, then the system cancels all the data or structural changes made by the statement.

The action of the triggering statement returns a single response, with the following information:

- Success or failure.
- Activity type of the triggering statement.
- A count of rows changed by the triggering statement.

The following points about some triggered action statements are important:

| IF this statement is specified ... | THEN ... |
|------------------------------------|--|
| INSERT... SELECT | direct references by this statement to the triggering table are treated as outer references. |
| Atomic Upsert | the condition must be on a primary index, whether unique or nonunique. |

Triggers and Referential Integrity

The following trigger cases return a referential integrity violation error to the requestor for all forms of referential integrity:

- The triggering statement is an UPDATE on the child table in the relationship.
- The triggering statement is a DELETE on child table in the relationship.
- The triggers are cascaded.

The following trigger cases for tables with referential integrity relationships are valid if and only if the relationship is defined using batch referential integrity (see [Batch Referential Integrity Constraints](#)):

- The triggering statement is a DELETE operation on the parent table in the relationship.
- The triggering statement is an UPDATE operation on the parent table in the relationship.
- The triggering statement is an INSERT operation on the child table in the relationship.
- The triggering statement is an INSERT operation on the child table in the relationship and defines an INSERT ... SELECT operation as the triggered action statement.
- For cascaded triggers, the system only handles INSERT operations as the triggering statement. For example, an INSERT operation on the parent table and an INSERT on the child table in the relationship.

No other cascaded triggers are valid.

Consider the following table definitions. The tables they define make up a referential integrity checking hierarchy. Batch RI is defined on tables *child_tab* (referring to *parent_tab*) and *grandchild_tab* (referring to *child_tab*).

Note that the foreign keys for these tables are defined implicitly using REFERENCES clauses without specifying FOREIGN KEY, but the referential integrity checking they define is the same as it would be if the keywords FOREIGN KEY had been specified.

Triggers are defined on these tables.

```
CREATE TABLE parent_tab (
  prime_key INTEGER NOT NULL,
  column_2  INTEGER,
  null_col  CHARACTER DEFAULT NULL)
UNIQUE PRIMARY INDEX (prime_key);

CREATE TABLE child_tab (
  prime_key INTEGER NOT NULL,
  null_col  CHARACTER DEFAULT NULL,
  for_key   INTEGER REFERENCES WITH CHECK OPTION parent_tab
            (prime_key))
UNIQUE PRIMARY INDEX (prime_key);

CREATE TABLE grandchild_tab (
  prime_key INTEGER NOT NULL,
  column_3  INTEGER,
  grc_key   INTEGER REFERENCES WITH CHECK OPTION child_tab
            (prime_key))
UNIQUE PRIMARY INDEX (prime_key);
```

Consider the following definitions of the referential levels among these tables and the respective triggering statements that are supported:

| At this level ... | The triggering statement is ... | On this table in the RI relationship ... |
|-------------------|--|--|
| 1 | <ul style="list-style-type: none"> • DELETE or • UPDATE | parent |
| 2 | <ul style="list-style-type: none"> • DELETE or • UPDATE | child |

There are batch referential integrity relationships between the table pairs as listed in the following table:

| At this level ... | There is a batch referential integrity relationship between these tables ... |
|-------------------|---|
| 1 | <ul style="list-style-type: none"> • <i>parent_tab</i> • <i>child_tab</i> |

| At this level ... | There is a batch referential integrity relationship between these tables ... |
|-------------------|---|
| 2 | <ul style="list-style-type: none"> • <i>child_tab</i> • <i>grandchild_tab</i> |

Triggers and Stored Procedures

Stored procedures called from within a triggered action statement can be one or more of the SQL statements summarized in the following table:

| Trigger Type | Valid Statement |
|--------------|--|
| AFTER | <ul style="list-style-type: none"> • ROLLBACK • SELECT AND CONSUME • UPDATE (all forms) • EXEC of macros that contain valid SQL statements |
| BEFORE | <ul style="list-style-type: none"> • SET clause (INSERT and UPDATE row triggers only). • EXEC of macros that contain no data-changing SQL statements |

Triggered statements must run within the same transaction as the statement that fires their trigger. Stored procedures can change the state of a transaction, but the statements within the body of a trigger *cannot* change the transaction state.

Executing the SQL CALL statement does not initiate a transaction: it is the execution of the first SQL request or expression inside the stored procedure that initiates a transaction. The system then assigns a request number and transaction number to it if a transaction number has not already been assigned. The system then increments the request number for each SQL request inside the stored procedure. However, the transaction number of the SQL requests depends on the session mode and the status of the transaction in which the CALL request was submitted.

For this reason, the following statements are *not* permitted inside the body of any stored procedure that is called from a trigger:

- All DDL statements
- All DCL statements
- The BEGIN TRANSACTION and END TRANSACTION statements
- All exception handling statements

This is not strictly true. Although a stored procedure called by a trigger can validly contain condition handlers, most of them can never be executed when the procedure is called because the first failure on a triggered statement causes the entire request to abort and roll back.

In general, stored procedures support the following three types of parameters:

- IN
- OUT

- INOUT

However, because triggered actions cannot return output to the requestor, the OUT and INOUT parameters are *not* allowed in any stored procedure that is called from a trigger.

You can pass the old-values correlation name and the new-values correlation name as parameters, but not the old-values-table alias, new-values-table alias, or old-new-table alias.

Triggers and External Stored Procedures That Make SQL Calls

If an external stored procedure uses CLIV2 or JDBC to submit an SQL statement that is not allowed to be executed from within a trigger (see [Triggered Action Statements](#)), the system returns one of the following:

- A failure message to a CLIV2-based external stored procedure.
- An exception to a JDBC-based external stored procedure.

The system returns the message to the external procedure to provide it with an opportunity to post the error and then to either cleanly close or disconnect any external files or connections it had established. The only remaining course of action is for the procedure to return something to the caller.

The following outcomes occur depending on what the external stored procedure returns.

| IF the external procedure ... | THEN ... |
|--|--|
| returns with its own error | it can set the SQLSTATE to its own exceptions code and the original fail condition will not be known to the caller of the procedure. |
| returns no error (the SQLSTATE value is '00000') | the system returns the original failure to the caller. |
| attempts to submit another request | the system returns an error because the procedure submitted a request after receiving a trigger fail message. |

The system terminates the triggering request in all of these cases.

The following table documents examples of three possible outcomes for the following scenario.

Suppose a trigger is defined as follows.

```
CREATE TRIGGER trig2
AFTER INSERT ON t2
FOR EACH ROW (CALL sqlxspex(2));
```

You then submit the following two requests from the external stored procedure using either CLIV2 or JDBC function calls, where the table named *ThisIsNotATable* does not exist:

```
INSERT INTO t2 VALUES (1,1);
DELETE FROM ThisIsNotATable ALL;
```

Because there is no table named *ThisIsNotATable* in the current database, the system returns an error to the procedure.

| IF the procedure ... | THEN the outcome is as follows ... |
|--|--|
| returns its own error | <p>the error message reports whatever SQLSTATE code and text message is coded in the procedure, for example:</p> <pre>*** Failure 7504 in UDF/XSP/UDM rgs.sqlxspex: SQLSTATE U0001: Statement# 1, Info =0 *** Total elapsed time was 1 second.</pre> <p>Note that the outcome of this example is not determined by the system, but is entirely dependent on how the procedure is coded.</p> <p>The example is designed to show how an external procedure handles the error the system returns to the trigger. In this case, the procedure was coded to set the SQLSTATE to 'U0001' for a 7504 error, but any valid SQLSTATE code and user-written text message could have been used.</p> |
| returns with a SQLSTATE of '00000' | <p>the error message reports the original error that was reported to the external stored procedure:</p> <pre>*** Failure 3807 SQLXSPEx:Object 'ThisIsNotATable' does not exist. Statement# 1, Info =0 *** Total elapsed time was 1 second.</pre> <p>In this case, the original error is the failure of the DELETE request that the procedure attempted to execute against a table that does not exist.</p> |
| ignores the error and attempts to submit another SQL request | <p>the procedure is terminated with the following error message:</p> <pre>*** Failure 7836 The XSP mydbase.sqlxspex submitted a request subsequent to receiving a trigger fail message. Statement# 1, Info =0 *** Total elapsed time was 1 second.</pre> |

Triggers and MERGE Requests

MERGE requests can be used as *triggering* statements to fire triggers; however, they cannot be used as *triggered* statements.

When a MERGE request executes, triggers defined on UPDATE actions become activated if a WHEN MATCHED clause is specified, and triggers defined on INSERT actions become activated if a WHEN NOT MATCHED clause is specified.

The order of activation of UPDATE and INSERT triggers is the same as the order of WHEN MATCHED and WHEN NOT MATCHED clauses in the MERGE statement.

There is a slight deviation between the Vantage implementation of triggers with MERGE and the ANSI SQL:2011 standard for the implementation of triggers with MERGE. The ANSI SQL:2011 standard recommends that triggers be implemented in MERGE requests as follows.

| First Clause Specified When Updates to Rows and Inserts of New Rows in the Target Table Occur | Sequence of Actions During MERGE |
|---|--|
| WHEN MATCHED (UPDATE specification) | <ol style="list-style-type: none"> 1. All BEFORE triggers associated with UPDATE action are applied. 2. The updates specified by the UPDATE specification are applied. 3. Constraints are checked, which may result in executing referential actions. 4. AFTER triggers associated with the UPDATE action are applied. 5. BEFORE triggers associated with the INSERT action are applied. 6. Inserts specified by the INSERT specification are applied. 7. Constraints are checked, which might result in executing referential actions. 8. AFTER triggers associated with the INSERT action are applied. |
| WHEN NOT MATCHED (INSERT specification) | <ol style="list-style-type: none"> 1. BEFORE triggers associated with the INSERT action are applied. 2. Inserts specified by the INSERT specification are applied. 3. Constraints are checked, which may result in executing referential actions. 4. AFTER triggers associated with the INSERT action are applied. 5. All BEFORE triggers associated with the UPDATE action are applied. 6. The updates specified by the UPDATE specification are applied. 7. Constraints are checked, which might result in executing referential actions. 8. AFTER triggers associated with the UPDATE action are applied. |

In contrast, Vantage implements triggers in MERGE requests as follows:

| First Clause Specified When Updates to Rows and Inserts of New Rows in the Target Table Occur | Sequence of Actions During MERGE |
|---|--|
| WHEN MATCHED (UPDATE specification) | <ol style="list-style-type: none"> 1. All BEFORE triggers associated with UPDATE action are applied. 2. All BEFORE triggers associated with INSERT action are applied. 3. The updates specified by the UPDATE specification are applied. 4. Constraints are checked, which might result in executing referential actions. 5. Inserts specified by the INSERT specification are applied. 6. Constraints are checked, which might result in executing referential actions. 7. AFTER triggers associated with UPDATE action are applied. 8. AFTER triggers associated with INSERT action are applied. |
| WHEN NOT MATCHED (INSERT specification) | <ol style="list-style-type: none"> 1. All BEFORE triggers associated with INSERT action are applied. 2. All BEFORE triggers associated with UPDATE action are applied. 3. The updates specified by the UPDATE specification are applied. 4. Constraints are checked, which might result in executing referential actions. 5. Inserts specified by the INSERT specification are applied. 6. Constraints are checked, which might result in executing referential actions. |

| First Clause Specified When Updates to Rows and Inserts of New Rows in the Target Table Occur | Sequence of Actions During MERGE |
|---|--|
| | 7. AFTER triggers associated with INSERT action are applied. 8. AFTER triggers associated with UPDATE action are applied. |

For information about the MERGE statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Triggers and Query Bands

Neither the SESSION nor the TRANSACTION form of the SET QUERY_BAND statement can be specified as a triggered action statement in a trigger definition.

Trigger Support for Teradata Unity

Teradata Unity sends request-specific context information as part of a request that executes a trigger to enable Vantage to change the result of the executed trigger indirectly by substituting a value predefined by Teradata Unity for a non-deterministic result. Vantage makes this context information available to a trigger when it is executed from the default connection for the session.

Consider, for example, the following table and trigger definitions, where Teradata Unity overrides the RANDOM functions specified in the select lists of the triggered INSERT ... SELECT requests specified in the trigger definitions.

```
CREATE TABLE t1 (
  a INTEGER,
  b INTEGER)
PRIMARY INDEX (a);

CREATE TABLE t2 (
  a INTEGER,
  b INTEGER)
PRIMARY INDEX (a);

CREATE TABLE t3 (
  a INTEGER,
  b INTEGER)
PRIMARY INDEX(a);

CREATE TRIGGER trigt1
  AFTER INSERT ON t1 (
```

```

INSERT t2
  SELECT a, RANDOM(200, 300)+1
  FROM t1;);

CREATE TRIGGER trigt2
  AFTER INSERT ON t2 (
    INSERT t3
      SELECT a, RANDOM(200, 300)+1
      FROM t2;);

```

Database Objects That Cannot Be Referenced In A Trigger Definition

You cannot reference any of the following database objects in a trigger definition:

- | | |
|--|--|
| <ul style="list-style-type: none"> • Global temporary tables • Hash indexes • Recursive views | <ul style="list-style-type: none"> • Trace tables • Views • Volatile tables |
|--|--|

SET as a Triggered Action Statement

A SET clause assigns a value to a column, variable, or parameter.

You can use the SET clause in a trigger definition as a triggered SQL statement if the triggering action retains the row in the target table after its action is completed. For example, you can specify a SET clause in UPDATE and INSERT BEFORE row triggers, but not in a DELETE BEFORE row trigger because the DELETE statement removes a row set from the target table.

The following rules apply to the use of the SET clause:

- SET can modify values of a new transition table row, referred to as NEW ROW, but it cannot modify the OLD ROW.
- SET can use values from both the NEW ROW and OLD ROW.
- Each SET clause can assign a value to only one column.
- The scalar expression specified with a SET expression can contain any combination of the following elements:
 - System variables.
 - Functions with arguments that are scalar subexpressions.
 - Scalar, but not aggregate, user-defined functions.
 - Expressions referencing NEW or OLD row columns.
 - Constants.
- If a column name in the SET clause is not qualified explicitly, then the system qualifies that column with the correlation name for NEW row.

- If there is a CHECK column or referential integrity constraint on the column to which the SET assignment clause is applied, then that constraint is enforced only on the final value of the column after all the SET clauses of the BEFORE row trigger are applied.
- You can specify multiple SET clauses that update the same column.
- When the same column is updated more than once in multiple SET clauses, the final value assigned to the column is the value assigned by the last SET clause in the sequence, and the earlier updates are lost.
- You can assign a UDT expression to a UDT column. However, the mutator SET clause syntax is not supported.

To work around this, use the standard non-mutator SET clause syntax with a column reference on one side of the equal sign and a UDT expression that contains mutators on the other side.

Cascading and Recursion of Triggers

Cascading is not itself an element of triggers, but row triggers can create a cascade when a statement fires a trigger, which in turn fires another trigger, and so on. Thus the outcome of one triggering event can itself become another trigger.

The following rules apply to trigger cascading and recursion:

- BEFORE triggers cannot cascade because they have no data-changing statements.
- Trigger recursion, including self-referencing triggers, is valid. In other words, you can make backward references to triggering statements in a cascade of triggers.
- Cascaded row triggers that refer to a transition table are not valid.
- To prevent an infinite recursive loop, the system imposes a limit of 16 cascaded or recursive triggers. This limit is enforced at run time, and the system rolls back requests with cascading of triggers that exceed this limit.

Restrictions on Creating and Using Triggers

The following restrictions apply to the creation and use of triggers:

- You can define triggers only for persistent base tables.
You cannot define triggers for any of the following database objects:
 - Error tables
 - Global temporary tables, including global temporary trace tables
 - Views
 - Volatile tables
- BEFORE statement triggers are not valid.
- BEFORE triggers cannot have data-changing statements as their triggered action (triggered SQL statements).
- You cannot define triggers and hash indexes on the same table.
- You can specify UDT comparisons in the WHEN clause as long as the UDTs have defined orderings.

- If you reference a NEW_TABLE, OLD_TABLE, or OLD_NEW_TABLE transition table in a WHEN condition, the reference *must* be made:

- From a subquery
- Using correlation names for the referenced transition tables

The typical subquery used for this purpose is an aggregate that returns a scalar value.

- Aggregates cannot appear on the left hand side of the search condition specified for a WHEN clause. Aggregates *can* appear on the right hand side of the search condition.
- Positioned (updatable cursor) update and delete operations cannot fire a trigger. An attempt to do so generates an error.

You must disable all triggers defined on a subject table prior to positioning cursors for update or delete operations on it.

- You cannot use an INSERT ... SELECT AND CONSUME statement as either of the following:
 - Triggered action statement.
 - Triggering statement.
- You cannot refer to a recursive view, a WITH clause, or a WITH RECURSIVE clause in the definition of a trigger.

Triggers and Identity Columns

The following rules and restrictions apply to triggers and INSERT operations into tables with identity columns.

- You cannot create a trigger definition if its triggering statement is an INSERT on a subject table that has an identity column and its triggered action statement or WHEN condition references that column.
- If a triggered statement makes row references to the table that is involved in the triggering statement, and the triggered statement is an INSERT operation into a table with a GENERATED ALWAYS identity column, Vantage substitutes a dummy value for the USING value for the referenced row and then generates the actual identity column number in the Dispatcher before sending it to its destination AMP.

Unsupported Trigger Features

Vantage does not support triggers created in Teradata Database 5.0 and earlier that specify any of the following no longer valid features:

- INSTEAD OF as the trigger action time
- BEFORE statement triggers
- BEFORE row triggers with data-changing statements

The system reports an error if old triggers with any of these features are found by a triggering event on a table. Any such trigger must be redefined if it is to be used in the current release.

Restrictions and Limitations for NoPI Tables

You cannot create an UPDATE (Upsert Form) or MERGE trigger on a NoPI table.

Restrictions and Limitations for Normalized Table Columns

The following restrictions apply to using triggers with the columns of normalized tables.

- You cannot specify a normalize Period column for the NEW ROW and NEW TABLE options.
- You can specify a normalize Period column for the OLD ROW and OLD TABLE options.
- Vantage does not fire a trigger on normalized rows for UPDATE or INSERT operations.

Restrictions and Limitations for Load Utilities

You cannot use FastLoad, MultiLoad, or Teradata Parallel Transporter to load data into base tables that have enabled triggers. Otherwise, the system returns an error message and the loading process aborts.

You can load data into the base tables if you first disable all triggers on those tables using the ALTER TRIGGER statement and then run the appropriate data loading utility. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

You can then reenable the disabled triggers after the data loading operation completes.

Load utilities like TPump that perform standard SQL inserts and updates can update tables that have enabled triggers.

Dropping or Deleting Database Objects That Contain Triggers

Vantage imposes restrictions on DROP and DELETE operations on the following database objects to ensure that triggers are not orphaned, as indicated by the following table:

| IF you specify this statement ... | IN this state ... | THE result is ... |
|--|---|--|
| DROP TABLE | triggers are defined on the specified table | an error message. You must first drop all the triggers before the table can be dropped. |
| <ul style="list-style-type: none"> • DELETE DATABASE • DELETE USER | at least one table in the named database or user is the subject table of triggers created in a different database | an error message. You must first drop all such triggers before the database or user can be deleted. |

Triggers, Partitioned Primary Indexes, and the ALTER TABLE Statement

You must disable some triggers before you can perform an ALTER TABLE request that modifies or revalidates the primary index for a table.

| ALTER TABLE Request Modifies or Revalidates the Primary Index with this Option | Disable these Triggers |
|--|--|
| WITH DELETE | All DELETE triggers on <i>table_name</i> . |
| WITH INSERT | <ul style="list-style-type: none"> All DELETE triggers on <i>table_name</i>. All INSERT triggers on <i>save_table</i>. |

You can reenable the triggers after the ALTER TABLE request completes.

Performance Issues for Triggers

Triggers are active database objects. The actions of creating triggers on tables and then executing them involve several system resource issues.

Application designers must evaluate the relative advantages of triggers and the possible performance impact while defining them. For example, a row trigger can complete a much larger amount of work than is done in the triggering statement, which is an SQL DML statement.

There is no limit on the number of rows that can be changed by row triggers or statement triggers. For example:

| THIS type of triggered action ... | RESULTS in ... |
|-----------------------------------|--------------------------------|
| INSERT | adding one row to a table. |
| INSERT... SELECT | adding many rows to a table. |
| UPDATE | updating one or multiple rows. |

In many instances, the actions carried out by triggers are sufficiently useful that their use is justified even after the resulting impact on system performance is taken into consideration.

Guidelines for Creating Triggers

The following guidelines for developing triggers are intended to minimize their performance impact:

- Design the triggering statement so that in normal situations it changes as few rows as possible, preferably a single row or a small number of rows
- Avoid complex cascading actions by row triggers.

For instance, you can specify an appropriate condition in the WHEN clause to limit the number of times the clause is tested.

- Use triggers only where complex processing and validation of data are required.
- Use other alternatives, such as macros and declarative constraints, when they are more advantageous.
- Avoid writing applications that reference tables associated with triggers that change a large number of rows because such applications can impact throughput significantly.
- Avoid writing triggers that add workload to the data-changing statements that are triggered.

For example:

- For statements that change a single row per request, additional action is needed for each trigger fired.
- For statements that might change a large number of rows per request, a row trigger has to make a number of secondary changes for each affected row.
- Avoid writing row triggers that access the triggering table directly or the transition tables through OLD TABLE or NEW TABLE correlation names that require join or aggregation operations because of their potentially high performance impact.
- Compare the respective effects of row and statement triggers for an application before deciding on which to implement.

Rules for Using Scalar Subqueries in Triggers

The following rules apply to specifying scalar subqueries in triggers:

- You can specify an ABORT or ROLLBACK statement with a scalar subquery in the body of a trigger. However, Vantage processes any uncorrelated scalar subquery you specify in the WHERE clause of an ABORT statement in a row trigger as a single-column single-row spool instead of as a parameterized value.
- You can specify a DELETE statement with a scalar subquery in the body of a trigger. However, Vantage processes any uncorrelated scalar subquery you specify in the WHERE clause of a DELETE statement in a row trigger as a single-column single-row spool instead of as a parameterized value.
- You can specify an INSERT statement with scalar subqueries in the body of a trigger.
- You cannot specify a uncorrelated scalar subquery as a value in the multivalue of a simple INSERT in the body of a row trigger.
- Vantage processes any uncorrelated scalar subquery specified in the SELECT component of an INSERT ... SELECT in a row trigger as a single-column single-row spool instead of as a parameterized value.
- You can specify an UPDATE statement with scalar subqueries in the body of a trigger.

However, Vantage processes any uncorrelated scalar subqueries specified in the WHERE or SET clauses of an UPDATE statement in a row trigger as a single-column single-row spool instead of as a parameterized value.

When To Use Row Triggers

Row triggers are a more natural choice than statement triggers for various reasons, including performance and ease of operation.

When a data-changing statement with one or more triggers is executed, the system creates one (for delete or insert operations) or more (for update operations) transition tables before the statement executes.

For row triggers, the system scans the transition table set, accessing one row at a time. You cannot specify the ordering of the transition tables, so do not write triggered actions with outcomes that are intended to differ depending on the order of rows in the transition table.

The volume of work required to execute a row trigger can vary, depending on the information available to the triggered action, as described in the following table:

| Source of data for triggered action | Comment |
|--|--|
| The specific rows being processed | <ul style="list-style-type: none"> For a DELETE or UPDATE, the old value row is accessible if the REFERENCING clause provides a correlation name for the OLD row. For an INSERT or UPDATE, the new value is accessible if the REFERENCING clause provides a correlation name for the NEW row. Fields from these rows can be accessed as scalar values by means of the specified WHEN condition and by triggered action statements. |
| The transition tables composed of all the rows changed by the triggering statement | <ul style="list-style-type: none"> For a DELETE or UPDATE, the old rows are accessible as a temporary table if the REFERENCING clause provides a correlation name for the OLD TABLE. For an INSERT or UPDATE, the new rows are accessible if the REFERENCING clause provides a correlation name for the NEW TABLE. Columns in these tables are accessible by means of the specified WHEN condition and by triggered action statements as outer table references. <p>The typical use of such references is through aggregate statements accessing the correlation names and providing scalar values for use in the WHEN condition and triggered action statements.</p> |
| The full set of rows in the table being changed | <p>This behavior is similar to the behavior described in the transition tables row, but there is no syntax to indicate OLD or NEW values.</p> <ul style="list-style-type: none"> For BEFORE triggers, the rows accessed are the rows of the triggering table before any changes have been made to it. For AFTER triggers, the rows accessed are the rows of the triggering table after changes have been made to it. <p>This information comes from the triggering table and does not require a REFERENCING clause to be accessed.</p> <p>For these kinds of outer references, the typical use of direct references to the triggering table is by means of the following methods:</p> <ul style="list-style-type: none"> Aggregation statements accessing columns using the table name and providing scalar values for use in the WHEN condition. Triggered action statements. |

Advantages of Row Triggers

The most common use of row triggers is with OLD and NEW correlation names that reference current row values as scalar quantities. The following bullets describe the advantages of row triggers:

- The restriction that each row is processed only once guarantees that trigger executions are independent of one another.
- It is not generally necessary to wait for completion of the triggered action for a given row to initiate action for the next row.
- Because OLD and NEW correlation values can be accessed prior to dispatching the triggered action steps, most cases involve sending single AMP steps, so they do not require full table locks.

If the triggered action for a row changes data values in such a way that the WHEN condition search results for some rows might be affected, then the system must wait for completion of the triggered action before proceeding to the next row. A trigger with such characteristics cannot take full advantage of the Teradata parallel architecture.

When to Use Statement Triggers

Statement triggers are often a better choice than row triggers for multirow operations like an INSERT ... SELECT statement because their triggered action is executed only once, while the triggered action for a row trigger is executed once per each qualifying row of the transition table, which can impact performance negatively.

The WHEN condition is tested only once in statement triggers, and it might access information from OLD TABLE or NEW TABLE columns as well as columns from the triggering table.

Because the WHEN condition must provide a single result, aggregation is a typical use for OLD TABLE and NEW TABLE column references.

Although statement triggers fire only once, the join conditions created by WHEN conditions can have a performance impact. Note that specifying an OLD_NEW_TABLE reference, which is valid only for UPDATE AFTER statement triggers, eliminates a join operation and often eliminates the need for WHERE clause filtering conditions.

Triggers and Tactical Queries

Vantage triggers are parallelized. That is, they are implemented as multistatement requests along with the statement that caused them to fire. Being part of a multistatement request means that the triggers and the triggering statement become a single optimizing unit and a single recovery unit.

Performing a trigger can involve few or all AMPs. The presence of triggers can result in a query becoming an all-AMP operation, even if the base query is highly tuned for few AMPs, because both query and trigger share the same optimized plan.

A trigger execution can be a few AMPs operation or it can be an all-AMPs operation. The presence of triggers can cause a query to become an all-AMP operation even if it is highly tuned for few AMPs because the query and the trigger are both bundled into the same request.

Triggered insert, update, and delete statements are single-AMP operations when the qualifying condition helps derive the primary index value of the triggered table. In other cases, the Optimizer might choose Group AMP operations instead of all-AMP triggered statements.

To gain a better understanding of the impact executing the trigger has on system performance, EXPLAIN the triggering statement. In the following example, a trigger has been specified to perform each time a new supplier is added. To get a feeling for the performance impact of the trigger, you should EXPLAIN the INSERT request.

```
CREATE TRIGGER supp_insert AFTER INSERT ON supplier
REFERENCING NEW AS n
FOR EACH ROW
  (INSERT INTO suplog VALUES (n.s_suppkey, n.s_acctbal,
    n.s_nationkey);
);

EXPLAIN
INSERT INTO CAB.supplier
VALUES (353, 'Fosters', '133 Meadow', 13, '3108437272', 0.00, 'new')
```

Vantage returns the following EXPLAIN report, with the relevant text highlighted in bold:

```
Explanation
-----
1) First, we execute the following steps in parallel.
    1) We do an INSERT into CAB.supplier.
    2) We do an INSERT into CAB.suplog.
2) Finally, we send out an END TRANSACTION step to all AMPs involved
    in processing the request.
```

The relevant line of the report is 1.2, We do an INSERT into CAB.suplog.

This trigger is designed to insert a row into a Supplier Log table each time a new Supplier row is added to the database. Both inserts are parallel steps in the query plan, which is highly efficient for tactical queries.

You should always EXPLAIN any single-row update statement that is part of a tactical query, especially if triggers could be involved.

CREATE TYPE (ARRAY/VARRAY Form)

Supported ARRAY and VARRAY Data Type Elements

The element types of an ARRAY or VARRAY UDT must be chosen from any of the Teradata data type categories contained in the following list.

All supported predefined data types with the exception of the following types.

- BLOB
- CLOB
- LOB-based UDT
- Geospatial
- One-dimensional ARRAY types
- Multidimensional ARRAY types

Rules and Restrictions for One-Dimensional ARRAY and VARRAY Data Types

The one-dimensional ARRAY/VARRAY type is defined as a variable-length ordered list of values. It has a maximum number of values that you specify when you create the type. You can access each element value in a one-dimensional ARRAY/VARRAY type using a numeric index value. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

The database supports all Teradata data types for the element type of an ARRAY/VARRAY data type, including UDTs and Period types with the following exceptions:

- BLOB
- CLOB
- LOB UDTs (both distinct and structured types)
- Geospatial

Note that you also cannot specify a one-dimensional ARRAY/VARRAY type as the element type of an ARRAY/VARRAY type.

The following parameter data types are also not supported as element types for one-dimensional ARRAY/VARRAY types.

- VARIANT_TYPE
- TD_ANYTYPE

You can specify a one-dimensional ARRAY/VARRAY type for a table column or for a parameter to a UDF, method, external stored procedure, or SQL stored procedure. You can also use a one-dimensional ARRAY/VARRAY type to declare a local variable inside of an SQL stored procedure. The database supports ANSI-style DML syntax to enable you to access and manipulate the individual elements of a one-dimensional ARRAY/VARRAY value.

The following rules and restrictions apply to creating and using one-dimensional ARRAY and VARRAY data types.

- 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 external 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.
- The Oracle-compatible CREATE TYPE syntax for VARRAY does not support the REPLACE TYPE statement.

Neither does the VARRAY syntax support the optional NOT NULL clause for designating elements as not being null-constrained.

- As is true for structured and distinct UDTs, the database stores any one-dimensional ARRAY/VARRAY type that you create in the *SYSUDTLIB* database, along with its autogenerated constructor UDF, by default. This means that you must have either the UDTTYPE or UDTMETHOD privilege on *SYSUDTLIB* in order to create a one-dimensional ARRAY or VARRAY UDT.
- You can use the optional DEFAULT NULL clause to initialize all the elements of a one-dimensional ARRAY/VARRAY type to null at the time the type is created.

This clause is particularly useful for one-dimensional ARRAY/VARRAY types that are expected to be fully populated.

When you do this, the action prevents all subset operations such as average, update, or count for one-dimensional arrays from aborting and returning an error because they refer to an element that is not initialized.

- The one-dimensional ARRAY/VARRAY type only supports a lower bound of 1. This means that every one-dimensional ARRAY/VARRAY type that is created has a first array element indexed by the number 1. This rule permits the CREATE TYPE one-dimensional ARRAY/VARRAY syntax to be compatible with competitor syntax for arrays, and also to be compliant with the ANSI SQL:2011 standard for the ARRAY type structure.

Note that this is different from multidimensional ARRAY/VARRAY types, which support an optional user-specified lower bound. The CREATE TYPE syntax for one-dimensional ARRAY/VARRAY types is consistent with this rule because it only supports a size value rather than optional explicit lower and upper bounds.

- The maximum size of a one-dimensional ARRAY/VARRAY type, and its autogenerated transform string must not exceed 64 KB because the database stores its data within the row and because the maximum size of its autogenerated transform string is limited to the maximum size of a VARCHAR type, which is 64,000 Teradata Latin bytes.

Although there is no restriction with regards to the number of table columns that can be defined as a one-dimensional ARRAY/VARRAY type, the expectation is that a table will generally not contain more than one column with a one-dimensional ARRAY/VARRAY type, and unless the table is defined as a NoPI table, it will contain other columns, some of which might have smaller data types.

Additionally, the row header for each row consumes an additional set of bytes, so the maximum size of a one-dimensional ARRAY/VARRAY data type is unlikely to exceed approximately 70% of

the maximum row size. To enable the greatest degree of freedom for creators of one-dimensional ARRAY/VARRAY types, the only size restriction that the database enforces is that of the row size limit.

For example, if n is the cardinality of a one-dimensional ARRAY A of integer type, then $A[n]$ cannot have more than $(16,384 - x)$ elements ($1 \leq n \leq (16,384 - x)$), where x represents the number of extra bytes required for in-row storage of the ARRAY type to enable direct access to the individual elements of the array, as well as overhead for displaying the autogenerated transform string for the ARRAY.

- The following actions take place on the data dictionary on a CREATE TYPE request for a one-dimensional ARRAY/VARRAY. The actions listed are in addition to dictionary updates that normally occur for a CREATE TYPE request.
 - The row that is already inserted into *DBC.UDTInfo* for a CREATE TYPE request also populate the columns *DBC.UDTInfo.ArrayNumDimensions* and *DBC.UDTInfo.ArrayScope*.

See *Teradata Vantage™ - Data Dictionary*, B035-1092 for details about the values inserted into this row for a one-dimensional ARRAY type.

- One row is inserted into *DBC.TVFields* to record information about the element type specified for the one-dimensional ARRAY type. The field name for the element type is recorded as the predefined value *_ARRAYELEMENT*, with a field ID of 1025. If the element type specified for the one-dimensional ARRAY type is a UDT, then the name of the UDT element type and its TypeID are also recorded.

See *Teradata Vantage™ - Data Dictionary*, B035-1092 for details about the values inserted into this row for a one-dimensional ARRAY type.

System-Generated Default Functionalities For One Dimensional ARRAY and VARRAY UDTs

When you create a one-dimensional ARRAY/VARRAY UDT, Vantage automatically generates the following additional UDT-related functionality for you.

One-Dimensional Type Transform

This is the fromsql and tosql functionality associated with the transform of a one-dimensional ARRAY/VARRAY type. The one-dimensional ARRAY/VARRAY values are transformed to/from a *VARCHAR(length)* value, where *length* depends on the element type and the total number of elements defined in the one-dimensional ARRAY/VARRAY.

For the formatting of the transformed output, see [Transform Input/Output Strings for ARRAY/VARRAY Types](#).

Constructor Function and Constructor Method

Vantage automatically generates a default constructor function and an additional constructor method for each one-dimensional ARRAY or VARRAY type you create.

The default constructor function takes no parameters and allocates a one-dimensional ARRAY/VARRAY type instance.

Vantage sets all the elements of the one-dimensional ARRAY/VARRAY to an uninitialized state. This is different from setting the elements null.

The system returns an error to the requestor if any element is accessed from the ARRAY/VARRAY.

Vantage sets all of the elements to an uninitialized state only if the one-dimensional ARRAY/VARRAY type has been created without specifying the DEFAULT NULL option.

The constructor method takes one or more parameters, up to the number of parameters as defined by the declared size of the one-dimensional ARRAY/VARRAY data type, and initializes each element of the one-dimensional ARRAY/VARRAY with the corresponding value that is passed to it.

Vantage stores the element values of a one-dimensional ARRAY/VARRAY sequentially, so you must pass the parameters to the constructor in sequential order. The constructor initializes the elements of a one-dimensional ARRAY/VARRAY starting with the first element, moving from left to right.

Although there normally is an upper limit of 128 parameters for a constructor method, the number of parameters you can specify for a one-dimensional ARRAY/VARRAY constructor method is only limited by the maximum number of elements that you have defined for the one-dimensional ARRAY/VARRAY type.

If a one-dimensional ARRAY/VARRAY has n total elements, and you pass more than n parameters to the constructor, the system returns an error to the requestor.

One-Dimensional ARRAY/VARRAY Type Ordering

Vantage provides very basic ordering functionality for a newly created one-dimensional ARRAY/VARRAY type.

Vantage provides this ordering functionality to avoid hashing issues and to enable one-dimensional ARRAY/VARRAY types to be allowed as columns in SET tables.

You cannot make relational comparisons of one-dimensional ARRAY/VARRAY values.

You cannot use any of the relational comparison operators on one-dimensional ARRAY data, nor can you specify a one-dimensional ARRAY/VARRAY column in any of the following SQL DML clauses:

- INTERSECT set operator
- MINUS set operator
- UNION set operator
- DISTINCT operator
- WHERE clause
- ORDER BY clause
- GROUP BY clause
- HAVING clause

One-Dimensional ARRAY/VARRAY Type Casting

Vantage provides casting functionality for a one-dimensional ARRAY/VARRAY type. The system provides two autogenerated casts for all one-dimensional ARRAY/VARRAY types:

- VARCHAR to ARRAY/VARRAY

- ARRAY/VARRAY to VARCHAR

The format of the VARCHAR string as both the case source and target value is the same as the format of the tosql and fromsql transforms. For more information about the formats of these strings, see [Transform Input/Output Strings for ARRAY/VARRAY Types](#).

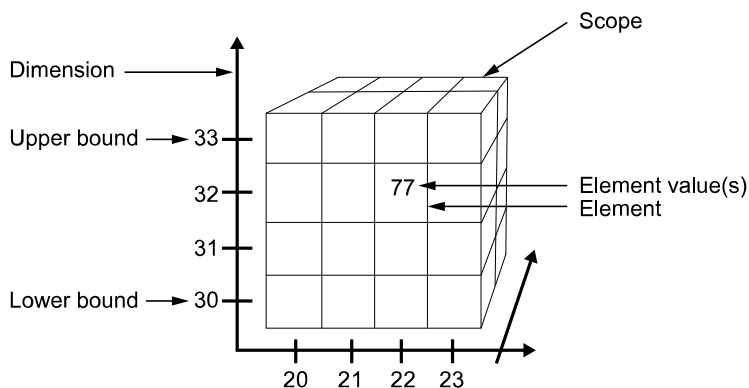
You can also create your own UDFs to perform casting operations between the elements of two one-dimensional ARRAY/VARRAY types.

Rules and Restrictions for Multidimensional ARRAY and VARRAY Data Types

A multidimensional ARRAY is a mapping from integer coordinates to an element type. A multidimensional ARRAY is described by a pair of lower and upper bounds for each of its dimensions.

You must create each multidimensional ARRAY/VARRAY type you need for your applications using a CREATE TYPE (ARRAY form) request. When you create a new multidimensional ARRAY/VARRAY type, you must explicitly specify the lower and upper boundaries for each dimension, as well as the element data type of the array. The element data type must be an existing Teradata SQL type. Once a multidimensional ARRAY/VARRAY type has been created, you can use it in much the same way as any other Teradata SQL data type.

The following graphic shows the constituents of a multidimensional ARRAY/VARRAY data type.



The multidimensional ARRAY/VARRAY type is defined as a variable-length ordered list of values. It has a maximum number of values for each of its dimensions that you specify when you create the type. You can access each element value in a multidimensional ARRAY/VARRAY type using a numeric index value. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Vantage supports all Teradata data types for the element type of a multidimensional ARRAY/VARRAY data type, including UDTs and Period data types with the following exceptions:

- BLOB
- CLOB
- LOB UDTs (both distinct and structured types)
- Geospatial

The following parameter data types are also not supported as element types for multidimensional ARRAY/VARRAY types.

- VARIANT_TYPE
- TD_ANYTYPE

You can specify a multidimensional ARRAY/VARRAY type for a table column, or for a parameter to a UDF, method, external stored procedure, or SQL stored procedure. You can also use a multidimensional ARRAY/VARRAY type to declare a local variable inside of an SQL stored procedure. Vantage supports an extended ANSI-style DML syntax to enable you to access and manipulate the individual elements of a multidimensional ARRAY/VARRAY value.

The following rules and restrictions apply to creating and using multidimensional ARRAY and VARRAY data types.

- 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.
- As is true for structured and distinct UDTs, the database stores any ARRAY type that you create in the *SYSUDTLIB* database along with its autogenerated constructor UDF, by default.

Because of this, you must have the appropriate UDT privileges on *SYSUDTLIB* to create an ARRAY or VARRAY type. This means that you must have either the UDTTYPE or UDTMETHOD privilege on *SYSUDTLIB* to create an ARRAY or VARRAY UDT.

- You can use the optional DEFAULT NULL clause to initialize all the elements of a multidimensional ARRAY/VARRAY type to null at the time the type is created.

This clause is particularly useful for multidimensional ARRAY/VARRAY types that are expected to be fully populated.

When you do this, the action prevents all subset operations such as average, update, or count for multidimensional arrays from aborting and returning an error because they refer to an element that is not initialized.

- There are two ways to specify the array boundaries for a multidimensional ARRAY/VARRAY type. You can specify the dimensions of a multidimensional ARRAY/VARRAY type using any combination of these two methods.

Regardless of the specification method you use, an attempt to access an element that is outside the specified domain of a multidimensional ARRAY returns an error to the requestor.

The two methods of specifying array boundaries for a multidimensional ARRAY/VARRAY type are as follows.

- Explicitly specify lower and upper bounds for each dimension, separating the two with a colon.

The lower bound must be less than or equal to the upper bound.

These bounds are signed integer numbers, which means that you can specify negative numbers using this form. The upper and lower bounds are stored as 32-bit integer values internally.

Therefore, the numeric range available to you for the lower and upper bounds corresponds to the maximum ranges available for an integer value, -2147483648 to 2147483647.

You can use this range of values with the understanding that the total size of the array must be less than or equal to 64K - x, where x represents the number of extra bytes required for in-row storage of the ARRAY/VARRAY type to enable direct access to individual elements of the array and overhead for displaying the autogenerated transform string for the ARRAY/VARRAY. The size of the array is influenced not only by the number of elements, but also by the element type of the array.

- Specify a single unsigned integer value to signify the maximum size of the dimension using ANSI-style syntax.

ANSI-style syntax implicitly defines the lower bound of the array to be 1.

- The maximum size of a type created as a multidimensional ARRAY/VARRAY type and its autogenerated transform string must not exceed 64 KB because the database stores its data within the row and because the maximum size of its autogenerated transform string is limited to the maximum size of a VARCHAR type, which is 64,000 Teradata Latin bytes.

Although there is no restriction on the number of columns of a table that can be defined as a multidimensional ARRAY/VARRAY type, the expectation is that a table will generally not contain more than one column with a multidimensional ARRAY/VARRAY type, and unless the table is defined as a NoPI table, it will contain other columns, some of which might have smaller data types.

Additionally, the row header for each row consumes an additional set of bytes, so the maximum size of a multidimensional ARRAY/VARRAY type is unlikely to exceed approximately 70% of the maximum row size. To enable the greatest degree of freedom for creators of multidimensional ARRAY/VARRAY types, the only size restriction that the database enforces is the row size limit.

For example, if m and n are the cardinalities for dimensions 1 and 2 respectively, of a 2-dimensional ARRAY A of integer type, then $A[m][n]$ cannot have more than $(16,384 - x)$ elements ($1 \leq m \cdot n \leq (16,384 - x)$), where x represents the number of extra bytes required for in-row storage of the ARRAY type to enable direct access to the individual elements of the array and the overhead for displaying the autogenerated transform string for the ARRAY.

There is a limit to the maximum number of dimensions that can be declared within the scope of the array. The minimum number of dimensions that you can create is two, and the maximum number of dimensions that you can create is five.

- The following actions take place on the data dictionary on a CREATE TYPE request for a multidimensional ARRAY type. The actions listed are in addition to the dictionary updates that normally occur for a CREATE TYPE request.
 - The row that is already inserted into *DBC.UDTInfo* for a CREATE TYPE request also populates the columns *DBC.UDTInfo.ArrayNumDimensions* and *DBC.UDTInfo.ArrayScope*.

See *Teradata Vantage™ - Data Dictionary*, B035-1092 for details about the values inserted in this row for a multidimensional ARRAY type.

- One row is inserted into *DBC.TVFields* to record information about the element type specified for the multidimensional ARRAY type. The field name for the element type is recorded as the predefined value *_ARRAYELEMENT* with a field ID of 1025. If the element type specified for the multidimensional ARRAY type is a UDT, then the name of the UDT element type and its TypeID are also recorded.

See *Teradata Vantage™ - Data Dictionary*, B035-1092 for details about the values inserted in this row for a multidimensional ARRAY type.

System-Generated Default Functionalities for Multidimensional ARRAY and VARRAY UDTs

Once you have created a multidimensional ARRAY/VARRAY UDT, Vantage automatically generates the following additional UDT-related functionality for you.

- A multidimensional type transform.

This is the fromsql and tosql functionality associated with the transform of a multidimensional ARRAY/VARRAY type. The multidimensional ARRAY/VARRAY values are transformed to/from a *VARCHAR(length)* value, where length depends on the element type and the total number of elements defined in the multidimensional ARRAY/VARRAY.

See [Transform Input/Output Strings for ARRAY/VARRAY Types](#) for the formatting of the transformed output.

- A constructor function and constructor method.

Vantage automatically generates a default constructor function and an additional constructor method for each multidimensional ARRAY/VARRAY type you create.

- The default constructor function takes no parameters and allocates a multidimensional ARRAY/VARRAY instance in the database with the scope defined by the cross product of all interval expressions in the multidimensional ARRAY/VARRAY data type, where each interval expression has this syntax:

```
[ { lower_bound : upper_bound } | maximum_size ]
```

Note:

You must type the colored or bold brackets.

lower_bound

Lower bound for the multidimensional array, an INTEGER value.

upper_bound

Upper bound for the multidimensional array, an INTEGER value.

maximum_size

Maximum size of the multidimensional array, up to 64,256 bytes.

Vantage sets all the elements of the multidimensional ARRAY to an uninitialized state. This is different from setting the elements null.

The system aborts the request and returns an error to the requestor if any element is accessed from the ARRAY/VARRAY.

Vantage sets all of the elements to an uninitialized state only if the multidimensional ARRAY/VARRAY type has been created without specifying the DEFAULT NULL option.

- The constructor method takes one or more parameters, up to the number of parameters defined by the cross product of all of the interval expressions in the multidimensional ARRAY/VARRAY data type, where the interval expressions are either in the form [lower_bound : upper_bound] or are in the form [maximum_size], and initializes each element of the multidimensional ARRAY/VARRAY with the corresponding value that is passed to it. See the table in the previous bulleted item for the definitions of the multidimensional array boundary variables.

The database stores the element values of a multidimensional ARRAY/VARRAY in row-major order. Row-major means that the first dimension, which is leftmost in the scope specification, is the major dimension, and as you move toward the last dimension, the dimensions become less and less major. This applies to arrays of an arbitrary number of dimensions. Most modern computer languages use row-major storage.

The constructor method initializes the elements of the multidimensional ARRAY/VARRAY starting with the first element as defined in the scope specification and proceeds in row-major order through the elements.

Although there normally is an upper limit of 128 parameters for a constructor method, the number of parameters you can specify for a multidimensional ARRAY/VARRAY types is only limited by the maximum number of elements that can be defined for the multidimensional ARRAY/VARRAY type. If a multidimensional ARRAY/VARRAY has n total elements, and you pass more than n parameters to the constructor method, the system aborts the request and returns a message to the requestor.

- A multidimensional ARRAY/VARRAY type ordering.

Vantage provides basic ordering functionality for a newly created multidimensional ARRAY/VARRAY type.

Vantage provides this ordering functionality to avoid hashing issues and to enable multidimensional ARRAY/VARRAY types to be allowed as columns in SET tables.

You cannot make relational comparisons of multidimensional ARRAY/VARRAY values, though you can make relational comparisons of the individual elements of a multidimensional ARRAY/VARRAY type. See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for details about how to do this.

You cannot specify a multidimensional ARRAY/VARRAY column in any of the following SQL DML clauses.

- INTERSECT set operator
- MINUS set operator
- UNION set operator
- DISTINCT operator
- WHERE clause
- ORDER BY clause
- GROUP BY clause
- HAVING clause
- A multidimensional ARRAY/VARRAY type casting.

Vantage provides casting functionality for a multidimensional ARRAY/VARRAY type. The system provides two autogenerated casts for all multidimensional ARRAY/VARRAY types.

- VARCHAR to ARRAY/VARRAY
- ARRAY/VARRAY to VARCHAR

The format of the VARCHAR string as both the cast source and target value is the same as that of the `tosql` and `fromsql` transforms. See [Transform Input/Output Strings for ARRAY/VARRAY Types](#) for more information about the formats of these strings.

You can also create your own UDFs to perform casting operations between all or a subset of the elements of two multidimensional ARRAY/VARRAY types.

Unexpected Row Length Errors for UTF-8 Session Character Set Data

A problem that is sometimes seen with ARRAY/VARRAY type columns is unexpected row length errors when dealing with UTF-8 character data. This problem only occurs when the ARRAY/VARRAY type is defined with a CHARACTER or VARCHAR element type. For example, suppose you create table `arrayc4` that includes an ARRAY data type column. You insert data into the table that contains data imported from a UTF-8 client session character set. No problems occur when you insert these rows into the table. But then when you attempt to select the entire row, again with a UTF-8 session character set, the system returns the following error.

```
SELECT *
FROM arrayc4;

*** Failure 3577 Row size or Sort Key size overflow.
        Statement# 1, Info =0
*** Total elapsed time was 1 second.
```

This error does not occur in all cases. It depends on how large the arrays are and how much internal storage they require.

To summarize, the impact of this is that you can create a table, insert data into it, and then have client utilities such as FastExport fail when they attempt to export all of the columns in the table to a client system.

Similar problems can sometimes be seen in other situations in Vantage with both character and non-character data. It is not exclusive to UTF-8 character data.

Some of the possible workarounds for dealing with this problem are listed below.

- You can modify the default export width for the user who is experiencing this problem. The new export width must be one that does not increase the row size from its server storage size when it exports data to a client.

A possible problem with this workaround is that the export width applies to each array element individually, so elements that require expansion might be truncated even though other elements that contract are present so the complete row size would be correct.

- There are two approaches to working around the problem for character string data.
 - Reduce the number of columns the request generates.

This includes reducing or eliminating the number of columns that Vantage generates internally, but does not return to the user. An example is the data rows to which the system appends the BYNET sort key. While Vantage does append the sort key to the row physically, generating a temporary column that exists in a spool, that sort key is no longer appended to the row when the system returns the sorted data to the requestor.

If appending the BYNET sort key to a row causes a row length error to occur, you could try to sort by the first n characters of the character string instead of sorting by all of the characters in the string. This applies to the transformed format for the array, which is VARCHAR.

- Reduce the size of some or all of the columns being generated.

Whether this is possible or not depends on how strictly your physical database design must follow your logical design. If experience indicates that smaller data types (such as SMALLINT in place of INTEGER) or narrower character data type widths can be used, then experiment with that to reduce or eliminate row size errors.

- For non-character data, the optimal solution to row size errors is to reduce the number of columns returned by queries that are experiencing problems with unexpected row size errors.

It is not possible to suggest more explicit workarounds to this problem without knowing the details of your database schema.

Transform Input/Output Strings for ARRAY/VARRAY Types

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for information about the transform input/output strings for ARRAY and VARRAY types.

ARRAY and VARRAY Columns in SELECT Requests

You cannot specify an ARRAY or VARRAY column in the ORDER BY, GROUP BY, or HAVING clauses of a SELECT request.

ARRAY and VARRAY Columns in an Index Definition

You cannot specify an ARRAY or VARRAY column in a primary, secondary, hash, or join index.

Related Information

See the following topics and documents for information related to the ARRAY and VARRAY types.

- *Teradata Vantage™ - Data Types and Literals*, B035-1143
- “CREATE TYPE (ARRAY/VARRAY Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

CREATE TYPE (Distinct Form)

Internal And External Representations Of Distinct UDTs

The internal and external representations of distinct UDTs are indicated in the following table:

| FOR this type of representation ... | A distinct UDT is stored in the following format: |
|-------------------------------------|---|
| internal | the same as its underlying predefined data type. |
| client | the same as its underlying predefined data type. This is handled by the system-generated transform for the UDT that is created when the distinct UDT is created. |

See the appropriate Teradata Tools and Utilities documentation for details of how a particular client application deals with distinct UDTs.

Similarities With The Syntax For CREATE TYPE (Structured Form)

The syntax for CREATE TYPE (Distinct Form) is similar to that for the CREATE TYPE (Structured Form) statement.

The major distinguishing characteristics for distinct types are the following:

- The AS *predefined_data_type* clause.
For CREATE TYPE (Structured Form), the corresponding clause is AS *attribute_name* list with an optional INSTANTIABLE specification.
- The mandatory FINAL specification.
- The inability to create a user-defined constructor method signature for a distinct type.

Structured UDTs allow you to define constructor method signatures.

Character Set Issues for Naming Distinct UDTs

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 external 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.

Naming Conventions: Avoiding Name Clashes Among UDFs, UDMs, and UDTs

For UDF, UDM, and UDT names:

- The DatabaseID, TVMName column pair must be unique within the DBC.TVM table. Use the DBC.Tables2 view to view rows from DBC.TVM.
- The signature of the *database_name.routine_name(parameter_list)* routine must be unique.

UDFs, UDMs, and UDTs can have the same SQL names as long as their SPECIFIC names and associated routine signatures are different. In the case of UDTs, the SPECIFIC name reference is to the SPECIFIC names of any method signatures within a UDT definition, not to the UDT itself, which does not have a SPECIFIC name.

The value of *database_name* is always SYSUDTLIB for UDFs associated with UDTs, including UDFs used to implement the following functionality on behalf of a UDT:

- Casts
- Orderings
- Transforms

The Database ID column entry is always the same. The name uniqueness is dependent on the TVMName column value only.

TVMName Entry for UDTs and UDMs

The following describes the TVMName entry for UDTs and UDMs.

UDTs created by a CREATE TYPE request have a SPECIFIC name that is system-generated based on the specified UDT name. For information on object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The database generates the SPECIFIC name by truncating any UDT names that exceed the permitted number of characters.

When you submit a CREATE TYPE statement, the system automatically generates a corresponding UDF to obtain a UDT instance.

The SPECIFIC name of a UDT, UDM or UDF must be unique to avoid name clashes.

| Type of External Routine | SPECIFIC Name |
|--|--|
| UDT | always its UDT name. |
| <ul style="list-style-type: none"> • UDM • UDF | <p>user-defined.</p> <p>There are two exceptions to this rule for structured types:</p> <ul style="list-style-type: none"> • System-generated observer methods. • System-generated mutator methods. <p>The SPECIFIC names for these are always the same as the names of the structured UDT attributes on which they operate.</p> <p>Note that for UDMs, the SPECIFIC name is defined in its signature within its associated UDT, not within the CREATE METHOD statement.</p> |

The signatures of external routines must be unique. The following rules apply:

- For every UDT you create, the system generates a UDF with the following signature: `SYSUDTLIB.UDT_Name()`.
No other UDF can have this signature.
- When you create a UDM, the system treats it as a UDF for which the data type of its first argument is the same as the UDT on which that UDM is defined.
For example, a UDM named `area()`, defined on the UDT named `circle`, would have the signature `SYSUDTLIB.area(circle)`. It follows that there can be no other UDF with this same signature.

For a single database (SYSUDTLIB) Teradata UDT environment, the following rules apply:

- A UDT and a SYSUDTLIB UDF with no parameters cannot have the same name.
- A method for a structured UDT cannot have the same name as any of the attributes for that type if the signature of the method and the signature of either the observer or mutator methods for the attribute match.

You must define a transform group for each UDT you create. Because the system creates a transform group for you automatically when you create a distinct UDT, you cannot create an additional explicit transform group without first dropping the system-generated transform. The names of UDT transform groups need not be unique, so you can use the same name for all transform groups.

The names of transform groups are stored in `DBC.UDTTransform`.

The system adds SPECIFIC method names to the `TVMName` column in `DBC.TVM` for:

- Observer and mutator methods, which are auto-generated for structured type attributes.
- Instance methods and constructor methods created by `CREATE TYPE`, `ALTER TYPE`, or `CREATE METHOD` statements where the coder does not specify a SPECIFIC method name.

A SPECIFIC name of up to 28 characters is generated based on the UDT and attribute names.

The SPECIFIC name is generated as the concatenation of the following elements in the order indicated:

1. The first 8 characters of the UDT name.
2. A LOW LINE (underscore) character.
3. The first 10 characters of the for observer attribute name, mutator attribute name, instance method name, or constructor method name, as appropriate.
4. A LOW LINE (underscore) character.
5. The last 8 HEXADECIMAL digits of the routine identifier assigned to the observer, mutator, instance, or constructor method name, as appropriate.
6. A character sequence is appended, `_O` for observer, `_M` for mutator, `_R` for instance, or `_C` for constructor, appropriate.

The remaining characters, up to the 30th character, are filled with SPACE characters.

Function of Distinct UDTs

The concept of distinct UDTs is analogous to the `typedef` construct in the C and C++ programming languages. Unlike predefined SQL data types, however, which are only moderately strongly typed, distinct

UDTs are strongly typed, which makes them ideal for enforcing domains. Note that you *cannot* declare CHECK constraints on UDT columns, so the UDT-as-domain concept cannot be fully implemented.

For example, you might want to define *euro* and *USDollar* as two different distinct UDTs even though both are internally represented as DECIMAL numbers. If *euro* and *USDollar* are defined as separate distinct data types, they cannot be compared directly or used interchangeably. In this example, a *euro* value cannot be compared with or stored in a *USDollar* type column.

A distinct UDT shares its internal representation with a predefined data type (referred to as its source type), but it is considered to be a separate and incompatible type for most operations. You cannot define a distinct UDT based on any of the Period data types. Distinct UDTs encapsulate their underlying predefined data type. This means that operations and behaviors associated with the underlying data type are not valid, and users of a distinct UDT are only able to interact with its encapsulated value by means of user- or system-defined transforms, orderings, and castings, and manipulations by means of instance methods (see [CREATE TRANSFORM and REPLACE TRANSFORM](#), [CREATE ORDERING and REPLACE ORDERING](#), [CREATE CAST and REPLACE CAST](#), and [CREATE METHOD](#), respectively).

You can drop any of the system-generated semantics for a UDT (see [System-Generated Default Functionalities For Distinct UDTs](#)) and replace them with your own definitions using the following SQL statements:

- DROP CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- DROP ORDERING (see [DROP ORDERING](#))
- DROP TRANSFORM (see [DROP TRANSFORM](#))

Vantage supports the definition of instance methods for distinct UDTs, which is an extension to the ANSI SQL standard. For example, you can define an instance method named *round()* to use with a new distinct UDT.

System-Generated Default Functionalities For Distinct UDTs

Once you have created a distinct UDT, Vantage automatically generates the following additional UDT-related functionality for you.

- A transform group with one transform that maps the distinct UDT to its source data type for use by client applications, and another that maps the source data type to its corresponding distinct UDT for Vantage applications.

The system assigns a name to the generated transform group in the format `TD_Identifier`, truncated to 30 characters, where *Identifier* is the unqualified part of the distinct UDT name.

This transform group contains both `tosql` and `fromsql` functionality, supporting the import and export of the UDT between the client system and the database.

You can drop or replace this system-generated transform functionality using the DROP TRANSFORM and REPLACE TRANSFORM statements, respectively (see [DROP TRANSFORM](#) and [CREATE TRANSFORM and REPLACE TRANSFORM](#)).

The system-generated fromsql functionality converts a distinct data type value to its corresponding source data type value.

Conversely, the system-generated tosql functionality converts a source data type value to a distinct data type value. The external type of a distinct data type is, by default, its source data type.

If a distinct UDT has a defined transform group, whether system-generated or user-defined, and the column list of a SELECT statement contains a distinct UDT column or expression, the system automatically converts the distinct type to its external type by means of its defined fromsql transform routine.

- A full map ordering to support comparisons if the source type is not a LOB.

Unless the source type is a LOB, the system automatically create an ordering such that when two values of a distinct type are compared, the corresponding source type values are compared. For example, when two *euro* values are compared, the result is the same as comparing two DECIMAL values.

When the system compares two distinct UDT values, it pursues the following process:

1. Convert the UDTs to their source data types.
2. Compare the source data types.

The result of this comparison is used as the result of the distinct UDT comparison.

You cannot compare a distinct UDT column or expression directly with a source type value. Either the distinct UDT value or the source type value must first be converted to the other by means of explicit built-in casts.

Similarly, you cannot compare different distinct types directly; you must first convert one of the distinct UDTs to the other type by means of a user-defined cast.

As in the case of the system-generated transform functionality, you can drop or replace the ordering functionality the system generated by default and replace it with your own transforms.

When the source type is a LOB, special rules apply. Suppose that the source type of a distinct type named *image* is BLOB. By default, no comparison operation is supported because you cannot compare LOB data types. Note, however, that although no system-generated ordering functionality is produced, you must still define an ordering for *image* before you can use it to define the column type of any table. An attempt to use a newly created data type that has no ordering defined as the type of any column of any table returns an error to the requestor.

Because UDTs follow the rules of strong typing, you cannot directly compare a distinct type value with a value that has its source data type. Instead, you must explicitly cast the source type value to the distinct UDT using the system-generated cast, or vice versa, before you perform a comparison between them.

See [CREATE ORDERING and REPLACE ORDERING](#) for more information about orderings.

- A cast for converting a distinct UDT value to a source type value.

You can invoke this cast either implicitly or explicitly during assignment operations. The syntax for explicitly invoking this cast is as follows:

```
CAST (distinct_UDT_expression AS predefined_data_type).
```

An assignment operation is defined as any of the following SQL operations:

- Inserts
- Updates
- Parameter passing

You can optionally define additional casting operations and drop or replace the system-generated casts

See [CREATE CAST and REPLACE CAST](#) for more information about casts.

- A cast for converting a source type value to a distinct UDT value.

You can invoke this cast either implicitly or explicitly during assignment operations. The syntax for explicitly invoking this cast is as follows:

```
CAST (predefined_type_expression AS distinct_UDT_name).
```

An assignment operation is defined as any of the following SQL operations:

- Inserts
- Updates
- Parameter passing

You can optionally define additional casting operations and drop or replace the system-generated casts

Note:

You can drop transforms, orderings, and casts only if the associated UDT is not being used as the column data type in any table. If you attempt to drop the ordering or transform functionality for a UDT being used as a table column type, the system returns an error to the requestor.

You *can* replace the transforms, orderings, and casts associated with a UDT without having to first drop any columns that have the type.

Casting Character String Attributes

When the source UDT or string being manipulated by a cast operation is a VARCHAR or CHARACTER type, the server character set that is associated with that source type must be one of the following:

- The server character set that was declared for that source type within the CREATE TYPE declaration that created the distinct type UDT.
- The character set that was in effect at the time that the CREATE TYPE declaration was executed.

Enabling Full Functionality For a Distinct UDT

When you create a distinct UDT, Vantage creates all the objects or definitions listed as Yes under the Mandatory? column in the following table to enable the full functionality of the type.

| Functionality | Defining DDL Statement | Purpose | Mandatory? |
|--|--|---|--|
| Definition of a structured UDT. | CREATE TYPE (Structured Form) | Declares the body and attributes of a structured UDT. | Yes |
| Definition of an instance method. | CREATE METHOD | Declares any SQL-invoked instance methods to be associated with the structured UDT. | No |
| Definition of the ordering for comparing structured UDT values. | CREATE ORDERING | Registers the UDF or method to be used as an ordering routine to compare the structured UDT value with other values. | Yes |
| Declaration of the external ordering routine referenced by CREATE ORDERING. | <ul style="list-style-type: none"> CREATE FUNCTION or CREATE METHOD | Declares the ordering routine referenced in the CREATE ORDERING definition. | Yes |
| Transformation of a UDT between client and server data types in both directions. | CREATE TRANSFORM | Registers the UDF and UDF or method to be used as a tosql() and fromsql() transform routine to pass the structured UDF in both directions between the client and server. | Yes |
| Declaration of the external transformation routine referenced by CREATE TRANSFORM. | <ul style="list-style-type: none"> CREATE FUNCTION or CREATE METHOD | Declares the tosql() and fromsql() external routines referenced in the CREATE TRANSFORM definition. | Yes |
| Registration of an external UDF or method casting routine. | CREATE CAST | Registers a UDF or method to be used as a casting routine to cast: <ul style="list-style-type: none"> from one structured UDT to another structured UDT. from a structured UDT to a predefined data type. from a predefined data type to a structured UDT. | Yes A pair of implicit casts is necessary to achieve full functionality for a distinct UDT: one that casts from the UDT to its fromsql transform target type and one that casts from its tosql transform source type to the UDT. Any additional |

| Functionality | Defining DDL Statement | Purpose | Mandatory? |
|--|---|--|--------------------------------|
| | | | cast definitions are optional. |
| Declaration of the external casting routine referenced by CREATE CAST. | <ul style="list-style-type: none"> • CREATE FUNCTION or • CREATE METHOD | Declares the external casting routine referenced in the CREATE CAST statement. | No |

SYSUDTLIB Database

SYSUDTLIB is a system database that contain all UDTs, their methods, cast routines, ordering routines, transform routines, and any UDFs that are used as cast routines, ordering routines, or transform routines. Because SYSUDTLIB is a system database, it is created by a DIP script. SYSUDTLIB is created as a database, not a user, so you cannot log on as user SYSUDTLIB.

When the system resolves SQL requests that involve UDTs, it only searches SYSUDTLIB for UDTs and UDT-related database objects. Because of this, you need not explicitly specify a database name for UDTs. If you do specify a database name, however, it must be SYSUDTLIB.

You should not place non-UDT-related objects in SYSUDTLIB. The database requires space for every UDT you define as well as for every method or UDT-related UDF created within it. SYSUDTLIB space usage is dependent primarily on the space required to store the code files implementing the UDM or UDF, or the space required to contain internally generated source code that is used to acquire an instance of a UDT.

The initial space allocation for SYSUDTLIB is 0. The DBA must modify the definition for SYSUDTLIB to assign enough space that UDTs, UDMs, and UDT-related UDFs can be created within it. If you attempt to create a UDT-related object in SYSUDTLIB when there is insufficient space to do so, the system returns an error to the requestor. As time passes, you might need to increase the storage associated with SYSUDTLIB to accommodate the growing number of UDTs, UDMs, and UDT-related UDFs.

For details on modifying the definition for *SYSUDTLIB*, see [MODIFY DATABASE](#).

External Routines

ANSI SQL defines the following family of SQL-invoked routines:

- Procedures.
- User-defined functions.
- User-defined methods.

The ANSI SQL:2008 standard categorizes these SQL-invoked routines as either external routines or SQL routines.

An external routine is a user-defined function (UDF), method (UDM), or procedure that is written in an external, that is, non-SQL language such as C or C++, while SQL routines are written entirely in the SQL language.

When an external routine associated with a UDT is invoked, the system passes a handle to the UDT argument instead of passing the UDT value. Given the handle, the external routine can either retrieve or set the value of a UDT argument by means of a set of library functions provided by Teradata for this purpose. For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Tasks You Must Perform To Create A Distinct UDT

Because distinct types are created with inherent transform, ordering, and casting capabilities (see [System-Generated Default Functionalities For Distinct UDTs](#)), you need not take any action to create or activate any of these capabilities. These various functionalities are generated silently and activated at the time you submit the CREATE TYPE request.

This means that to create a minimal, fully functional distinct UDT, you need submit only a CREATE TYPE request. The following associated functionalities are then generated for you automatically by the system:

- An ordering to enable two distinct UDTs to be compared
- A transform group to enable the current distinct UDT to be passed between the client and TeVantage.
- A cast to enable casting, including implicit casting on assignment, from the underlying predefined data type to the distinct UDT and from the distinct UDT to the underlying predefined data type.

If your applications of the current distinct UDT require additional functionality, you can write instance methods in either the C or C++ languages, and then submit CREATE INSTANCE METHOD statements to declare any SQL-invoked methods to be associated with their distinct UDT.

You can define your own ordering, transform, and casting functionalities.

You must perform the following high-level general procedure.

1. Drop the system-generated casts, orderings, or transforms you want to replace using the appropriate SQL DROP statements from the following list:
 - DROP CAST (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*).
 - DROP ORDERING (see [DROP ORDERING](#)).
 - DROP TRANSFORM (see [DROP TRANSFORM](#)).
2. Code the appropriate external routines in either C or C++ (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147)
3. Create new casts, ordering, or transforms for the UDT using the appropriate SQL CREATE statements from the following list:
 - CREATE CAST (see [CREATE CAST and REPLACE CAST](#)).
 - CREATE ORDERING (see [CREATE ORDERING and REPLACE ORDERING](#)).
 - CREATE TRANSFORM (see [CREATE TRANSFORM and REPLACE TRANSFORM](#)).

Method Signatures

Unlike other SQL database objects, the definition for a method is split between CREATE METHOD (see [CREATE METHOD](#)) and its signature as defined in its associated CREATE TYPE statement.

Unlike a UDF definition, the following components of a method are defined only in its signature:

- SPECIFIC NAME clause
- LANGUAGE clause
- PARAMETER STYLE clause
- [NOT] DETERMINISTIC clause
- SQL data access (NO SQL) clause

Note:

The RETURNS clause is defined both in the CREATE METHOD definition *and* in the CREATE TYPE signature for the method.

Effect of Platform on Storage Size of Distinct UDTs

The size of a distinct UDT value is often larger for byte-aligned format systems than it is for byte-packed format systems. This depends on the predefined type used to define the distinct UDT. The size of some predefined types, such as all the numeric types, is not affected by the underlying platform.

The size of a distinct UDT on a byte-packed format platform is never larger than the same underlying type on byte-aligned format systems. For details, see *Teradata Vantage™ - Database Design*, B035-1094.

Period and Geospatial Data Types Not Supported For Distinct UDTs

You cannot create distinct UDTs from Period or Geospatial data types.

Migrating or Copying a UDT to a Different System

The object code (including libraries) supporting a UDT must first be installed on the destination system if the type is migrated or copied from one system to another.

Related Information

| For more information on ... | See ... |
|-----------------------------|---|
| Stored procedures | <ul style="list-style-type: none"> • CREATE PROCEDURE and REPLACE PROCEDURE (External Form) • CREATE PROCEDURE and REPLACE PROCEDURE (SQL Form) |
| User-defined functions | <ul style="list-style-type: none"> • CREATE FUNCTION and REPLACE FUNCTION (External Form) |

| For more information on ... | See ... |
|-----------------------------|--|
| | <ul style="list-style-type: none">• CREATE FUNCTION (Table Form) |
| User-defined methods | CREATE METHOD. |

CREATE TYPE (Structured Form)

Internal And External Representations Of Structured UDTs

Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

The internal and external representations of structured UDTs are indicated in the following table:

| FOR this type of representation ... | A structured UDT is stored in the following format: |
|-------------------------------------|---|
| internal | A concatenation of the data types of its individual attributes. |
| client | however, it is defined by its fromsql transform functionality. See CREATE TRANSFORM and REPLACE TRANSFORM for details. Client software does not see UDT values, but values having a predefined data type representing a UDT on the Teradata platform. How a platform UDT is transformed to a client predefined type is entirely at the discretion of the transform developer. |

See the appropriate Teradata Tools and Utilities documentation for details of how a particular client application deals with structured UDTs.

Similarities With The Syntax For CREATE TYPE (Distinct Form)

The syntax for CREATE TYPE (Structured Form) is similar to that for the CREATE TYPE (Distinct Form) statement.

The major distinguishing characteristics are the following:

- The AS *attribute_name* list with an optional INSTANTIABLE specification.
For CREATE TYPE (Distinct Form), the corresponding clause is the AS *predefined_data_type* clause.
- The mandatory NOT FINAL specification.
For CREATE TYPE (Distinct Form), the mandatory specification is FINAL.
- The ability to create constructor method signatures, which you cannot do for distinct types.

Function of Structured UDTs

The concept of structured UDTs is analogous to the structure construct defined for the C and C++ programming languages. In contrast with distinct UDTs, for which there is a one-to-one correspondence between a UDT and an underlying predefined data type, a structured UDT is a set of named attributes, limited to 512 per UDT definition and to roughly 4,000 total attributes per UDT, that can be treated as an

encapsulated unit rather than as separate fields. See [Workaround for Adding More Than 512 Attributes to a Structured UDT](#).

The attributes of a structured UDT can be any mix of predefined data types, distinct UDTs, or structured UDTs. Encapsulation of structured UDTs provides the ability to make modifications to any or all of the attributes inside the type without modifying the applications that access data having that type.

To manipulate structured UDT data, you are restricted to using the following operations and method types:

- Casting (see [CREATE CAST and REPLACE CAST](#)).
- Ordering (see [CREATE ORDERING and REPLACE ORDERING](#)).
- Transformation (see [CREATE TRANSFORM and REPLACE TRANSFORM](#)).
- Instance methods of the following types:
 - Observer methods
 - Mutator methods

See [System-Generated Observer and Mutator Methods](#).

- Constructor methods (see [CREATE METHOD](#), [System-Generated Constructor Functions](#), and [Constructor Methods](#)).

When you create a structured UDT, the system automatically generates the following instance method types for each attribute of the UDT:

- An observer.
- A mutator.

You can use these two types of instance method to retrieve (observe) information from an attribute or to update (mutate) the information in an attribute, respectively.

You can drop or replace user-defined methods. See [ALTER TYPE](#) and [REPLACE METHOD](#). You cannot drop or replace system-generated methods because they are intrinsic components of the structured UDT attributes they manipulate. You must submit an ALTER TYPE statement with an appropriate DROP METHOD clause to drop a method. Similarly, the ALTER METHOD statement allows you to manipulate only the protection mode of the method or to recompile and relink it.

You can add new attributes to an existing structured UDT or drop existing attributes from it using the ALTER TYPE statement.

For each attribute dropped, the system also removes its associated observer and mutator methods.

For each attribute added, the system automatically generates the appropriate observer and mutator methods to support that attribute.

Vantage does not support inheritance for structured UDTs.

Default Capabilities of Structured UDTs

Unlike the case for distinct UDTs, the system does *not* automatically generate any transform, ordering, or cast functionality for a structured UDT. You must explicitly provide these capabilities yourself by writing an

external C or C++ routine and then creating the appropriate cast, ordering, and transform definitions using the following DDL statements:

- CREATE CAST (see [CREATE CAST and REPLACE CAST](#)).

Also see [Default Cast For a Structured UDT](#).

- CREATE ORDERING (see [CREATE ORDERING and REPLACE ORDERING](#)).

Also see [Default Ordering For a Structured UDT](#).

If you are creating a structured UDT that might be used for a hash index definition, you must ensure that its ordering function never returns a null. If it does, the system aborts the request and returns an error to the requestor.

- CREATE TRANSFORM (see [CREATE TRANSFORM and REPLACE TRANSFORM](#)).

Also see [Default Transform Group For a Structured UDT](#)

You *must* create explicit ordering and transform functionality for all structured UDTs that you create. If you attempt to specify a structured UDT as the data type for any column of any table, but you have not created ordering and transform functionality for that UDT, the system returns an error to the requestor.

You are *not* required to create explicit casting functionality for a structured UDT; however, you must create a cast if the structured UDT is to perform in certain SQL expressions such as the USING modifier, or to participate in various loading operations such as Fastload and MultiLoad.

[Enabling Full Functionality For a Structured UDT](#) summarizes the implications for structured UDTs of this and the topics that follow by explaining which functionalities must be explicitly user-defined before a newly created structured UDT can be used as the type of a column table.

Default Cast For a Structured UDT

Unlike the case for distinct UDTs, the system does *not* create a default cast for a newly created structured UDT. The exceptions to this are the Period and Geospatial data types, which are Teradata internal structured UDTs, and their default casts are predefined.

You can optionally create casts to support any conversions you need to support the structured UDT.

You are *not* required to create explicit casting functionality for a structured UDT; however, you must create a cast if the structured UDT is to perform in certain SQL expressions such as the USING request modifier, or to participate in various loading operations such as Fastload and MultiLoad.

To determine if a structured type has casts defined for it, use the HELP TYPE statement (see [HELP TYPE](#)).

Default Ordering For a Structured UDT

Unlike the case for distinct UDTs, the system does *not* create a default ordering for a newly created structured UDT; however, you must create an ordering for a new structured type before you can use it as a column type for a table or to perform operations on it that involve comparisons, ordering, grouping, sorting, or determination of its uniqueness. The exceptions to this are the Period and Geospatial data types, which are Teradata internal structured UDTs, and their default casts are predefined.

If you attempt to use a new structured UDT without first defining an ordering for it, the request aborts and the system returns an error to the requestor.

To define an ordering for a new structured UDT, use the CREATE ORDERING statement (see [CREATE ORDERING](#) and [REPLACE ORDERING](#)).

To determine if a structured UDT already has a defined ordering, use the HELP TYPE statement (see [HELP TYPE](#)).

Default Transform Group For a Structured UDT

Unlike the case for distinct UDTs, the system does *not* create a default transform group for a newly created structured UDT. However, you must create a transform group for a new structured type before you can use it as a column type for a table or use it in conjunction with operations involving the import or export of data defined with that structured type. If you attempt to use a new structured UDT without first defining a transform group for it, the request aborts and the system returns an error to the requestor.

To define a transform group for a new structured UDT, use the CREATE TRANSFORM statement. See [CREATE TRANSFORM](#) and [REPLACE TRANSFORM](#).

To determine if a structured UDT has a defined transform, use the HELP TYPE statement. See [HELP TYPE](#).

System-Generated Constructor Functions

The system automatically generates a constructor function for each structured UDT you create. This constructor function returns a UDT value with all attributes initialized to an attribute value of NULL. This does *not* mean that a field value having the type is itself null. In fact, you can assign a null attributed structured UDT to a column specified with the NOT NULL column attribute.

Externally, the constructor function is a UDF whose name is the same as the structured UDF. You invoke the constructor method with no parameter, for example: `udt_name()`.

System-Generated Observer and Mutator Methods

When you create a structured UDT, the system automatically generates an observer method and a mutator method for each of its attributes.

To query the value of an attribute, use its observer method, which has the same name as the attribute.

An observer method returns the current value of its associated attribute. Observer methods are invoked without parameters, for example:

```
SELECT column_name.attribute_name()
FROM udt_table;
```


The empty parentheses after the observer method name are required for use with Vantage. Although the current ANSI SQL standard defines empty parentheses as optional, empty parentheses are mandatory to preserve compatibility with existing applications.

To update the value of an attribute or to insert a row that contains a column defined with a structured UDT, use its mutator method, which has the same name as the attribute.

Mutator methods are invoked with one parameter, the new value of the attribute. The mutator method returns a new instance of the structured UDT that is identical to the input UDT except for the specified attribute, whose value is set to the value of the parameter.

The following example shows a the mutator invocation in an UPDATE request:

```
UPDATE udt_table
SET column_name = column_name.attribute_name(new_attribute_value);
```

You can invoke multiple mutator methods together to initialize different attributes in one expression.

The following example illustrates this usage with an INSERT request:

```
INSERT INTO table1
VALUES
(column_1_value,      udt_name().attr_1(attr_1_value).attr_2(attr_2_value));
```

The UDT expression in this INSERT request is executed the following order:

1. The constructor function *udt_name()* is invoked.
The result is a default UDT instance.
2. The mutator method for attribute1 is invoked.
The result is a UDT instance with its *attr_1* set to *attr_1_value*.
3. The mutator method *attr_2* is invoked.
The result is a UDT instance with its *attr_2* set to *attr_2_value*.
4. The final result is a UDT instance with its *attr_1* set to *attr_1_value* and *attr_2* set to *attr_2_value*.

Note that mutators are methods, with the same routine resolution rules as any other method or UDF.

UDFs are limited with respect their predefined data type parameter compatibility. For example, UDFs do not inherently support most of the implicit Teradata predefined data type-to-predefined data type conversions. The best practice for invoking a mutator is to explicitly cast its mutator parameter data type to be the same as the declared attribute type.

Handling Character Sets

If a predefined data type is defined as the character type for an attribute, you can also specify a CHARACTER SET clause to associate a character set with the attribute. If you do not specify a character

set, then the system stores the character set that was in effect at the time the UDT was created in *DBC.TVFields* as the character set for the character attribute.

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 external procedure object even if the new object name contains only single-byte characters. Otherwise, the database aborts the request and returns an error to the requestor. Instead, use a multibyte session character set.

When a character attribute is manipulated by either an observer or mutator method, the character set associated with that attribute type must be one of the following:

- The character set that was declared for that attribute in the CREATE TYPE statement that created the structured UDT.
- The character set that was in effect at the time that the CREATE TYPE statement was executed.

Collation information is not stored with the character type, so the collation that is in effect is always the effective collation sequence at the time the UDT is accessed.

The CHARACTER SET clause influences the auto-generated routines. An explicitly specified CHARACTER SET clause registers the auto-generated:

- Observer method to return a character type value of the specified character set.
- Mutator method to pass in a character parameter type of the specified character set.

The CHARACTER SET clause is the *only* clause supported as an attribute for a predefined data type. The system does *not* support column attributes such as CASESPECIFIC. The default case sensitivity that is in effect at the time that the attribute is accessed (at the time the autogenerated routine is invoked) is the effective case sensitivity.

Enabling Full Functionality For a Structured UDT

When you create a structured UDT, the system automatically generates a constructor function in the SYSUDTLIB database for each structured type. The constructor function returns a structured type instance whose attributes are set to their default values. The default value for any attribute of a structured UDT is null.

Because the system generates this functionality automatically by means of interaction with dictionary tables, and not by submitting a CREATE FUNCTION request, it is not possible to submit a SHOW FUNCTION request to report the create text DDL for the system-generated constructor.

Unlike the case for distinct UDTs, the system does not automatically generate casting, ordering, or transform functionalities for structured UDTs. See [Default Capabilities of Structured UDTs](#), [Default Cast For a Structured UDT](#), [Default Ordering For a Structured UDT](#), and [Default Transform Group For a Structured UDT](#). Instead, you must perform all of the statements listed as Yes under the Mandatory? column in the following table to enable the full functionality of a newly created structured UDT.

| Functionality | Defining DDL Statement | Purpose | Mandatory? |
|--|---|---|------------|
| Definition of a structured UDT. | CREATE TYPE (Structured Form) | Declares the body and attributes of a structured UDT. | Yes |
| Definition of a constructor method. | CREATE METHOD or CREATE FUNCTION | Declares any SQL-invoked constructor methods to be used with the structured UDT. Vantage generates a constructor function for a newly created structured UDT automatically. See System-Generated Constructor Functions . | No |
| Definition of an instance method. | CREATE METHOD and CREATE TYPE (Structured Form) | Declares any SQL-invoked instance methods to be associated with the structured UDT. Vantage generates an observer and a mutator instance method for each attribute of a newly created structured UDT automatically. See System-Generated Observer and Mutator Methods . | No |
| Definition of the ordering for comparing structured UDT values. | CREATE ORDERING | Registers the UDF or method to be used as an ordering routine to compare the structured UDT value with other values. | Yes |
| Declaration of the external ordering routine referenced by CREATE ORDERING. | <ul style="list-style-type: none"> CREATE FUNCTION or CREATE METHOD | Declares the ordering routine referenced in the CREATE ORDERING definition. | Yes |
| Transformation of a UDT between client and server data types in both directions. | CREATE TRANSFORM | Registers the UDF and UDF or method to be used as a tosql() and fromsql() transform routine to pass the structured UDF in both directions between the client and server. | Yes |
| Declaration of the external transformation routine referenced by CREATE TRANSFORM. | <ul style="list-style-type: none"> CREATE FUNCTION or CREATE METHOD | Declares the tosql() and fromsql() external routines referenced in the CREATE TRANSFORM definition. | Yes |
| Registration of an external UDF or method casting routine. | CREATE CAST | Registers a UDF or method to be used as a casting routine to cast: <ul style="list-style-type: none"> From one structured UDT to another structured UDT. From a structured UDT to a predefined data type. From a predefined data type to a structured UDT. | Yes |

| Functionality | Defining DDL Statement | Purpose | Mandatory? |
|--|--|--|------------|
| | | Although it is not mandatory to create casting functionality to be able to use a structured UDT as the data type for a table column, you cannot perform some operations on that column unless a casting has been defined for the type. See Default Cast For a Structured UDT . The minimum requirement to achieve full functionality is the two implicit casts that cast from a UDT to a fromsql transform target predefined data type and from the tosql transform source data type to the UDT. | |
| Declaration of the external casting routine referenced by CREATE CAST. | <ul style="list-style-type: none"> • CREATE FUNCTION or • CREATE METHOD | Declares the external casting routine referenced in the CREATE CAST statement. | No |

If you do not create all the mandatory functionalities listed in this table, the structured UDT you create does not have full functionality. Furthermore, you cannot perform any unsupported operations on values having the data type. Attempts to do so return an error to the requestor.

If your applications of the current structured UDT require additional functionality, you can write instance or constructor methods in either the C or C++ language, and then submit CREATE METHOD requests to declare any SQL-invoked methods to be associated with their structured UDT. For details about writing multiple constructor methods for a structured UDT, see [Constructor Methods](#).

SYSUDTLIB Database

SYSUDTLIB is a system database that contain all UDTs, their methods, cast routines, ordering routines, transform routines, and any UDFs that are used as cast routines, ordering routines, or transform routines. Because SYSUDTLIB is a system database, it is created by a DIP script. SYSUDTLIB is created as a database, not a user, so you cannot log on as user SYSUDTLIB.

When the system resolves SQL requests that involve UDTs, it only searches SYSUDTLIB for UDTs and UDT-related database objects. Because of this, you need not explicitly specify a database name for UDTs. If you do specify a database name, however, it must be SYSUDTLIB.

You should not place non-UDT-related objects in SYSUDTLIB. The database requires space for every UDT you define as well as for every method or UDT-related UDF created within it. SYSUDTLIB space usage is dependent primarily on the space required to store the code files implementing the UDM or UDF, or the space required to contain internally generated source code that is used to acquire an instance of a UDT.

The initial space allocation for SYSUDTLIB is 0. The DBA must modify the definition for SYSUDTLIB to assign enough space that UDTs, UDMs, and UDT-related UDFs can be created within it. If you attempt to

create a UDT-related object in SYSUDTLIB when there is insufficient space to do so, the system returns an error to the requestor. As time passes, you might need to increase the storage associated with SYSUDTLIB to accommodate the growing number of UDTs, UDMs, and UDT-related UDFs.

For details on modifying the definition for SYSUDTLIB, see [MODIFY DATABASE](#).

Constructor Methods

A structured UDT can have more than one constructor method, each of which provides a different initialization option. Constructors have the same method name; however, their parameter lists are different, meaning that their method signatures are different.

The following is an example of a constructor method for the structured type *address*:

```
CREATE TYPE address
...
CONSTRUCTOR METHOD address(VARCHAR(20), CHARACTER(5) )
RETURNS address
SELF AS RESULT
LANGUAGE C
DETERMINISTIC
```

This constructor method takes two character strings as input and uses them to initialize the attributes of the *address* structured type.

You can invoke constructor methods using the NEW syntax (see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for details).

The following bullets briefly summarize how to invoke a constructor method:

- NEW syntax:

```
NEW structuredTypeName( ... )
```

- NEW example:

```
NEW address('10795 Via Del Campo', '92127')
```

Limit On the Number of Methods You Can Associate With a UDT

There is no fixed limit on the number of methods you can associate with a UDT. Generally speaking, the upper limit is approximately 500. The reason the limit cannot be expressed deterministically is that methods can have a variable number of parameters, and the aggregate number of method parameters affects this limit, which is due to Parser memory restrictions.

Assigning Structured UDTs

For purposes of structured UDTs, assignment operations are defined to be insert, update, and parameter passing SQL operations only.

A structured UDT value can only be assigned by default with a source of the same type. To support assigning a different data type to a structured UDT target, you must create a cast that can be invoked implicitly on assignments from the desired type to the structured UDT (see [CREATE CAST and REPLACE CAST](#)).

Effect of Platform on Storage Size of Structured UDTs

The size of a structured UDT value is larger for byte-aligned format systems than it is for byte-packed format systems. For more information, see *Teradata Vantage™ - Database Design*, B035-1094.

Migrating or Copying a UDT to a Different System

The object code (including libraries) supporting a UDT must first be installed on the destination system if the type is migrated or copied from one system to another.

Related Information

| For more information on ... | See ... |
|-----------------------------|---|
| Stored procedures | <ul style="list-style-type: none"> • CREATE PROCEDURE and REPLACE PROCEDURE (External Form) • CREATE PROCEDURE and REPLACE PROCEDURE (SQL Form) |
| User-defined functions | <ul style="list-style-type: none"> • CREATE FUNCTION and REPLACE FUNCTION (External Form) • CREATE FUNCTION (Table Form) |
| User-defined methods | CREATE METHOD . |

CREATE USER

Maximum Number of Users and Databases Per System

The maximum number of users and databases for any one system is 4.2 billion.

See also [CREATE DATABASE](#).

How Users and Databases Differ

It is often said that the only difference between a database user and a database is that a user has a password, while a database does not. That was once true, but no longer is. While it remains true that databases cannot be assigned a password, users can have many more attributes than databases.

User and database definitions share the following set of attributes in common.

- A permanent space specification
- A temporary space specification
- A spool space specification
- One or more account strings
- A BEFORE journal specification
- An AFTER journal specification
- A default journal table specification
- A fallback specification for all tables created within the permanent space for the user

Even in the minimal case, the definition of a database is always a proper subset of the definition of a user, while a user is always a proper superset of a database definition.

A user has all the attributes of a database, and can be defined to have a number of additional attributes such as those contained in the following list. A database cannot have any of the attributes in this list.

- A password
- A startup string
- A default role
- A default time zone
- A default date form
- A default database
- A default character set
- A default collation sequence
- A default user profile assigned using a GRANT request
- A user definition that can be partially external to the DBMS through remote user management software

When you define a user, you are defining a user in the generic sense of the term, but you are also defining a *database* that has the same name as the newly defined user. Note that these seemingly separate entities are actually a *single* database object. An important, but often overlooked, distinction between users and databases is that it is possible to define the database component of a user with a PERM space value of 0 bytes. A user defined in this way is essentially *only* a user, and has virtually *no* overlap with the attributes of databases.

Be aware that such pure users are, by default, terminal nodes in their hierarchy because they have no permanent space to pass on to child users or databases.

If you define a user with no permanent space, you must still define some minimum value of SPOOL space (and perhaps of TEMPORARY space if the user will be working with materialized global temporary tables) for the user. These can be inherited from the owner database or user or can be specified explicitly.

Reporting User Space

All space used, whether permanent, temporary, or spool, is reported in the standard database views under either the account ID specified at logon or the default account ID for the user if none is specified at logon.

PERMANENT, SPOOL, and TEMPORARY Space

You can define the maximum amount of disk space available for various purposes for any Teradata user.

If necessary, the system changes the defined PERMANENT, SPOOL, or TEMPORARY disk space limits to the highest multiple of the number of AMPs on the system less than or equal to the requested space, as outlined by the following table:

| IF you do not define this type of space for the user ... | THEN the system-assigned space is the largest value that meets both of the following criteria ... |
|--|--|
| SPOOL | <ul style="list-style-type: none"> • is a multiple of the number of AMPS on the system. • is not greater than the SPOOL space for the owner. |
| TEMPORARY | <ul style="list-style-type: none"> • is a multiple of the number of AMPs on the system. • is not greater than the TEMPORARY space for the owner. |

You should limit the amount of SPOOL space assigned to users to a level that is appropriate for the workloads undertaken by each user. While some users require more spool space than others, you should minimize the allotment of spool for any user to a size that is large enough to permit result sets to be handled and space management routines such as MiniCylPack to run when disk space becomes low, but small enough to force spool overflow errors when runaway queries resulting from Cartesian products and other common SQL coding errors occur.

See *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - Database Administration*, B035-1093 for more information about user space.

Purpose of the STARTUP String

The user startup string is a limited sequence of SQL requests that you can submit at any time after a session has been established. The length of a startup string is limited to 255 characters.

The startup string is processed when:

- You log on to the database through BTEQ.
- You establish a JDBC connection using the Teradata JDBC Driver and have specified the connection parameter RUNSTARTUP=ON. For more information about using a startup string with

the Teradata JDBC Driver, see the *Teradata JDBC Driver Reference*, available at <https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html>.

- An application that uses CLIV2 invokes the startup string using the CLIV2 RunStartUp function. If a startup string is not defined for the user, the database returns an error to the requestor.

Only BTEQ, the Teradata JDBC Driver, and CLIV2 support startup strings.

The remainder of this section describes how to use a startup string with BTEQ.

The following BTEQ example shows how the system invokes a startup string. Suppose you define the startup string for a user as 'ECHO ' '.SHOW CONTROLS' ';'.17. The .SHOW CONTROLS command returns the current settings of the BTEQ formatting command options.

When the user logs onto the database, the following events occur in the indicated order.

1. BTEQ submits a RunStartUp request to the database.
2. The database executes the specified ECHO request and returns the BTEQ command string .SHOW CONTROLS.
3. BTEQ executes the command returned to it, .SHOW CONTROLS.
4. The database returns the settings of the BTEQ formatting command options for the current session.

Suppose you define the startup string for another user as 'SELECT DATE, TIME;'

When this user logs onto the database, the following events occur in the indicated order.

1. BTEQ submits a RunStartUp request to the database.
2. The database executes the SELECT DATE, TIME request and returns the answer set.
3. BTEQ displays the returned answer set.

USING Request Modifier in STARTUP String

The USING request modifier is not supported in a STARTUP string. If it is specified, no error is indicated when the CREATE USER request is processed, and the USING request modifier text is stored with the rest of the startup string. However, when the string is invoked, a message is returned that a "Data or IndicData parcel" is not found or is not allowed.

Local Journaling

The LOCAL single AFTER image journal is supported analogously to a single BEFORE image journal:

- The access privileges required to create or drop a LOCAL single AFTER image journal are the same as for the analogous operation on a single BEFORE image journal.
- LOCAL single AFTER image journaling is restricted to non-fallback data tables.
- Archive/Recovery rules for a single LOCAL AFTER image journal and a single BEFORE image journal are the same, except that LOCAL single AFTER image journal is used with ROLLFORWARD only, while single BEFORE image journal is used with ROLLBACK only.

Activating Permanent Journaling

If you specify only `DEFAULT JOURNAL TABLE = table_name`, then the system creates a journal table, but does not activate it.

To activate the permanent journal, you must also specify either the `AFTER JOURNAL` journaling option or the `BEFORE JOURNAL` option or both.

This action causes permanent journaling to be activated for all tables created in this database afterward.

To determine which tables in your system are journal tables, use the following query:

```
SELECT DBC.dbase.databasename (FORMAT 'X(15)'),
       DBC.tvm.tvpname (FORMAT 'X(25)')
FROM DBC.tvm, DBC.dbase
WHERE DBC.dbase.databaseid=DBC.tvm.databaseid
AND   DBC.tvm.tablekind='j'
ORDER BY 1,2;
```

To determine which databases and users in your system currently have a default journal table defined for them, use the following query:

```
SELECT d.databasename (TITLE 'Database'), TRIM(dx.databasename)
||'. '||TRIM(t.tvpname) (TITLE 'Journal')
FROM DBC.dbase AS d, DBC.TVM AS t, DBC.dbase AS dx
WHERE d.journalid IS NOT NULL
AND   d.journalid <> '00'xb
AND   d.journalid = t.tvmid
AND   t.databaseid = dx.databaseid
ORDER BY 1;
```

To determine which tables in your system are currently being journaled, use the following query:

```
SELECT TRIM(Tables_DB)||'. '||TableName (TITLE 'Table',
CHARACTER(26)), 'Assigned To' (TITLE ' '), TRIM(journals_db)
||'. '||JournalName (TITLE 'Journals', CHARACTER(26))
FROM DBC.journals
ORDER BY 1,2;
```

You can also determine which tables in your system are currently being journaled using the following query that has a somewhat different syntax:

```
SELECT TRIM(d.databasename)||'. '||TRIM(t.tvpname) (FORMAT
'x(45)', TITLE 'Table'), TRIM(dj.databasename)
||'. '||TRIM(tj.tvpname) (TITLE 'Journal')
```

```

FROM DBC.TVM AS t, DBC.TVM AS tj, DBC.dbase AS d, DBC.dbase AS dj
WHERE t.journalid IS NOT NULL
AND t.journalid <> '00'xb
AND t.journalid = tj.tvmid
AND d.databaseid = t.databaseid
AND dj.databaseid = tj.databaseid
ORDER BY 1;

```

Specifying a Default Time Zone for a User

When you assign a default time zone to a user, each time that user logs onto the database, their session uses the assigned time zone offset.

The valid range for a time zone offset specified as \pm 'quotestring' is from -12:59 to +14.00.

If you set the default time zone offset for a user by specifying a non-GMT time zone string, the database can change the time zone automatically whenever there is a change in Daylight Saving Time. Only those time zone strings that are *not* defined in terms of GMT enable automatic adjustments for Daylight Saving Time. The GMT time zones are designed to be used for locations or time zones that do not follow Daylight Saving Time.

The following table documents the time zone options and their definitions.

| Option | Definition |
|---------------------|---|
| LOCAL | The value for the local time zone displacement defined as the system default for this user. Note that this is a persistent setting. If the system-defined default time zone should change, the value defined for the user remains at the value defined for it when she was created. To change the value, you must submit a MODIFY USER request (see MODIFY USER). |
| NULL | No default time zone displacement is defined for the user. |
| \pm 'quotestring' | An optionally signed text string that sets a non-system-default interval offset for converting the user default time zone displacement. The format is hh:mm. |
| 'time_zone_string' | Sets the value of the default time zone for the user 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 <i>GetTimeZoneDisplacement</i> that interprets the string and determines the appropriate time zone displacement for the session (see <i>Teradata Vantage™ - SQL Date and Time Functions and Expressions</i> , B035-1211 for information about the <i>GetTimeZoneDisplacement</i> UDF). Time zone strings that are expressed in terms of GMT do not enable automatic adjustments for Daylight Savings Time. Regions that do not follow Daylight Savings Time are only represented by GMT values. |

The following table documents the time zone strings supported by Vantage.

| Time Zone Strings | | |
|--|--|--|
| <ul style="list-style-type: none"> • Africa Egypt • Africa Morocco • Africa Namibia • America Alaska • America Aleutian • America Argentina • America Atlantic • America Brazil • America Central • America Chile • America Cuba • America Eastern • America Mountain • America Newfoundland • America Pacific • America Paraguay • America Uruguay • Argentina • Asia Gaza • Asia Iran • Asia Iraq • Asia Irkutsk • Asia Israel • Asia Jordan • Asia Kamchatka • Asia Krasnoyarsk • Asia Lebanon • Asia Magadan | <ul style="list-style-type: none"> • Asia Omsk • Asia Syria • Asia Vladivostok • Asia West Bank • Asia Yakutsk • Asia Yekaterinburg • Australia Central • Australia Eastern • Australia Western • Europe Central • Europe Eastern • Europe Kaliningrad • Europe Moscow • Europe Samara • Europe Western • Indian Mauritius • Mexico Central • Mexico Northwest • Mexico Pacific • Pacific New Zealand • Pacific Samoa • GMT-11 • GMT-10 • GMT-9 • GMT-8 • GMT-7 • GMT-6:30 • GMT-6 | <ul style="list-style-type: none"> • GMT-5 • GMT-4 • GMT-3 • GMT-2 • GMT-1 • GMT • GMT+1 • GMT+2 • GMT+3 • GMT+3:30 • GMT+4 • GMT+4:30 • GMT+5 • GMT+5:30 • GMT+5:45 • GMT+6 • GMT+6:30 • GMT+7 • GMT+8 • GMT+8:45 • GMT+9 • GMT+9:30 • GMT+10 • GMT+11 • GMT+11:30 • GMT+12 • GMT+13 • GMT+14 |

See *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211 for the definition of the GetTimeZoneDisplacement UDF that converts time zone values when Daylight Saving Time changes.

See *Teradata Vantage™ - Database Utilities*, B035-1102 for information about the DBS Control flags SystemTimeZoneHour, SystemTimeZoneMinute, and TimeDateWZControl and their control of the automatic conversion of Daylight Saving Time to standard time and back.

See *Teradata Vantage™ - Database Utilities*, B035-1102 for information about the tdlocaledef utility and how it is used to specify rules for time zone strings.

Related Information

- [Activating Permanent Journaling](#) for information about activating permanent journaling for existing tables.

- [ALTER TABLE \(Basic Table Parameters\)](#)
- [CREATE DATABASE](#)
- [MODIFY DATABASE](#)
- [MODIFY USER](#)
- [SET TIME ZONE](#)
- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for examples of the preceding statements.
- *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125 for information about how to create a custom export width table.

CREATE VIEW and REPLACE VIEW

UDT, Period, and Geospatial Expressions in Views

UDT, Period, and Geospatial expressions can be used in a view in any way that is consistent with the SQL specified in this document and with existing semantics for CREATE/REPLACE VIEW.

CREATE and REPLACE VIEW, CREATE and REPLACE RECURSIVE VIEW

CREATE/REPLACE VIEW and CREATE/REPLACE RECURSIVE VIEW (see [CREATE RECURSIVE VIEW](#) and [REPLACE RECURSIVE VIEW](#)) share most of the same code and are variations of one another. They are documented separately for clarity.

Limit On Request Size

The maximum size of the fully expanded text for a view is 2 MB.

Because Vantage expands source text in view definitions to fully qualify object names and to normalize expressions, it is possible for a valid view definition to be created, but to be unusable in practice because of stack overflows in the Syntaxer at execution time. In this case, the system returns an error.

WITH RECURSIVE Clause Usage

You cannot specify a WITH RECURSIVE clause (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146) in a view definition.

TOP *n* and TOP *m* PERCENT Clause Usage

You can restrict a view to only *n* rows or *m* percent of the rows in an underlying base table by specifying a TOP *n* or TOP *m* PERCENT clause, respectively, in the view definition. This option provides a fast method to obtain a restricted, statistically non-random sample of table rows without scanning the entire table. For example, you might create a view that specifies only the top 10 rows, or only the top 10 percent of the rows, from an *orders* table.

TOP *m* PERCENT and Rounding

If $(m * \text{COUNT}(*)) / 100.0$ does not evaluate to an integer value, then Vantage rounds it up to the next highest integer value; otherwise, the intended result is not achieved. The following table shows how rounding occurs:

| IF $(m * \text{COUNT}(*)) / 100.0$ evaluates to this value ... | THEN the system rounds it up to this value ... |
|--|--|
| 0.1 | 1 |

| IF ($m \times \text{COUNT}(\ast)/100.0$) evaluates to this value ... | THEN the system rounds it up to this value ... |
|--|--|
| 9.2 | 10 |

Vantage only applies this rounding up to the first 15 digits following the decimal point. For example, if $(m \times \text{COUNT}(\ast)/100.0)$ evaluates to 0.0000000000000001, or 1×10^{-16} , then the system does *not* round it up to 1.

WHERE, HAVING, and QUALIFY Clause Usage

The WHERE clause specifies a conditional expression by which rows for the view are to be selected.

The HAVING clause specifies a conditional expression by which the groups defined by a GROUP BY clause are to be selected.

The QUALIFY clause specifies a conditional expression used to filter the results of a previously computed ordered analytical function according to user-specified conditions.

You cannot specify a SAMPLE clause within a subquery predicate within a WHERE, HAVING, or QUALIFY clause.

Joins can only be defined in the HAVING clause of a view when that HAVING clause is part of a subquery.

Do not use SELECT * in a subquery if any underlying table or view might be modified.

Names are resolved and columns are defined when the view is defined; therefore, changes to the table are not incorporated into the view definition if the SELECT * construct is used.

ORDER BY Clause Usage

You can specify an ORDER BY clause in a view definition only if you also specify either the TOP *n* or TOP *m* PERCENT option. Without a TOP *n* clause, the results on which the ORDER BY should be applied is ambiguous, and therefore not meaningful semantically.

An ORDER BY clause *is* semantically meaningful when used with a TOP *n* subselect operation because it affects the actual results of the subselect in the view definition. For TOP *n* subselect operations, the ORDER BY specification also controls the final result of the subselect operation.

If you want to create a view that also includes the WITH TIES option for TOP *n*, then you must also specify an ORDER BY clause; otherwise, the system ignores the WITH TIES specification.

Rules and Restrictions for Creating, Replacing, and Using Views

To avoid unexpected or incorrect results when querying a view, observe the following rules when you create the view definition.

- An operation on a view that references multiple tables/views causes those tables/views to be joined. The efficiency of such an operation depends on how it uses the view (for example, whether it makes use of indexes defined for the tables).

- Any user with the privilege to use a view has the privilege to override a lock specified by a view definition.
- If tables referenced by a view are modified to add or remove columns, or change data types of existing columns, then attempts to access the view can result in error messages or unexpected results. This is true whether the view references table columns by name or by SELECT *.
- You can create views that reference global temporary, global temporary trace, and volatile tables.
- You cannot create a view on a queue table (see [CREATE TABLE \(Queue Table Form\)](#)).
- You cannot refer to a recursive view, a WITH clause, or WITH RECURSIVE clause in a view definition.
- You can use BLOB, CLOB, UDT, Period, and Geospatial columns in views as long as they do not violate the restrictions on large object use with SQL or the semantics of creating or replacing a view.

The restrictions are named in the following bulleted list:

- You cannot create a base table with more than 32 LOB columns, nor can you alter a base table to have more than 32 LOB columns; however, a view definition can include as many LOB columns as its underlying base tables support.
- You cannot specify BLOB, CLOB, UDT, Period, and Geospatial columns in the GROUP BY or HAVING clauses of a view definition.
- BLOB, CLOB, UDT, Period, and Geospatial columns can be components of a view subject to the restrictions provided in this list and the semantics of view definitions.

For more information about large objects and Period data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

For more information about Geospatial data types, see *Teradata Vantage™ - Geospatial Data Types*, B035-1181.

- Changing the DateFormat or using the tdlocaledef utility to change the default Date format does not change the format of date columns in a view, which derive from the underlying tables and views. However, expressions in a CREATE VIEW statement that require date validation against the DateFormat or default Date format can generate an error message if the DateFormat or default Date format changes. To avoid date string validation errors, specify date constants as ANSI date literals and specify formats in a FORMAT phrase.

If you change the *tdlocaledef.txt* file and issue a tpareset command, the new format string settings affect only those tables that are created *after* the reset. Existing table columns continue to use the extant format string in *DBC.TVFields* unless you submit an ALTER TABLE request to change it (see [ALTER TABLE \(Basic Table Parameters\)](#)).

- The database checks the CREATE VIEW or REPLACE VIEW text for syntax errors, but not for semantic errors. Because of this, it is possible to create a view that is syntactically valid, but not valid semantically. No message is returned when this occurs.
- If there are semantic errors in the view definition, then the request aborts and the database returns a message to the requestor when you perform a DML request that accesses tables through the view.
- You can only specify an ORDER BY clause in a view definition if you also specify either the TOP *n* or TOP *m* PERCENT option. For details, see [ORDER BY Clause Usage](#).

You cannot specify an ORDER BY clause in a derived table subquery within a view definition.

Support for Global Temporary, Global Temporary Trace, and Volatile Tables

You can reference global temporary, global temporary trace, and volatile tables in a view.

When you access a view that involves a global temporary, global temporary trace, or volatile table, then all references to that table are mapped to the corresponding global temporary, global temporary trace, or volatile table within the current session.

When an application performs an INSERT request that inserts rows to a global temporary table through a view, and that table is not materialized in the current session, then an instance of the table is created when the INSERT request performs. You cannot insert rows into a global temporary trace table through a view.

If an INSERT request is issued against a volatile table, all references to volatile tables are mapped to the volatile tables with those names in the current session. If a volatile table referenced in the view has been dropped from the session prior to the time the view performs, then the system returns an error to the requestor.

Updatable Views

Not all valid view definitions are updatable. In other words, there are several types of view definitions that do not permit you to perform DELETE, INSERT, or UPDATE operations against them. The general rule for updatability of a view is that there must be a 1:1 correspondence between a row defined in a view and the same row defined in an underlying base table on which that view is defined. If that 1:1 correspondence does not exist, then the view is not updatable.

The following features, when specified as a component of a view definition, automatically make that view non-updatable, or read-only.

- Expressions
- Joins
 - If a view definition specifies more than one table, it is not updatable.
- Any form of derived column
- Any form of aggregation, including the following specific types:
 - Ordered analytical functions
 - Aggregate functions
- Any form of aggregation-related clauses, including the following:
 - GROUP BY clause
 - HAVING clause
 - QUALIFY clause
- Set operators, including all forms of the following:
 - EXCEPT and MINUS
 - INTERSECT

- UNION
- A TOP *n* or TOP *m* PERCENT clause
- The DISTINCT operator
- A WHERE clause that specifies a nested table expression that references the same table as is referenced by the main WHERE clause for the view definition

You also cannot execute any of the following HELP statements against a non-updatable view.

- HELP CONSTRAINT
- HELP INDEX
- HELP STATISTICS

WITH CHECK OPTION Clause in Views

WITH CHECK OPTION pertains to updatable views. Views are typically created to restrict which base table columns and rows a user can access in a table. Base table column projection is specified by the columns named in the *column_name* list of a DELETE, INSERT, MERGE, SELECT, or UPDATE request, while base table row restriction is specified by an optional WHERE clause.

The WITH CHECK OPTION clause is an integrity constraint option that restricts the rows in the table that can be affected by an INSERT or UPDATE request to those defined by the WHERE clause. If you do not specify WITH CHECK OPTION, then the integrity constraints specified by the WHERE clause are not checked for updates. Base table constraints are not affected by this circumvention and continue to be enforced.

Unless you have compelling reasons not to honor the WHERE clause conditions specified by a view definition, you should always specify a WITH CHECK OPTION clause in all your updatable view definitions to protect the integrity of your databases (see *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 for more information about various aspects of maintaining system integrity).

| When WITH CHECK OPTION is ... | Any insert or update made to the table through the view ... |
|-------------------------------|---|
| specified | only inserts or updates rows that satisfy the WHERE clause. |
| not specified | ignores the WHERE clause used in defining the view. |

For example, a query could specify WHERE *column_1* > 10, which restricts the rows that can be seen through the view having a value for *column_1* greater than 10, while the base table could have values less than or equal to 10 as long as no constraint in the table definition prevented values in that range from being inserted.

Joins and Aggregation in View Definitions

When defining a non-updatable view, you can specify one or more aggregate operators (SUM, AVG, MAX, MIN, and COUNT) in the SELECT expression list or in the conditional expression of a HAVING

clause. Creating views with aggregation can eliminate the need to create and delete temporary tables. For many applications, an aggregate join index might be a more optimal solution. For more information about aggregate join indexes, see [Aggregate Join Indexes](#), [Optimizer Rules for Using Aggregate Join Indexes in a Query Plan](#), and *Teradata Vantage™ - Database Design*, B035-1094.

Aggregate operators cannot be used in the conditional expression of a WHERE clause.

Joins and Aggregates on a View

Create a report for a set of times and for each destination that includes an average and a maximum value of the count and sums. The purpose of the report is to determine potential loss of revenue by destination.

Create the view *loss_summary_view*.

```
CREATE VIEW loss_summary_view (week, from_code, to_code, count_a,
                             sum_x, sum_y, sum_z)
AS SELECT c.week, h.from_code, h.to_code, COUNT(h.a),SUM(h.x),
SUM(h.y), SUM(h.z)
   FROM history AS h, calendar AS c
   WHERE c.month = 100610
   AND    c.day = h.day
   GROUP BY 1, 2, 3 ;
```

The SELECT request that uses *loss_summary_view* to create the report looks like this.

```
SELECT lsv.week, ld.to_location, AVG(lsv.count_a),
MAX(lsv.count_a),AVG(lsv.sum_x), MAX(lsv.sum_x),          AVG(lsv.sum_y),
MAX(lsv.sum_y), AVG(lsv.sum_z),
MAX(lsv.sum_z)
   FROM loss_summary_view AS lsv, location_description AS ld
   WHERE lsv.to_code = ld.to_code
   GROUP BY 1, 2;
```

This example joins the *cust_file* table with the aggregate view *cust_prod_sales* to determine which companies purchased more than \$10,000 worth of item 123:

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

The SELECT request that uses *cust_prod_sales* to create the report looks like this.

```
SELECT company_name, sales
   FROM cust_prod_sales AS a, cust_file AS b
```

```
WHERE a.custno = b.custno
AND   a.pcode = 123
AND   a.sales > 10000;
```

SELECT Expression Can Include Data Definition Attributes

When defining a non-updatable view, any expression in the SELECT expression list can include a data definition (Data Type, FORMAT attribute, TITLE attribute, and so on). Data definition attributes determine the form of the view display.

The data definition attributes for a view column can differ from those defined for the associated column of the underlying base tables or views. However, not all data definitions are relevant to view expressions.

For example, range constraints and DEFAULT declarations cannot be used in the expression list of a view definition. Instead, you should use the WHERE clause, or the HAVING clause if GROUP BY is specified, to define range constraints in a view definition.

Views and the LOCKING Request Modifier

When a view is defined with a LOCKING modifier, the specified lock is placed on the underlying base table set each time the view is referenced in an SQL statement.

Overriding LOCKING FOR READ in a View

A LOCKING modifier that specifies locking for ACCESS can be used in a CREATE VIEW statement to give concurrent access to ad hoc users and users who will modify the data.

A READ lock in a view can be overridden by placing a LOCKING ... FOR ACCESS modifier on the view.

For example, assume that an ad hoc user selects data at the same time another user attempts to modify the data. The READ lock placed by the ad hoc user prevents the modifying user from accessing the data because update transactions require a WRITE lock. By creating a view for the ad hoc user that specifies an ACCESS lock, that user cannot prevent a modifying user from completing any desired table modifications.

An ACCESS lock allows data to be retrieved during write activities. Be aware that a view defined with an ACCESS lock might display inconsistent results.

Using Views To Control Access To The Partitions of a PPI Table

You can use views to control access to the partitions of a PPI table. For instance, create a view for SELECT ACCESS to the table and another view for data maintenance. Conditions in the views can restrict access to only the appropriate partitions. The views can be replaced to vary the conditions. Grant privileges only on the views.

If you define a view on a PPI table, that view does not contain the system-derived PARTITION column or the system-derived PARTITION#L *n* columns, where the value of *n* ranges from 1 - 15, inclusive, unless you specify it explicitly in the select list of its definition.

In general, you should not include the system-derived PARTITION or PARTITION#L *n* columns from the underlying tables in view definitions to prevent users from referencing them. A better strategy is to specify conditions on the partitioning columns, because the values of PARTITION (or a given PARTITION#L *n*) might change for rows if the partitioning of an underlying base table is altered. See [ALTER TABLE \(Basic Table Parameters\)](#).

View definitions that include the system-derived PARTITION column or the system-derived PARTITION#L *n* columns *are* appropriate for DBAs.

View Definitions and Character Sets

A view can contain explicit [VAR]CHARACTER(*n*) [CHARACTER SET ...] clauses. If the server character set (for example, CHARACTER SET LATIN) is not specified, the view definition expands the [VAR]CHARACTER(*n*) clause to include that specification.

Expansion is done according to the following rules:

- If the clause is applied to an expression within the body of the view, it takes the server character set of the expression if the expression is of type CHARACTER.
- Otherwise, the server character set is set to the default character set of the creator/modifier of the view.

The default session character set of the creating user of the view is used because otherwise the user default session character set would depend on each individual user, and two users with different default session character sets could get different results from the view.

For example, if the CREATE USER default character set is Kanji1, then the view is expanded as shown below.

Original view definition:

```
CREATE TABLE table_1 (
  cu CHARACTER(5) CHARACTER SET  UNICODE);

CREATE VIEW view_1 AS
  SELECT cu (CHARACTER(6))
  FROM table_1;
```

Expanded view definition:

```
CREATE VIEW v1 AS
  SELECT cu (CHARACTER(6), CHARACTER SET  UNICODE)
  FROM table_1;
```

Similarly, the following view definition is expanded as shown below.

Original view definition:

```
CREATE VIEW v2 AS  
  SELECT 123 (CHARACTER(12));
```

Expanded view definition:

```
CREATE VIEW v2 AS  
  SELECT 123 (CHARACTER(12), CHARACTER SET KANJI1);
```

Related Information

- [CREATE RECURSIVE VIEW and REPLACE RECURSIVE VIEW](#)
- [DROP MACRO](#)

DATABASE - DROP USER

These topics provide supplemental usage information about selected SQL DDL statements alphabetically from DATABASE through DROP USER.

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

DATABASE

Logging Sessions On and Creating a Default Database

When sessions are logged on, an initial default database is set. Normally, this is the defined default database of the user.

You can also use the CREATE PROFILE and MODIFY PROFILE statements to modify the default database. This default database is used until the end of the session, or until a subsequent DATABASE statement is entered. You can include the DATABASE statement in the user startup string.

When Fully Qualified Names Are Required

Using unqualified names is easiest when the default database is the only database referred to by an SQL statement. When an SQL statement references tables, views, or other objects from more than one database, unqualified names can apply to only one of those databases.

When encountering an unqualified table name, the system looks for an object by that name in all of the following databases:

- The default database.
- Any databases referenced by the SQL statement.
- The login user database for a volatile table by that name.

The search must find the table name in only one of those databases. The system returns the “ambiguous table name” error message if the table name exists in more than one of those databases. The default database does not receive priority over any other database.

Therefore, if you reference more than one database, you must use the fully qualified form for names that are contained in more than one of the databases.

The following syntax shows the fully-qualified form referencing, for example, a table:

```
database_name.table_name
```

DATABASE Statement Not Valid In 2PC Mode

In 2PC mode, DATABASE is treated as a DDL statement, so it is not valid.

Resolving Unqualified References

The default database of the executing user, not the creating user, is used to resolve unqualified references to database objects in a macro data definition statement (DDL). Therefore, object references in a macro data definition statement should be fully qualified in the macro body (see [CREATE MACRO and REPLACE MACRO](#)).

DELETE DATABASE

DELETE DATABASE Removes Privileges

The DELETE DATABASE statement removes all privileges to the objects that were dropped. Because the Dump and Restore utility does not automatically recover these privileges, they must be reestablished following the restore operation.

Locks

When a DELETE DATABASE statement is performed, Vantage places an EXCLUSIVE lock on the database from which objects are being deleted.

Referenced Objects

If the database you want to delete contains a table that is the subject table for one or more triggers defined in any other database or user, you must first drop those triggers. Otherwise, the statement aborts and the system returns an error message.

DELETE DATABASE Does Not Delete Journal Tables

The DELETE DATABASE statement does not delete a journal table. You must enter a MODIFY DATABASE request to remove a journal table.

DELETE DATABASE and Global Temporary or Volatile Tables

Rules for temporary and volatile tables are as follows:

| IF a table is this type ... | THEN the following things occur when you delete a database ... |
|-----------------------------|--|
| global temporary | <ul style="list-style-type: none"> If you specify ALL, then all objects, including the materialized global temporary tables within the target database, are dropped. If you do not specify ALL, the request aborts if there are any materialized global temporary tables; else all objects in the target database are dropped. |
| volatile | All objects in the target database are dropped except volatile tables. |

DELETE DATABASE and Queue Tables

If a transaction is delayed while waiting to perform a SELECT AND CONSUME operation on a queue table in the specified database, and you delete that database, then the system aborts the delayed transaction.

DELETE DATABASE and Java External Stored Procedures

When you delete a database, you must also delete all Java external stored procedures, Java UDFs, and JAR files.

When you delete the database, the system internally performs the following actions on Java external stored procedure-related dictionary tables. If there are Jars defined in the current database, the system:

- Updates the corresponding row for the current database in *DBC.Dbbase* with an incremented JarLibRevision number.
- Deletes all rows in *DBC.Jar_Jar_Usage* where the JarDatabaseld matches the ID of the database to be deleted.
- Deletes all rows in *DBC.Routine_Jar_Usage* and *DBC.Jars* where the Databaseld matches the ID of the database to be deleted.

DELETE DATABASE and SQL UDFs

A DELETE DATABASE statement deletes all the database objects contained in the database. If database objects contained within other databases or users reference any of the deleted objects, the referencing objects are no longer valid.

An SQL UDF can reference different database objects in its definition, including UDTs. It is always possible that such objects might be dropped either individually, or collectively when you submit a DELETE DATABASE request. When this happens, it can invalidate the UDF. Conversely, an SQL UDF can itself be referenced by various database objects. If an SQL UDF in such a relationship is dropped, or if its containing database or user is dropped, those database objects then lose their validity.

For example, suppose you have database *df2* that contains a table referenced by an SQL UDF in database *p/s*, and for some reason you decide to delete the contents of *df2*. The next time a request references the affected SQL UDF, the system aborts the request and returns an error to the requestor.

DELETE DATABASE and Error Tables

When you delete a database that contains an error table, the system drops the error table regardless of the containing database for its data table. The system also drops the rows for the error table in the *DBC.ErrorTbls* system table. For more information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

You cannot delete a database that contains a data table with an error table that is contained by another database.

The remedy for this situation is as follows:

- Find all the error tables that are defined on the tables in the database being deleted by using the following SELECT statement:

```
SELECT  BaseTblDbName, BaseTblName, ErrTblDbName, ErrTblName
FROM    DBC.ErrorTblsV
```

```
WHERE    BaseTblDbName <> ErrTblDbName
AND      BaseTblDbName = CurrentDBName;
```

where *CurrentDBName* is the database name of the database being deleted.

- Drop all error tables from the list produced by this query.
- Resubmit the DELETE DATABASE statement.

DELETE DATABASE and Online Archive Logging

You can delete a database that has already initiated online archive logging on the entire database or on a subset of its tables, but not while an archive operation is active.

The system stops online archive logging on the database or the specified subset of its tables when the database is deleted and also deletes all of the associated log subtables at that time.

After DELETE Structure Remains Intact

After all the data tables, views, triggers, stored procedures, user-defined functions, and macros have been deleted from a database, the structure of the emptied database remains intact, as if its CREATE statement had just been performed, and it can be used in a subsequent restore operation. The DELETE DATABASE statements is used when a database has been only partially restored and is not in a usable state.

DELETE DATABASE and Statistics

When you delete a database, Vantage drops all of the objects in the database and all of the statistics that have been collected on those objects.

DELETE USER

DELETE USER Removes Privileges

The DELETE USER statement removes all privileges to the objects that were dropped. Because the Dump and Restore utility does not automatically recover these privileges, they must be reestablished following the restore operation.

Locks

When a DELETE USER statement is processed, the system places an EXCLUSIVE lock on the user from which objects are being dropped.

Referenced Objects

To delete contains a table that is the subject table for one or more triggers defined in any other database or user, you must first drop those triggers; otherwise, the statement aborts and the system returns an error message.

DELETE USER Does Not Delete Journal Tables

The DELETE USER statement does not delete a journal table. You must use a MODIFY USER statement to remove a journal table.

DELETE USER and Global Temporary or Volatile Tables

Rules for temporary and volatile tables are as follows:

| IF a table is this type ... | THEN the following things occur when you delete a user ... |
|-----------------------------|--|
| global temporary | <ul style="list-style-type: none"> If you specify ALL, then all objects, including the materialized global temporary tables within the target user, are dropped. If you do not specify ALL, the request aborts if there are any materialized global temporary tables; else all objects in the target user are dropped. |
| volatile | All objects in the target user are dropped except volatile tables. |

DELETE USER and Queue Tables

If a transaction is delayed while waiting to perform a SELECT AND CONSUME operation on a queue table in the specified user, and you delete that user, then the system aborts the delayed transaction.

DELETE USER and Java External Stored Procedures

When you delete a user, you must also delete all Java external procedures, Java UDFs, and JAR files.

When you delete the user, the system internally performs the following actions on Java external procedure-related dictionary tables:

1. If there are Jars defined in the current user, then the system updates the corresponding row for the current database in *DBC.Dbase* with an incremented JarLibRevision number.
2. If there are Jars defined in the current user, then the system deletes all rows in *DBC.Jar_Jar_Usage* where the JarDatabaseld matches the ID of the user to be deleted.
3. If there are Jars defined in the current user, then the system deletes all rows in *DBC. Routine_Jar_Usage* where the Databaseld matches the ID of the user to be deleted.
4. If there are Jars defined in the current user, then the system deletes all rows in *DBC.Jars* where the Databaseld matches the ID of the user to be deleted.

DELETE USER and SQL UDFs

A DELETE USER request deletes all the database objects contained within the specified user. If database objects contained within other databases or users reference any of the deleted objects, the referencing objects are no longer valid.

An SQL UDF can reference different database objects in its definition, including UDTs. It is always possible that such objects might be dropped either individually, or collectively when you submit a DELETE USER request. When this happens, it can invalidate the UDF. Conversely, an SQL UDF can itself be referenced by various database objects. If an SQL UDF in such a relationship is dropped, or if its containing database or user is dropped, those database objects then lose their validity.

For example, suppose you have database *df2* that contains a table referenced by an SQL UDF in user *p/s*, and for some reason you decide to delete the contents of *df2*. The next time a request references the affected SQL UDF, the system aborts the request and returns an error to the requestor.

After DELETE Structure Remains Intact

After all the data tables, views, triggers, stored procedures, user-defined functions, and macros have been deleted from a user, the structure of the emptied user remains intact, as if its CREATE statement had just been performed, and it can be used in a subsequent restore operation. The DELETE USER statement is used when a user has been only partially restored and is not in a usable state.

DELETE USER and Statistics

When you delete a user, Vantage drops all of the objects in the user and all of the statistics that have been collected on those objects.

DROP FUNCTION

External UDF .so Linkage Information

This topic does not apply to SQL UDFs.

There is only one UDF .so file per application category per database per node. The system does not generate .so files for Java UDFs because the source code is contained in the associated JAR files.

| Application Category Value | Format of the .so File |
|----------------------------|--------------------------------|
| > 0 | libudf_dbid_librevno_AppCat.so |
| = 0 | libudf_dbid_librevno.so |

Whenever you create, replace, or drop a UDF, the UDF .so file for that database has to be relinked with all the other UDF object files and then redistributed to all the nodes.

The following example shows the output of a UDF creation on a Linux system. The .so file, libudf_03ee_n.so, increments each time a UDF is created, replaced, or dropped. This DROP FUNCTION operation was the ninth such UDF-related operation in the current database, so the .so file is named libudf_03ee_9.so.

The Unknown statement and 5607 Warning messages are normal.

```
DROP FUNCTION another;

*** Unknown statement, type=124 complete.
*** Warning: 5607 Check output for possible warnings encountered
    in compiling XSP/UDF.
*** Total elapsed time was 1 second.
```

Check output for possible compilation warnings.

```
-----
/usr/bin/cc -G -g -Xc -I /tpasw/etc -o libudf_03ee_9.so
pre_Find_Text.o pattern.o pre_really_long_function_name.o
long_name.o pre_udf_scalar_substr.o substr.o
pre_char2hexint.o char2hexint.o -ludf -lmw -lm
```

Dropping an Algorithmic Compression Function

You can only drop a function that algorithmically compresses or decompresses data if there are no existing references to that UDF by any column across all databases and users.

Dropping a Function For Which Statistics Are Being Collected

If you drop a UDF, the database no longer uses any statistics that involve it.

However, if the UDF is recreated with the same name and same return type, the database uses the statistics again.

If you change the functionality of the UDF however, you should recollect all of the statistics that had previously been collected on it.

DROP HASH INDEX

Restrictions on Dropping a Hash Index on a Table Concurrent With Dynamic AMP Sample Emulation on That Table

You cannot drop a hash index for a table while that table is subject to dynamic AMP sample emulation. To disable dynamic AMP sampling, contact the Teradata Support Center.

To use dynamic AMP sampling on the table from which you dropped the hash index, use the following general procedure:

1. Drop the hash index from the table on the target system.
2. Extract a fresh dynamic AMP sample from the target system.
3. Apply the fresh sample to the source system.

Considerations For Dropping Hash Indexes to Enable Client Batch Load Utilities

You cannot drop a hash index to enable batch data loading by utilities such as MultiLoad and FastLoad as long as queries are running that access that hash index. Each such query places a READ lock on the hash index while it is running, so it blocks the completion of any DROP HASH INDEX transaction until the READ lock is removed. Furthermore, as long as a DROP HASH INDEX transaction is running, batch data loading jobs against the underlying table of the hash index cannot begin processing because of the EXCLUSIVE lock DROP HASH INDEX places on the base table that defines the index.

DROP INDEX

Reasons to Drop a Secondary Index

There are several reasons you might want to drop a secondary index:

- EXPLAIN reports might indicate that access patterns for a table have changed and the index is no longer selected by the Optimizer.

When this happens, you not only waste storage resources by retaining the index, but you also impact system performance by maintaining it.

- You cannot use MultiLoad to load rows into a table that has a USI, so you might need to drop any USIs defined on a table so you can load new rows into.

After the MultiLoad operation completes, you can then recreate the index.

- The performance of some maintenance operations and large update jobs can be better when no secondary indexes are defined on the affected table because of the necessary parallel maintenance that must be done on secondary index subtables.

When you encounter this issue, it can be more high-performing to drop the index, perform the maintenance or update task, and then recreate the index afterward because in many cases, it is more efficient to recreate an index than it is to update it in parallel with the updates made to the base data table or join index on which it is defined.

UNIQUE and PRIMARY INDEX Constraints

If a UNIQUE or PRIMARY KEY constraint has been created implicitly as a USI, you can remove it using a DROP INDEX statement that explicitly references the column set on which the constraint is defined.

Restrictions on Dropping a Secondary Index on a Table Concurrent With Dynamic AMP Sample Emulation on That Table

You cannot drop a secondary index for a table while that table is subject to dynamic AMP sample emulation. To disable dynamic AMP sampling, contact the Teradata Support Center.

To use dynamic AMP sampling on the table from which you dropped the secondary index, use the following general procedure:

1. Drop the secondary index from the table on the target system.
2. Extract a fresh dynamic AMP sample from the target system.
3. Apply the fresh sample to the source system.

Considerations For Dropping Secondary Indexes to Enable Client Batch Load Utilities

You cannot drop a secondary index to enable batch data loading by utilities such as MultiLoad and FastLoad as long as queries are running that access that secondary index. Each such query places a READ lock on the hash index while it is running, so it blocks the completion of any DROP INDEX transaction until the READ lock is removed. Furthermore, as long as a DROP INDEX transaction is running, batch data loading jobs against the underlying table of the index cannot begin processing because of the EXCLUSIVE lock DROP INDEX places on the base table that defines the index.

DROP JOIN INDEX

Restrictions on Dropping a Join Index on a Table Concurrent With Dynamic AMP Sample Emulation on That Table

You cannot drop a join index for a table while that table is subject to dynamic AMP sample emulation. To disable dynamic AMP sampling, contact the Teradata Support Center.

To use dynamic AMP sampling on the table from which you dropped the join index, use the following general procedure:

1. Drop the join index from the table on the target system.
2. Extract a fresh dynamic AMP sample from the target system.
3. Apply the fresh sample to the source system.

Considerations For Dropping Join Indexes to Enable Client Batch Load Utilities

You cannot drop a join index to enable batch data loading by utilities such as MultiLoad and FastLoad as long as queries are running that access that join index. Each such query places a READ lock on the join index while it is running, so it blocks the completion of any DROP JOIN INDEX transaction until the READ lock is removed. Furthermore, as long as a DROP JOIN INDEX transaction is running, batch data loading jobs against the underlying tables of the join index cannot begin processing because of the EXCLUSIVE lock DROP JOIN INDEX places on the base table set that defines the index.

DROP MACRO

Additional DROP Processes for Java External Procedures

The DROP PROCEDURE statement performs some extra bookkeeping on the dictionary tables referenced by the *SQLJ* database, as follows:

1. The same validation of privileges that are required to drop a non-Java external procedure are performed for Java procedures.
2. The database deletes the row in the *DBC.Routine_Jar_Usage* table that indicated that the Java routine being dropped used the JAR specified in its EXTERNAL NAME clause.

These operations do not change either the Java method or the JAR file associated with the Java external procedure being dropped in any way. The request drops only the Java procedure, and any associations made to it from within the *SQLJ* database and dictionary tables.

Referenced Objects

Dropping a table or view does not delete the views, macros, or stored procedures that reference the dropped table or view. You need to drop referencing views, macros, and stored procedures explicitly.

Views, macros, and stored procedures fail after you drop a referenced table or view. If you replace a dropped table or view, the referencing views, macros, and stored procedures operate and provide the expected results only if the replacements have the same structure.

You cannot drop a table on which a trigger, hash index, or join index is defined. You must first drop the index, then drop the table definition.

DROP ORDERING

Restrictions On Dropping An Ordering

If the specified ordering does not exist, the system aborts the drop request and returns an error to the requestor.

There are no restrictions on dropping an ordering. The system always honors a DROP ORDERING request whether it is referenced by a view, stored procedure, trigger, or macro.

However, for an ordering to be dropped, the UDT on which it is defined must not be used by any *table* in the system. There is no restriction on replacing an ordering definition, however, so if you only need to replace, but not drop, an ordering definition to correct or revise its definition, you should use the REPLACE ORDERING statement instead. See [CREATE ORDERING and REPLACE ORDERING](#).

The drop ordering operation succeeds even if the ordering is referenced by a view, stored procedure, trigger, or macro. However, if you attempt to use a view, stored procedure, trigger or macro that references a dropped ordering operation, the request aborts and returns an error to the requestor.

Dropping System-Generated Orderings For Distinct UDTs

You can drop the system-generated orderings for distinct UDTs.

Effects of Dropping An Ordering On Subsequent Operations

There are two consequences of dropping the ordering for a UDT:

- You can no longer specify that UDT in any comparison operation.
Otherwise, the request aborts and the system returns an error to the requestor.
- You cannot specify that UDT as the data type for any SET table.
Otherwise, the request aborts and the system returns an error to the requestor.

DROP ORDERING And System-Generated Constructor UDFs

Note that dropping an ordering also causes the system-generated UDF constructor function for the UDT to be recompiled invisibly (invisible in that the system does not return any compilation messages unless the compilation fails for some reason, in which case the system returns an appropriate error message to the requestor).

DROP STATISTICS (Optimizer Form)

Lock-Related Restrictions

The system places row-hash-level WRITE locks on *DBC.StatsTbl* while dropping statistics. These locks prevent parsing of new requests against the data table for which the statistics had been collected until the drop operation completes.

Transaction-Related Restrictions

DROP STATISTICS must be entered as a single statement request, as the only statement in a macro, or as the last or only statement in a Teradata session mode explicit transaction enclosed by BEGIN TRANSACTION and END TRANSACTION statements.

Dropping Statistics On UDT, BLOB, CLOB, Period, and Geospatial Columns

You cannot drop statistics on UDT, BLOB, CLOB, Period, or Geospatial columns.

When to Use DROP and COLLECT STATISTICS

If you suspect that your table statistics are not current, you should recollect them. The statistics recollection logic retains the history of the information from the prior histograms, which is used to extrapolate stale statistics to freshen them.

The formerly recommended practice of dropping stale statistics is now deprecated because when you drop statistics, you also drop the appropriate history records.

If the column demographics are drastically changed and the history records are no longer able to determine the column trend for the new data, drop the statistics to eliminate the history records and recollect them.

Executing a COLLECT STATISTICS request writes new table demographics over the existing values, so you really do not need to drop old values prior to collecting new ones if you plan to recollect statistics for an entire table.

Rules and Restrictions for DROP STATISTICS

The following rule applies to the DROP STATISTICS statement.

- If you drop the statistics on an object using explicit column specification, Vantage does not drop the object-level statistics for the object.

To do this, you must use the object-level DROP STATISTICS syntax.

DROP TABLE

DROP Processes for DROP TABLE

When you drop a table, the system frees the disk space used by the dropped table (including any secondary index subtables) and any fallback copy, removes any explicit access privileges on the object, and removes the metadata for the dropped object from the data dictionary.

Note:

The data dictionary does not maintain metadata for volatile tables.

Locks for DROP TABLE

The following locks are placed by DROP TABLE:

- The DROP places an all-AMP EXCLUSIVE lock on the object itself as identified by its TVMId value, for example, TableId.
- Among other dictionary locks, the DROP places an all-AMP WRITE lock on the partitioned DBC.AccessRights table on the single partition corresponding to the table unique Databaseld and TVMId. This partition contains the complete list of privileges on the object to be deleted by the DROP operation.

DROP TABLE and Queue Tables

If a transaction is delayed while waiting to perform a SELECT AND CONSUME operation on the specified queue table, then the system cancels that transaction immediately after it drops the table.

DROP TABLE and Error Tables

You cannot drop an error table using the DROP TABLE statement. You must use DROP ERROR TABLE.

To drop a data table has an error table associated with it, you must first drop its error table.

You must drop the error table in a separate DROP ERROR TABLE request. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The system adds a row to the DBC.ErrorTbls system table only when you create an error table, so you do not need to delete any rows from DBC.ErrorTbls when you drop its associated data table.

The DBC.ErrorTblsV or DBC.ErrorTblsVX views displays *Dropped Database*, *Dropped Table*, or *Dropped User*, respectively, in place of the dropped database, data table, or error table creator name. See *Teradata Vantage™ - Data Dictionary*, B035-1092.

DROP TABLE and Statistics

When you drop a table definition, Vantage also drops all the statistics that have been collected on the table.

If you create a new table with the same name, recollecting statistics on the views and queries revalidates the statistics and makes them usable again.

DROP TRANSFORM

Function of DROP TRANSFORM

DROP TRANSFORM drops the transform group that translates the specified UDT between its client and server representations.

The system does not drop the associated external transform routines when you perform DROP TRANSFORM.

Restrictions On Dropping A Transform

There must be an existing transform group definition to be able to drop that definition. If there is no existing transform group defined for the specified UDT, the request aborts and the system returns an error to the requestor.

The system always honors a DROP TRANSFORM request whether it is referenced by a view, stored procedure, trigger, or macro.

However, for a transform to be dropped, the UDT on which it is defined must not be used by any *table* in the system. There is no restriction on replacing a transform definition, however, so if you only need to replace, but not drop, a transform definition to correct or revise its definition, you should use the REPLACE TRANSFORM statement instead. See [CREATE TRANSFORM and REPLACE TRANSFORM](#).

The drop transform operation succeeds even if the ordering is referenced by a view, stored procedure, trigger, or macro. However, if you attempt to use a view, stored procedure, trigger or macro that references a dropped ordering operation, the request returns an error to the requestor.

DROP TRANSFORM And System-Generated Constructor UDFs

Note that dropping a transform also causes the system-generated UDF constructor function for the UDT to be recompiled without returning any compilation messages, unless the compilation fails. Then, the system returns an appropriate error message to the requestor.

DROP USER

Journal Tables and DROP USER

If a user contains a journal table in its space, then the user cannot be dropped until the journal table is removed from the system. A MODIFY USER statement must be used to drop the journal table.

Join and Hash Indexes and DROP USER

If a user contains a table referenced by a join or hash index contained in a different database or user, then you cannot drop the user that contains the referenced table unless you first drop the referencing hash or join index.

See the following for more information:

- [DROP HASH INDEX](#)
- [DROP JOIN INDEX](#)

Triggers and DROP USER

If a user contains a table referenced by a trigger contained within a different database or user, then you cannot drop the user that contains the referenced table unless you first drop the referencing trigger.

For more information, see [DROP MACRO](#).

Query Bands and DROP USER

You cannot drop a user who is currently logged on as a trusted or permanent user through a proxy connection.

When you drop a user who is a trusted or permanent proxy user, the system deletes the rows for that user in DBC.ConnectRulesTbl. For information about DBC.ConnectRulesTbl, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

DBQL Rules and DROP USER

Before you can drop a user for whom one or more DBQL rules have been created, you must first remove those rules for that user. See the DROP USER “Example 2” topic in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Dropping a Populated User

A user must not contain any database objects or you cannot drop it. In other words, you *cannot* drop a populated user. Complete the following procedure to ensure that a user is empty before you attempt to drop it.

1. Drop any hash and join indexes in another database or user that reference a table in the current user. See [DROP HASH INDEX](#) and [DROP JOIN INDEX](#).
2. Drop any triggers in another database or user that reference a table in the current user. See [DROP MACRO](#).
3. Drop any journal tables in the current user. See [MODIFY USER](#).
4. Perform either of the following procedures:
 - Delete the current user. See [DELETE USER](#).
 - Drop all the objects within the current user.

See the appropriate statement set from the following list for more information:

- [DROP FUNCTION](#)
- [DROP INDEX](#)
- DROP PROFILE and DROP ROLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

5. Drop the current user.

DROP Processes

The drop operation verifies that the user is empty, checks that the user does not own any other databases or users, drops the user, and adds the PERM and TEMPORARY space that the drop makes available to that of the immediate owner database or user.

After a user is dropped, it cannot be recovered by using the Dump and Restore utility unless it is restored.

Locks

When a DROP USER statement is processed, the system places an EXCLUSIVE lock on the user being dropped.

Among the dictionary locks required for dropping the user, WRITE locks for deleting privileges from the partitioned DBC.AccessRights table are placed as follows:

- An all-AMP WRITE lock is placed on the partition range assigned to objects in the dropped user/ database. Although DROP USER can only be executed after all objects in the user have been dropped, a final delete is done to clean up any leftover privileges on the user's dropped objects.
- For DROP USER only, an optional single-AMP WRITE lock is placed on the user's RowHash in all partitions if the user has been granted explicit privileges on objects in other users or databases that need to be deleted.

Drop Processes and Java External Procedures

After a DROP USER request has been processed successfully, and before returning its status to the client application, the UDF symbol cache must be spoiled. This is done as part of the dictionary cache spoiling

process. Because you must successfully submit a DELETE USER request before you can submit the DROP USER request, these actions will have already occurred. See [DELETE USER](#).

In conjunction with the DROP USER operation, the following actions are taken on the appropriate dictionary tables:

1. The system checks *DBC.Jar_Jar_Usage* to determine if any JARs in the user to be deleted are in the SQL-Java path of another JAR.
If so, the request aborts and the system returns an error to the requestor.
2. Delete all rows in *DBC.Jar_Jar_Usage* where the value for the *JarDatabaseId* column matches the ID of the user to be deleted.
3. Delete all rows in *DBC.Routine_Jar_Usage* where the value for the *DatabaseId* column matches the ID of the user to be deleted.
4. Delete all rows in *DBC.Jars* where the value for the *DatabaseId* matches the ID of the user to be deleted.

Referenced Objects

You cannot drop a user in which a hash or join index is defined without first dropping the index.

You cannot drop a user that contains a table on which a hash or join index is defined without first dropping the index.

END LOGGING - SET TIME ZONE

For syntax information and examples of how to use these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

END QUERY LOGGING

Query Logging and SQL Transactions

You can only terminate query logging in Teradata session mode if you make the request outside the boundaries of an explicit (BT/ET) transaction.

You cannot perform END QUERY LOGGING in Teradata session mode within an explicit transaction.

You cannot perform END QUERY LOGGING in ANSI session mode.

Determining Whether an Account Was Specified for BEGIN QUERY LOGGING

If you begin query logging on a specific account or account set, then you must perform one of the following actions to end query logging for that account or account set.

| TO end query logging for ... | You must submit the following as the object of the ON clause in an END QUERY LOGGING request... |
|---|---|
| only a specified account or account set for a specific user | the name of that account or the names of some or all of the accounts in the account set This option deletes all rules for the specified account set for <i>user_name</i> . |
| all rules for all users | ALL RULES This option deletes all rules from <i>DBC.DBQLRuleTbl</i> , sets logging off for all active sessions, and clears all rules from the rules cache. |

You can determine whether an account has been specified explicitly or not by submitting the following query against the *DBQLRulesV* view.

```
SELECT *
FROM DBQLRulesV;
```

This query reports all active rules, including whether accounts are specified for them or not.

Effect of END QUERY LOGGING on the Contents of DBQL Caches

END QUERY LOGGING requests flush all DBQL caches except the Summary cache.

Resetting Use Counts and Timestamps

When you submit an END QUERY LOGGING request for a user or database, the system use counts and timestamps reset.

Removing All Query Logging Rules

If you specify the WITH NONE option for a BEGIN QUERY LOGGING request, then you must submit an END QUERY LOGGING request to remove those rules.

The ALL RULES phrase deletes all the rules in *DBQLRuleTbl*, sets logging off for all active sessions, and clears all rules in all DBQL caches.

Related Information

- *Teradata Vantage™ - Database Administration*, B035-1093
- [BEGIN QUERY LOGGING](#)
- [REPLACE QUERY LOGGING](#)
- In *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144:
 - BEGIN QUERY LOGGING
 - END QUERY LOGGING
 - REPLACE QUERY LOGGING

FLUSH QUERY LOGGING

FLUSH QUERY LOGGING Advantages

The FLUSH QUERY LOGGING statement provides some advantages over BEGIN QUERY LOGGING and END QUERY LOGGING statements. The FLUSH QUERY LOGGING statement enables you to flush the various DBQL and TDWM caches at any time. You can submit a FLUSH QUERY LOGGING statement and specify whichever options you choose, and the result is a synchronous flushing of the specified caches.

FLUSH QUERY LOGGING also provides the advantage of flushing selected caches, which you cannot do by specifying periodic BEGIN QUERY LOGGING and END QUERY LOGGING requests. FLUSH QUERY LOGGING also enables you to flush TDWM caches, which can otherwise only be flushed by the database when its timer tells it to do so.

Flushing Options

The following table describes the flushing options for this statement.

| Option | Cache Set Flushed | Table Set Updated |
|---------|------------------------------------|--|
| ALL | All ALLDBQL and all ALLTDWM caches | <ul style="list-style-type: none"> • DBC.ObjectUsage • DBQLExplainTbl • DBQLObjTbl • DBQLLogTbl • DBQLParamTbl • DBQLSqlTbl • DBQLStepTbl • DBQLSummaryTbl • DBQLXMLTbl • TDWMEventHistory • TDWMEventLog • TDWMExceptionLog |
| ALLDBQL | All DBQL caches | <ul style="list-style-type: none"> • DBQLExplainTbl • DBQLObjTbl • DBQLLogTbl • DBQLParamTbl • DBQLSqlTbl • DBQLStepTbl • DBQLSummaryTbl • DBQLXMLTbl |
| ALLTDWM | All TDWM caches | <ul style="list-style-type: none"> • TDWMEventHistory • TDWMEventLog • TDWMExceptionLog |

| Option | Cache Set Flushed | Table Set Updated |
|---------------|--------------------------------|-------------------|
| DEFAULT | DBQL Default Table cache | DBQLLogTbl |
| EXPLAIN | DBQL Explain Table cache | DBQLExplainTbl |
| LOCK | DBQL Lock Logger cache | DBQLXMLLockTbl |
| OBJECTS | DBQL Objects Table cache | DBQLObjTbl |
| PARAMINFO | DBQL Parameter Table | DBQLParamTbl |
| SQL | DBQL SQL Table cache | DBQLSqlTbl |
| STEPINFO | DBQL Step Table cache | DBQLStepTbl |
| SUMMARY | DBQL Summary Table cache | DBQLSummaryTbl |
| TDWMEVENT | TDWM Event Log Table cache | TDWMEventLog |
| TDWMEXCEPTION | TDWM Exception Log Table cache | TDWMExceptionLog |
| TDWMHISTORY | TDWM Event History Table cache | TDWMHistory |
| USECOUNT | ObjectUsage cache | DBC.ObjectUsage |
| XMLPLAN | XML Table | DBQLXMLTbl |

Advantages of Manually Flushing Query Logs

Using FLUSH QUERY LOGGING requests synchronously flushes all DBQL and TDWM caches across the whole system. All the PEs flush respective caches at the same time. When a FLUSH QUERY LOGGING request returns a success response to the client, the cache contents are written to their respective tables.

Note that submitting FLUSH QUERY LOGGING requests too frequently has the same negative performance impact as setting the DBS Control DBQL flag too low.

FLUSH QUERY LOGGING enables all the DBQL and TDWM caches to be flushed synchronously and does not return control to the client until all the caches have been flushed into the appropriate tables.

Performance Issues

The DBQL and TDWM dictionary tables are updated from the caches the database maintains for each of those tables. Without FLUSH QUERY LOGGING, Vantage flushes the caches either based on a timer that flushes them by default every 10 minutes, or when they become full.

This method of updating these logs can cause various data inconsistencies. For example, the DBQLObjTbl cache might get flushed before the DBQLLogTbl cache, which can make it appear as if the rows in DBQLObjTbl are orphaned. Vantage eventually flushes the corresponding row in the DBQLLogTbl cache, but the cache might not be flushed for another 10 minutes. You can modify the flush rate to smaller values by adjusting the DBS Control flag DBQLFlushRate, but flushing the the logging caches more frequently can negatively impact system performance, so this practice is not generally recommended.

Using FLUSH QUERY LOGGING requests to flush the logging caches manually has the overhead of adding additional task to the AMPs for updating their associated dictionary tables. You should always keep this extra overhead in mind and use manual query log cache flushing responsibly.

Note that frequent use of FLUSH QUERY LOGGING requests on a very busy system is not recommended. On a busy system with both DBQL and or TDWM enabled, the log tables are often updated as their respective caches fill, which can frequently occur before the default 10 minute timer expires. An optimal strategy is to use FLUSH QUERY LOGGING requests to flush those caches that Vantage does not flush frequently and where data inconsistencies occur.

Related Information

See the following for more information about query logging.

- *Teradata Vantage™ - Database Administration*, B035-1093
- [BEGIN QUERY LOGGING](#)
- [END QUERY LOGGING](#)
- [REPLACE QUERY LOGGING](#)
- In *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144:
 - BEGIN QUERY LOGGING
 - END QUERY LOGGING
 - FLUSH QUERY LOGGING
 - REPLACE QUERY LOGGING

MODIFY DATABASE

Specifying Multiple Options

Several options can be included in a single MODIFY DATABASE request. No order is imposed, but the database reports an error if options are duplicated or conflicting options are included.

Two or More Databases and Users Can Share a Journal Table

Two or more databases and users can share a journal table. However, if the sharing databases and users retain as their default the name of a journal table that has been dropped from the system, subsequent CREATE TABLE requests that use that default journal abort, and the system return an error message to the requestor.

A Dropped Journal Table Cannot Be Restored

A dropped journal table cannot be restored. Archives containing data tables that reference the dropped journal are no longer valid and cannot be used.

Changing the Default Journal Table Contained Within the Modified Database

Two modify requests are needed to change a default journal table that resides in the database being modified.

1. The first request is used to DROP the present journal table from the system.
The system aborts the request and returns an error message to the requestor if the journal table is being used by active data tables.
2. After the table is dropped, a second request is required to define a new default journal table.

Local Journaling

LOCAL single AFTER image journals are supported analogously to single BEFORE image journals as follows:

- The DROP DATABASE privilege is required to create or drop LOCAL single AFTER image journals. This is the same privilege required to perform the analogous operation on single BEFORE image journals.
- LOCAL single AFTER image journaling is restricted to non-fallback data tables.

MultiLoad, Teradata Parallel Transporter, and FastLoad are impacted by the use of LOCAL journaling.

Activating Permanent Journaling

If you specify only `DEFAULT JOURNAL TABLE = table_name`, then the system creates a journal table, but does not activate it.

To activate the permanent journal, you must also specify either the `AFTER JOURNAL` journaling option or the `BEFORE JOURNAL` option or both.

This action causes permanent journaling to be activated for all tables created in this database afterward.

To determine which tables in your system are journal tables, use the following query:

```
SELECT DBC.dbase.databasename (FORMAT 'X(15)'),
       DBC.tvm.tvpname (FORMAT 'X(25)')
FROM DBC.tvm, DBC.dbase
WHERE DBC.dbase.databaseid=DBC.tvm.databaseid
AND   DBC.tvm.tablekind='j'
ORDER BY 1,2;
```

To determine which databases and users in your system currently have a default journal table defined for them, use the following query:

```
SELECT d.databasename (TITLE 'Database'), TRIM(dx.databasename)
||'. '||TRIM(t.tvpname) (TITLE 'Journal')
FROM DBC.dbase AS d, DBC.TVM AS t, DBC.dbase AS dx
WHERE d.journalid IS NOT NULL
AND   d.journalid <> '00'xb
AND   d.journalid = t.tvmid
AND   t.databaseid = dx.databaseid
ORDER BY 1;
```

To determine which tables in your system are currently being journaled, use the following query:

```
SELECT TRIM(Tables_DB)||'. '||TableName (TITLE 'Table',
CHARACTER(26)), 'Assigned To' (TITLE ' '), TRIM(journals_db)
||'. '||JournalName (TITLE 'Journals', CHARACTER(26))
FROM DBC.journals
ORDER BY 1,2;
```

You can also determine which tables in your system are currently being journaled using the following query that has a somewhat different syntax:

```
SELECT TRIM(d.databasename)||'. '||TRIM(t.tvpname) (FORMAT
'x(45)', TITLE 'Table'), TRIM(dj.databasename)
||'. '||TRIM(tj.tvpname) (TITLE 'Journal')
```

```

FROM DBC.TVM AS t, DBC.TVM AS tj, DBC.dbase AS d, DBC.dbase AS dj
WHERE t.journalid IS NOT NULL
AND t.journalid <> '00'xb
AND t.journalid = tj.tvmid
AND d.databaseid = t.databaseid
AND dj.databaseid = tj.databaseid
ORDER BY 1;

```

Locks and Concurrency

When you do not specify the SPOOL, TEMPORARY, or PERM options, MODIFY DATABASE places an ACCESS lock on the database being altered to run concurrently with all DML requests.

Modifying the SPOOL, TEMPORARY, and PERM options require an EXCLUSIVE lock.

This statement also places a rowhash-level WRITE lock on the database for DBC.DBase. The SPOOL, TEMPORARY, and PERM options then place a second lock on the immediate parent for DBC.DBase. For PERM, the second lock is a rowhash-level WRITE lock. For SPOOL or TEMPORARY, the second lock is a rowhash READ lock.

The SPOOL, TEMPORARY, and PERM options also place a rowhash-level WRITE lock on DBC.DataBaseSpace. The PERM option then places a second rowhash-level WRITE lock on the immediate parent for DBC.DataBaseSpace.

Related Information

See [Activating Permanent Journaling](#) for information about activating permanent journaling for existing tables.

Also see [ALTER TABLE \(Basic Table Parameters\)](#), [CREATE DATABASE](#), [MODIFY DATABASE](#), and [MODIFY USER](#).

MODIFY USER

Option Checking

You can specify multiple options within a single MODIFY USER request; however, the database aborts the request and returns an error if you specify duplicate or conflicting options.

STARTUP Options

You can specify either of the following options for STARTUP.

| Option | Description |
|--------------------|---|
| NULL | The existing startup string for the user, if any, is to be deleted. |
| <i>quotestring</i> | <p>A replacement for the user startup string, which consists of one or more SQL statements that are performed when the user logs onto Vantage.</p> <p>A startup <i>quotestring</i> can be up to 255 characters, must be terminated by a SEMICOLON character, and enclosed by APOSTROPHE characters.</p> <p>A startup Kanji <i>quotestring</i> can be up to 255 bytes, must be terminated by a SEMICOLON character, and enclosed by APOSTROPHE characters.</p> <p>The USING request modifier is not supported, and if a DDL request is specified, no other request is allowed in the string.</p> |

See [Purpose of the STARTUP String](#) for more information about startup strings.

TIME ZONE Option

The following table describes the TIME ZONE options in detail.

| Option | Description |
|----------------------------|--|
| LOCAL | <p>Changes the value for the local time zone displacement to that defined for the system default.</p> <p>Note that this is a persistent setting. If the system-defined default time zone should change, the value defined for the user remains at the value defined for it when the user was created. To change the value, you must submit a new MODIFY USER request.</p> |
| NULL | No default TIME ZONE displacement is defined for the user. |
| \pm <i>'quotestring'</i> | <p>An optionally signed value that sets a non-system-default interval offset for converting the user TIME ZONE displacement.</p> <p>The format is hh:mm.</p> |
| <i>'time_zone_string'</i> | <p>Changes the value of the default time zone for the user to the displacement specified by <i>'time_zone_string'</i>.</p> <p>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 <i>Teradata Vantage™ - SQL Date and Time Functions and Expressions</i>, B035-1211 for information about this system-</p> |

| Option | Description |
|--------|--|
| | defined UDF). Time zone strings that are expressed in terms of GMT do not enable automatic adjustments for Daylight Saving Time. Regions that do not follow Daylight Saving Time are only represented by GMT values. The valid time zone strings are listed in the following table. |

The following table lists the valid time zone strings for the *'time_zone_string'* option.

| Valid Time Zone Strings | | | |
|--|--|---|---|
| <ul style="list-style-type: none"> • Africa Egypt • Africa Morocco • Africa Namibia • America Alaska • America Aleutian • America Argentina • America Atlantic • America Brazil • America Central • America Chile • America Cuba • America Eastern • America Mountain • America Newfoundland • America Pacific • America Paraguay • America Uruguay • Argentina • Asia Gaza • Asia Iran • Asia Iraq | <ul style="list-style-type: none"> • Asia Irkutsk • Asia Israel • Asia Jordan • Asia Kamchatka • Asia Krasnoyarsk • Asia Lebanon • Asia Magadan • Asia Omsk • Asia Syria • Asia Vladivostok • Asia West Bank • Asia Yakutsk • Asia Yekaterinburg • Australia Central • Australia Eastern • Australia Western • Europe Central • Europe Eastern • Europe Kaliningrad • Europe Moscow • Europe Samara | <ul style="list-style-type: none"> • Europe Western • Indian Mauritius • Mexico Central • Mexico Northwest • Mexico Pacific • Pacific New Zealand • Pacific Samoa • GMT-11 • GMT-10 • GMT-9 • GMT-8 • GMT-7 • GMT-6:30 • GMT-6 • GMT-5 • GMT-4 • GMT-3 • GMT-2 • GMT-1 • GMT • GMT+1 | <ul style="list-style-type: none"> • GMT+2 • GMT+3 • GMT+3:30 • GMT+4 • GMT+4:30 • GMT+5 • GMT+5:30 • GMT+5:45 • GMT+6 • GMT+6:30 • GMT+7 • GMT+8 • GMT+8:45 • GMT+9 • GMT+9:30 • GMT+10 • GMT+11 • GMT+11:30 • GMT+12 • GMT+13 • GMT+14 |

DEFAULT DATABASE

If you modify the current DEFAULT DATABASE, the default database is changed for later user sessions. You can submit a DATABASE request to change the default database for the current session (see [DATABASE](#) for information about the DATABASE statement).

The default database for a user cannot be set to NULL, though you can specify the name of a database that does not exist. Note that if you do this, the system aborts the request and returns an error when the user attempts to create or reference an object within a database that does not exist.

To remove the current DEFAULT DATABASE, you can set it to the user name associated with the MODIFY USER request. This is the default setting if a DEFAULT DATABASE clause has not been specified.

Modifying the Default Time Zone for a User

When you assign a default time zone to a user, each time that user logs onto Vantage, their session uses the assigned time zone offset.

The valid range for a time zone offset specified as \pm 'quotestring' is from -12:59 to +14:00.

If you set the default time zone offset for a user by specifying a non-GMT time zone string, Vantage can change the time zone automatically whenever there is a change in Daylight Saving Time. Only those time zone strings that are *not* defined in terms of GMT enable automatic adjustments for Daylight Saving Time. The GMT time zone strings are designed to be used for regions and time zones that do not follow Daylight Saving Time.

See SET TIME ZONE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for a comprehensive list of the valid time zone strings that you can use to set the time zone offset for a user and *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211 for the definition of the GetTimeZoneDisplacement UDF that converts time zone values when Daylight Saving Time changes.

See *Teradata Vantage™ - Database Utilities*, B035-1102 for information about the DBS Control flags SystemTimeZoneHour, SystemTimeZoneMinute, and TimeDateWZControl and their control of the automatic conversion of Daylight Saving Time to standard time and back.

See *Teradata Vantage™ - Database Utilities*, B035-1102 for information about the tdlocaledef utility and how it is used to specify rules for time zone strings.

Two MODIFY Requests Required to Change Default Journal Table

Two MODIFY requests are required to change a default journal table that resides in the user space being modified.

1. The first request drops the present journal table from the database.
2. After the table is dropped, a second request is required to define a new default journal table.

The system aborts the request and returns an error message to the requestor if the journal table is being used by active data tables.

Activating Permanent Journaling

If you specify only DEFAULT JOURNAL TABLE = table_name, then the system creates a journal table, but does not activate it.

To activate the permanent journal, you must also specify either the AFTER JOURNAL journaling option or the BEFORE JOURNAL option or both.

This action causes permanent journaling to be activated for all tables created in this database afterward.

To determine which tables in your system are journal tables, use the following query:


```

SELECT DBC.dbase.databasename (FORMAT 'X(15)'),
DBC.tvn.tvnname (FORMAT 'X(25)')
FROM DBC.tvn,DBC.dbase
WHERE DBC.dbase.databaseid=DBC.tvn.databaseid
AND   DBC.tvn.tablekind='j'
ORDER BY 1,2;

```

To determine which databases and users in your system currently have a default journal table defined for them, use the following query:

```

SELECT d.databasename (TITLE 'Database'),TRIM(dx.databasename)
||'. '||TRIM(t.tvnname)(TITLE 'Journal')
FROM DBC.dbase AS d,DBC.TVM AS t, DBC.dbase AS dx
WHERE d.journalid IS NOT NULL
AND   d.journalid <> '00'xb
AND   d.journalid = t.tvnid
AND   t.databaseid = dx.databaseid
ORDER BY 1;

```

To determine which tables in your system are currently being journaled, use the following query:

```

SELECT TRIM(Tables_DB)||'. '||TableName (TITLE 'Table',
CHARACTER(26)), 'Assigned To' (TITLE ' '),TRIM(journals_db)
||'. '||JournalName (TITLE 'Journals', CHARACTER(26))
FROM DBC.journals
ORDER BY 1,2;

```

You can also determine which tables in your system are currently being journaled using the following query that has a somewhat different syntax:

```

SELECT TRIM(d.databasename)||'. '||TRIM(t.tvnname) (FORMAT
'x(45)',TITLE 'Table'),TRIM(dj.databasename)
||'. '||TRIM(tj.tvnname) (TITLE 'Journal')
FROM DBC.TVM AS t, DBC.TVM AS tj, DBC.dbase AS d, DBC.dbase AS dj
WHERE t.journalid IS NOT NULL
AND   t.journalid <> '00'xb
AND   t.journalid = tj.tvnid
AND   d.databaseid = t.databaseid
AND   dj.databaseid = tj.databaseid
ORDER BY 1;

```

A Dropped Journal Table Cannot Be Restored

You cannot restore a dropped journal table. Archives containing data tables that reference a dropped journal are no longer valid and cannot be used.

Multiple Databases and Users Can Share a Journal Table

Multiple databases and users can share a journal table. However, if the sharing databases and users retain as their default the name of a journal table that has been dropped from the system, subsequent CREATE TABLE requests that use the corresponding default abort the request and return an error message to the requestor.

Local Journaling

LOCAL single AFTER image journals are supported analogously to single BEFORE image journals as follows:

- The DROP DATABASE privilege is required to create or drop LOCAL single AFTER image journals. This is the same privilege required to perform the analogous operation on single BEFORE image journals.
- LOCAL single AFTER image journaling is restricted to non-fallback data tables.

MultiLoad, Teradata Parallel Transporter, and FastLoad are impacted by the use of LOCAL journaling.

Locks and Concurrency

When you do not specify the SPOOL, TEMPORARY, or PERM options, MODIFY USER places an ACCESS lock on the database being altered to run concurrently with all DML requests.

Specifying the SPOOL, TEMPORARY, and PERM options require an EXCLUSIVE lock.

This statement also places a rowhash WRITE lock on the database for DBC.DBase. The SPOOL, TEMPORARY, and PERM options then place a second lock on the immediate parent for DBC.DBase. For PERM, the second lock is a rowhash-level WRITE lock. For SPOOL or TEMPORARY, the second lock is a rowhash-level READ lock.

The SPOOL, TEMPORARY, and PERM options also place a rowhash-level WRITE lock on DBC.DataBaseSpace. The PERM option then places a second rowhash-level WRITE lock on the immediate parent for DBC.DataBaseSpace.

Related Information

- [Activating Permanent Journaling](#) for information about activating permanent journaling for existing tables.
- [ALTER TABLE \(Basic Table Parameters\)](#)
- [CREATE DATABASE](#)

- [MODIFY DATABASE](#)
- [MODIFY USER](#)
- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for examples of the preceding statements.

RENAME FUNCTION (External Form)

RENAME FUNCTION (External Form) and UDTs

You cannot directly change the name of a UDF that performs the role of a transform, ordering, or cast for a UDT.

You must first drop the corresponding transform, ordering, or cast definitions. Then, rename the UDT. To drop an ordering or a transform, see [DROP ORDERING](#) or [DROP TRANSFORM](#). To drop a cast, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

RENAME FUNCTION (External Form) Changes Only One Function Name Per Request

You cannot change the specific function names and overloaded calling names for a function simultaneously. To do that, you must issue separate RENAME FUNCTION requests.

RENAME FUNCTION (External Form) Does Not Change the C or C++ Function Name

To change either the C/C++ function name or function entry point name for a function, you must use the REPLACE FUNCTION statement. You cannot use RENAME FUNCTION to rename a C/C++ function name.

When you rename a function, it retains its original C/C++ entry point name, even if you used the original function name to define that C/C++ entry point name.

RENAME FUNCTION (External Form) and Algorithmic Compression

A RENAME FUNCTION request for a function is only valid if the function is not used by any column across all databases and users for compressing and decompressing its data. This restriction is in place because the column is already compressed using a certain algorithm having stored context and cannot be decompressed if the algorithm name is changed.

Consider the following example. Assume that the algorithmic compression and decompression functions and the table that uses them are created in the order indicated.

```
CREATE FUNCTION scsu_comp ...;

CREATE FUNCTION scsu_decomp ...;

CREATE TABLE t1 (
  col1 INTEGER,
  col2 CHAR(10) COMPRESS ('abc', 'efg')
```

```
COMPRESS USING scsu_comp  
DECOMPRESS USING scsu_decomp);
```

Column *col2* in table *t1* references the algorithmic compression-related UDFs *scsu_comp* and *scsu_decomp* for compression and decompression respectively.

After having created table *t1*, a RENAME FUNCTION request on either *scsu_comp* or *scsu_decomp* returns an error to the requestor.

Character Set Issues for Renaming External Functions

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 external 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.

Rules for Renaming Overloaded Calling Names

The following rules apply to renaming an overloaded external function calling name.

- Only the call name is changed.
- The new overloaded calling name must be unique when its parameters are taken into consideration.
- The new overloaded name has the same parameters as the old one.

Rule for Renaming Specific External Function Names

A specific external function name cannot be the same as a table, view, macro, trigger, hash index, join index, procedure, or specific function name within the same database.

RENAME FUNCTION (SQL Form)

Rule for RENAME FUNCTION (SQL Form) and the Function Body

RENAME FUNCTION (SQL Form) does not change the UDF body.

RENAME FUNCTION (SQL Form) Changes Only One Function Name Per Request

You cannot change the specific function names and overloaded calling names for an SQL function simultaneously. To do that, you must issue separate RENAME FUNCTION requests.

Rules for Renaming Overloaded Calling Names

The following rules apply to renaming an overloaded function calling name.

- Only the call name is changed.
- The new overloaded calling name for an SQL function must be unique with respect to its parameters.
- The new overloaded SQL function name has the same parameters as the old SQL function name. You cannot change the parameters for an SQL function using a RENAME FUNCTION request.

Rule for Renaming Specific SQL Function Names

A specific SQL function name cannot be the same as a table, view, macro, trigger, hash index, join index, stored procedure, or specific function name within the same database.

RENAME PROCEDURE

Qualifying an Object From Another Database

To rename an SQL procedure that is not contained in the same database or user as your current default database setting, you must qualify both the old and new names with the name of the containing database or user for the object.

Note that you cannot change the immediate owner of the renamed object using this statement.

Character Set Issues for Renaming Methods

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 external 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.

Locks and Concurrency

When an SQL procedure, is renamed, an EXCLUSIVE lock is placed on the procedure being renamed.

RENAME TABLE

Qualifying an Object From Another Database

To rename a table that is not contained in the same database or user as your current default database setting, you must qualify both the old and new names with the name of the containing database or user for the object.

Note that you cannot change the immediate owner of the renamed object using this statement.

Locks and Concurrency

When a table is renamed, an EXCLUSIVE lock is placed on the item being renamed.

Renaming Volatile and Global Temporary Tables

You cannot rename a volatile table.

You can rename a global temporary table only when there are no materialized instances of that table anywhere in the system.

Tables That Have Hash or Join Indexes Cannot Be Renamed

You cannot rename a table on which a hash or join index is defined. You must first drop the index, and then you can rename the table.

Function of RENAME Table Requests

When you rename a table, Vantage only changes the table name. All statistics and privileges belonging to the table remain with it under the new name.

RENAME VIEW

Qualifying an Object From Another Database

To rename a view that is not contained in the same database or user as your current default database setting, you must qualify both the old and new names with the name of the containing database or user for the object.

Note that you cannot change the immediate owner of the renamed object using this statement.

Locks and Concurrency

When a view is renamed, an EXCLUSIVE lock is placed on the view being renamed.

REPLACE METHOD

Prerequisites to Replacing a Method

Before you can create or replace a method, you must first create a UDT that contains a prototype declaration, or method signature, for a method to be associated with it.

Before you can replace a method, you must first create it (see [CREATE METHOD](#)). This action completes the definition of the method body initiated when you created the UDT it is associated with.

Function of REPLACE METHOD

When you replace a method, Vantage resets its execution mode to PROTECTED by default. You must then reset its execution mode to NOT PROTECTED using an ALTER METHOD request (see [ALTER METHOD](#)).

REPLACE METHOD replaces the code of a method. Its intended use is to correct software errors in a method.

Recall that creation or replacement of a method requires that there be declarations within two DDL statements on behalf of the method being created or replaced.

- A method signature declaration is required within the CREATE TYPE request defining the UDT with which the method is associated.
- A CREATE METHOD request is required to complete the body definition for the method being created.

With respect to the execution mode of the created or replaced method, the execution mode of the method is changed back to PROTECTED after Vantage replaces its code.

The syntax for REPLACE METHOD does not follow the typical form for Teradata SQL REPLACE statements. The standard REPLACE statement creates the specified object or definition if it does not already exist, acting exactly like its corresponding CREATE statement. REPLACE METHOD diverges from this standard because the CREATE METHOD statement is not a standalone operation.

The creation of a method requires that the method signature must first be specified within a CREATE TYPE or ALTER TYPE request, followed by its declaration by means of a CREATE METHOD or REPLACE METHOD request.

Although REPLACE METHOD does not follow the typical behavior of REPLACE statements with respect to operating as a standalone operation, it does follow the spirit of other replace operations in that it enables the source code for the method to be replaced, recompiled, and redistributed, thus replacing the behavior of the method.

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 request aborts and 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.

Restrictions On The Replacement Of Methods That Implement Ordering Or Transform Operations On UDTs

The following restrictions apply to replacing methods that implement ordering or transform operations on UDTs.

- A method used to implement either ordering or transform functionality for a UDT can only be replaced if a complete definition already exists for it.

This means that the definitions for both the referenced UDT and the method to be replaced, as defined by the CREATE TYPE and CREATE METHOD statements, must already exist.

See [CREATE TYPE \(Distinct Form\)](#), [CREATE TYPE \(Structured Form\)](#), and [CREATE METHOD](#) for information about these statements.

- The REPLACE METHOD specification must be an exact match with the existing method specification in the data dictionary. This means that its name, parameter list, method entry point name as defined in the EXTERNAL clause, and so on must all match.
- The execution mode for the method being altered must be EXECUTE PROTECTED.

If the function is currently set to EXECUTE NOT PROTECTED mode, then you must perform an ALTER METHOD request (see [ALTER METHOD](#)) to change the mode to EXECUTE PROTECTED before you can perform the REPLACE METHOD request.

Unless all of these conditions have been met when you submit the REPLACE METHOD request, the system aborts it and returns an error to the requestor.

Character Set Issues for Renaming Methods

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 external procedure object even if the new object name contains only single-byte characters. Otherwise, the system aborts the request and returns an error to the requestor. Instead, use a multibyte session character set.

REPLACE QUERY LOGGING

REPLACE QUERY LOGGING is identical to BEGIN QUERY LOGGING except that it enables you to replace an existing query logging rule in DBC.DBQLRuleTbl by submitting a request to replace that rule with a new rule that has the options you specify for it. The following table is a high level summary of what a REPLACE QUERY LOGGING request does.

| IF a rule ... | THEN Vantage ... |
|---------------|------------------|
| exists | replaces it. |

| IF a rule ... | THEN Vantage ... |
|----------------|--|
| does not exist | creates one that satisfies the REPLACE QUERY LOGGING specifications. |

See [BEGIN QUERY LOGGING](#) for more information relevant to this statement.

REPLACE QUERY LOGGING Avoids the Problems of Ending an Existing Query Logging Rule Set

You can use an END QUERY LOGGING request like the following to end all the rules in the DBC.DBQLRuleTbl.

```
END QUERY LOGGING ON ALL RULES;
```

However, between the time all the rules have been ended and a new rule has begun with newly specified options, logging does not take place for the specified user, application, or account.

To avoid this problem, the REPLACE QUERY LOGGING statement simply replaces an existing rule using a request like one of the following.

```
REPLACE QUERY LOGGING WITH options ON user_name ACCOUNT = 'acct_ID'
```

or

```
REPLACE QUERY LOGGING WITH options on APPL_NAME=application_name;
```

Related Information

- [BEGIN QUERY LOGGING](#)
- [END QUERY LOGGING](#)
- [FLUSH QUERY LOGGING](#)
- In *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144:
 - BEGIN QUERY LOGGING
 - END QUERY LOGGING
 - FLUSH QUERY LOGGING
 - REPLACE QUERY LOGGING
- *Teradata Vantage™ - Database Administration*, B035-1093

SET QUERY_BAND

SET QUERY_BAND Syntax Rules

The following rules apply to the syntax of SET QUERY_BAND statements.

- Each name:value pair must be separated from other name:value pairs by a SEMICOLON character. No other separator is valid.
- The entire string of *name:value* pairs for a given SET QUERY_BAND request must be delimited by APOSTROPHE characters.
- The value component of a name:value pair can contain an = (equals) sign, but the name component cannot.
- Pair names cannot be repeated within a query band definition.
- When pad characters are located in the following positions, Vantage considers them to be extraneous and removes them from a query band string.
 - At the beginning or end of the query band string
 - Before and after an EQUALS SIGN (=) character
 - Before and after a SEMICOLON (;) character
- Parameterized requests are supported for the ODBC and JDBC APIs for SET QUERY_BAND ... FOR TRANSACTION *only*.

Query bands support the QUESTION MARK (?) parameter marker for parameterized requests submitted by the ODBC and JDBC APIs.

For example, the following SQL procedure definition is not valid because it attempts to pass a query band as the parameterized value *:qbin*.

```
CREATE PROCEDURE setqbmsr(IN qbin VARCHAR(60))
BEGIN
  SET QUERY_BAND = :qbin FOR TRANSACTION;
  INSERT INTO abc (1,2);
END;
```

Function of Query Bands

Query bands provide you with a way to set your own unique identifiers on Teradata sessions and transactions and to add those identifiers to the current set of session identification fields maintained in *DBC.SessionTbl* (see [HELP SESSION](#)).

A query band is a set of user- or middle tier application-defined name:value pairs that can be set on a session or a transaction to identify the originating source of a query. You can think of a query band as metadata that is wrapped around a request. Once a query band has been defined, the system passes it to the database as a list of name:value pairs in a string literal such as the following examples.

```
'org=Finance;report=EndOfYear;universe=west '
'Job=payroll;Userid=dg120444;Jobsession=1122;'
```

Although Vantage has several request identifiers such as user ID, account string, client ID, client address, application name, and so on, query bands provide the following additional useful functionality:

- A means for multitiered applications such as SQL generators and web applications that use session pooling to identify users and applications that would otherwise be unidentifiable for the management of chargeback, account, troubleshooting, and related tasks.
- Assigning different performance priorities for doing things like directly raising the priority of a high priority job.

For example, some applications need to give requests such as those issued for interactive reports a higher priority than requests issued for reports that generate output files.

- A means for grouping requests into arbitrary jobs for accounting and control purposes.

Query banding also provides a means for managing the application end of accounting and control.

The name and the value for each pair can be defined either by a user or by a middle tier application, permitting query bands to be customized to the unique needs of each application.

Following are some of the uses for query bands:

- You can create DBQL reports using query band name:value pairs to provide additional refinement for accounting and resource allocation purposes. Query bands are logged as a column in the DBQL detail logging table.

You can also use query band name:value pairs for debugging performance problems.

- You can use query band name:value pairs as system variables.

You can set a query band for a session and then retrieve information about it using UDFs or external procedures. See [Other Methods of Retrieving Query Band Information](#), and *Teradata Vantage™ - Application Programming Reference*, B035-1090.

- You can set a query band to assert any of the following for a trusted session.
 - A proxy user
 - A proxy role
 - Both a proxy user *and* a proxy role
- You can associate query band name:value pairs with Teradata dynamic workload management software filter rules. For more information, see *Teradata Vantage™ - Application Programming Reference*, B035-1090.

For directory-based connections and applications that all connect to Vantage with a single logon using a web-based application server, you can distinguish the originating source of requests using the query band to permit resource control using Teradata dynamic workload management software rules.

- You can define query band *name:value* pairs as workload classification attributes in Teradata dynamic workload management software.

This enables all requests from a single logon to be classified into different workloads based on the query band set by the originating application.

It also enables an application to set different priorities for different requests. For example, a GUI application might have some dialogs that require quick responses and others that submit long running reports that run in the background. The application can set a different query band for each type of job, causing the requests to be classified into different workloads and thus running at different priorities. As a result, instead of running the entire application at a high priority, the application can adjust the priorities of its requests to enable more optimal use of system resources.

Note that you cannot circumvent filter rules for Query Bands by declaring them to be bypassed. Note that Teradata dynamic workload management software refers to Query Bands by the object type code *QRYBND*.

Query Bands and Session-Level Performance

The session-level query band request is more expensive than the transaction-level query band request because the FOR SESSION option adds the overhead of updating the *queryband* column in *DBC.SessionTbl*, which enables it to be recovered after a system restart.

The VOLATILE option for SET QUERY_BAND FOR SESSION requests enables you to select performance over recoverability by reducing the cost of the request at the expense of being able to recover it after a system restart occurs.

For example, if you want a query band to be restored after a restart, you should submit a SET QUERY_BAND FOR SESSION request. However, if you prefer performance over the capability to restore an active query band, you should specify the VOLATILE keyword with a SET QUERY_BAND FOR SESSION request using this syntax:

```
SET QUERY_BAND = 'name=value;' FOR SESSION VOLATILE;
```

Note that you can specify both the UPDATE and the VOLATILE option within the same SET QUERY_BAND request. You can also use a combination of both versions of a SET QUERY_BAND FOR SESSION request. An application can set some *name:value* pairs FOR SESSION that are saved in *DBC.SessionTbl*, while specifying those *name:value* pairs that are specific to the request using the SET QUERY_BAND UPDATE and VOLATILE options. On a restart, Vantage will restore the query band saved in *DBC.SessionTbl*, but not the query band name-value pairs set using the VOLATILE option. For example:

```
SET QUERY_BAND = 'clientuser=x11;group=acct;' FOR SESSION;
SEL ...
SET QUERY_BAND = 'job=a1;' UPDATE FOR SESSION VOLATILE;
INS ...
INS ...
SET QUERY_BAND = 'job=b2;' UPDATE FOR SESSION VOLATILE;
INS ...
INS ...
```

If Redrive protection is active for the session and a SESSION VOLATILE query band is set, the query band will be recovered after a restart as part of the Redrive feature. This is true only for database restarts that are hidden from the application. For details about Redrive protection, see:

- *Teradata Vantage™ - Database Administration*, B035-1093
- The RedriveProtection and RedriveDefaultParticipation DBS Control fields in *Teradata Vantage™ - Database Utilities*, B035-1102
- [Reserved Redrive Query Band](#)

Query Bands and Trusted Requests

You can set the TrustOnly flag for a trusted user by specifying the WITH TRUST_ONLY option with a GRANT CONNECT THROUGH request (see *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for the syntax and rules for using GRANT CONNECT THROUGH requests).

When you either commit or roll back a transaction, Vantage removes both its query band and any proxy user who has been set with a query band for that transaction. Because you cannot control this by setting the TrustOnly flag, you should instead set a session query band to set the TrustOnly flag when you require it to be in effect for longer than a single transaction.

Query Bands and Trusted Sessions

The following reserved query bands are used by trusted sessions.

| Name | Description |
|-----------|---|
| ProxyRole | 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. |
| ProxyUser | 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. |

Trusted sessions provide you with the ability to authorize middle tier applications to assert user identities and roles for use in checking the privileges for, and logging queries of, individual users without establishing a logon session for each end user of the application. See *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 for an overview of the security issues presented by trusted sessions.

Trusted sessions identify permanent and application users for privilege checking and query auditing when end users make requests against Vantage through a middle tier application such as a web-based product ordering system. Trusted sessions can be used by any type of middle tier application that authenticates its end users and submits SQL requests to Vantage on their behalf.

A trusted session enables a middle tier application to assume the identity of a different user from the one who is logged on for privilege validation. Such a “different user” is referred to as a *proxy user*.

While it is possible to combine query bands and roles to obtain most of the functionality of trusted sessions, trusted sessions have the following advantages over combining the functionality of simple query bands with roles.

- You can set the proxy user and role using just one request, while you would otherwise need to submit two individual SET QUERY_BAND and SET ROLE requests to achieve the same result.
- ProxyUser is a separate column in the query log, while you would have to extract it from a query band.

Trusted sessions push the knowledge of what role can be set for an end user into the database, which is very advantageous for application development.

Proxy users do not log onto Vantage directly, but instead use an established database session, typically derived from a session connection pool. For a definition of connection pooling, see [Query Bands, Trusted Sessions, and Connection Pooling](#). Once a proxy user has been switched onto an active session, all subsequent requests that user makes operate using the privileges granted to the proxy user through a trusted user and both privilege checking and query logging are done using the name of the proxy user. See GRANT CONNECT THROUGH in *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

The following table describes the options for using trusted sessions.

| IF a proxy user is ... | THEN ... |
|---|--|
| a permanent database user | <p>Privileges, roles, or both can be granted to each of the permanent users.</p> <p>Proxy connect privileges can be granted to each permanent user through a trusted user.</p> <p>The application middleware can set the PROXYUSER name in the query band so the session can be switched to the proxy user.</p> <p>Subsequent requests can then run under the privileges of the proxy user.</p> <p>The permanent user can be used to connect as a proxy user or through a direct log onto Vantage.</p> <p>Vantage assigns the name of the proxy user in the trusted session to the name of the creator of any database objects the permanent user creates.</p> |
| an application user who is not known to Vantage | <p>The security administrator can create a role or set of roles with the privileges needed for the set of application users.</p> <p>The security administrator can grant trusted session privileges for the application users through a trusted user using the specified roles.</p> <p>The application middleware can set the query band so the session can be switched to the proxy user.</p> <p>Subsequent requests can then run under the privileges of the active roles of the proxy user.</p> <p>The application user can be used to connect as a proxy user, but cannot directly log onto Vantage.</p> <p>Vantage assigns the name of the trusted user in the trusted session to the name of the creator of any database objects the application user creates.</p> |

Query Bands, Trusted Sessions, and Row-Level Security

Vantage enforces row-level security for proxy users. The following list outlines this support at a high level.

- A proxy user who is also a permanent user with assigned constraints can perform DML requests on row-level security-protected tables.
- Row-level security constraints can either be assigned to a permanent user who is also a proxy user directly or indirectly by means of a profile that is assigned to the permanent user.
- If a proxy user is an application user, then its object-level security privileges are only defined by the roles associated with a session by a SET QUERY_BAND request.
- Proxy user access to row-level security tables follows the same rules as those that apply to trusted session users who access tables that are not protected with row-level security constraints.
- The row-level security constraints that are initially active for a session depend on the constraints that are either directly assigned to the user or that are assigned by means of a profile defined for the user, or both.

This might be the empty set if the neither the user nor the profile are assigned constraint values.

- Vantage determines the row-level security constraint values that are initially active for a session based on the following factors.
 - The initial connection of a session.
 - Execution of an initial SET QUERY_BAND request in the session.
 - Execution of a request that terminates an active transaction in the session.
 - Execution of a subsequent SET QUERY_BAND request that specifies the UPDATE option in the session.
 - Execution of a subsequent SET QUERY_BAND request that does not specify the UPDATE option in the session.

Each of these determining factors is described in more detail in the following five bullets.

You can change the active constraint values for a session to any of those directly allocated to the connecting user or to those allocated through the user profile with a SET SESSION CONSTRAINT request.

The following factors determine the row-level security constraint values that are initially active for a session.

- The initial connection of a session.

The constraint values are taken from the set that is allocated to the profile for the connecting user and from those directly allocated to the connecting user.

This might be the empty set if the user and profile have no allocated constraint values. You can submit a SET SESSION CONSTRAINT request to change the active constraint values for the session to any of those directly allocated to the connecting user or to those allocated through the profile for the connecting user.

- The first SET QUERY_BAND request that is executed in the session.

Execution of an initial SET QUERY_BAND request can also define the initially active row-level security constraints for the session.

The request causes the constraint values that are currently active for the session to be the empty set. The new constraint values that are assigned to the session depend on whether there is a proxy user

assigned as the query band. If so, the assigned constraints depend on whether the proxy user is a permanent user or an application user.

| IF the SET QUERY_BAND request ... | THEN ... |
|---|---|
| does not set a proxy user | the constraint values for the session are those of the connecting user. You can submit a SET SESSION CONSTRAINT request to change the active constraint values active for the session to any of those directly allocated to the connecting user or to those allocated by means of a profile. |
| sets a proxy user and the user is a permanent user | the constraint values for the session are those allocated to the profile for the proxy user and from those directly allocated to the proxy user, which might be the empty set. You can submit a SET SESSION CONSTRAINT request to change constraint values active for the session to either those directly allocated to the proxy user or to those allocated through the user profile. |
| sets a proxy user and the user is an application user | no constraint values are allocated to the session. You cannot submit a SET SESSION CONSTRAINT request to change the constraint values active for the session. Otherwise, the system aborts the request and returns an error to the requestor. |

- When you execute a request that terminates a transaction, the constraint values for the session depend on whether the session has an assigned proxy user and whether the query band is specified for the session or for only the current transaction.

| IF a proxy user is assigned ... | THEN the constraint values ... |
|---------------------------------|--|
| FOR SESSION | are not changed. The constraints remain those that are assigned to the proxy user. |
| FOR TRANSACTION | revert to those set at the initial connection of the session. The constraints are those that are assigned to the connecting user. |

The result of executing a subsequent SET QUERY_BAND request on the constraint values that are currently active for a session depends on whether you specify the UPDATE option or not.

| IF you submit the SET QUERY_BAND request ... | THEN the constraint values currently active for the session ... |
|--|---|
| with the UPDATE option | depend on whether the session currently has a proxy user assigned. Refer to the next table for the possible outcomes. |
| without the UPDATE option | are set to the empty set. The new constraint values for the session are the same as those defined by the rules in the preceding table. |

- When you submit a SET QUERY_BAND request and also specify the UPDATE option, the constraint values that are currently active for the session depend on whether the session currently has a proxy user assigned or not.

The following table describes the possible outcomes when there is a proxy user and a new proxy user is defined.

| FOR this type of user ... | THERE are ... |
|---------------------------|--|
| application | no constraint values allocated to the session if you specify the UPDATE option. You cannot submit a SET SESSION CONSTRAINT request to change the constraint values that are active for the session. Otherwise, the system aborts the request and returns an error to the requestor. |
| permanent | constraint values allocated to the session if you specify the UPDATE option. These are taken from the constraints that are allocated to the profile for the new proxy user and from those directly allocated to the proxy user. This might be the empty set. You can submit a SET SESSION CONSTRAINT request to change the active constraint values for the session to either those directly allocated to the proxy user or to those allocated through the user profile. |

- When you submit a SET QUERY_BAND request without also specifying the UPDATE option, the constraint values that are currently active for the session are reset to the empty set.

The new constraint values for the session are the same as those defined by the first SET QUERY_BAND request that is executed in the session. These constraint values are itemized for the second bulleted item in this list.

Differences Between Setting a Query Band for a Session and for a Transaction

The following table summarizes the differences between SET QUERY_BAND for a SESSION and for a TRANSACTION with respect to their validity for several SQL entities.

| FOR ... | SESSION is ... | TRANSACTION is ... |
|-------------------------|----------------|---|
| the VOLATILE option | supported | not supported. |
| multistatement requests | not supported. | supported. SET QUERY_BAND ... FOR TRANSACTION must be the first statement specified in the multistatement request. You can specify only one SET QUERY_BAND ... FOR TRANSACTION statement in a multistatement request. |
| macros | supported. | supported without restriction. |

| FOR ... | SESSION is ... | TRANSACTION is ... |
|--|--|---|
| | SET QUERY_BAND ... FOR SESSION must be the only statement in the macro. | |
| SQL procedures | not supported. | supported without restriction. You can also specify SET QUERY_BAND ... FOR TRANSACTION as part of a SQL procedure multistatement request. See the row for multistatement requests above. |
| JDBC | supported, but not for ? parameters. | supported without restriction. |
| ODBC | supported, but not for ? parameters. | supported without restriction. |
| changing a proxy user within a transaction | not supported. | supported without restriction. |

High-Level Process for Trusted Sessions

The following event sequence outlines the general process stages undertaken to use a trusted session.

1. The security administrator creates CONNECT THROUGH privileges for an appropriate *trusted_user:permanent* | *application_user* pair using a GRANT CONNECT THROUGH request (see *Teradata Vantage™ - SQL Data Control Language*, B035-1149).
2. The middle tier application creates a connection pool to Vantage.
3. The application end user authenticates itself to the middle tier application and requests a service to submit a query to Vantage.

The method by which the application end user authenticates itself to the middle tier application is not described here because its authentication is the responsibility of the application, not of Vantage.

4. The middle tier application establishes a connection within the connection pool.
5. The middle tier application sets the active session identity and role for the application end user by submitting an appropriate SET QUERY_BAND request to Vantage.
6. Vantage verifies the application end user has been granted trusted session access through the middle tier application database connection.
7. The middle tier application submits an SQL request to Vantage on behalf of the application end user.
8. Vantage verifies the privileges for the request based on the active roles defined for the application end user.
9. Vantage returns the result set to the middle tier application, which then forwards the result set to the application end user.
10. Vantage records the identity of the application end user in any rows inserted into Access Log and Database Query Log tables as appropriate.

| IF the end user makes its connection as this kind of proxy user ... | THEN its identity is logged using this name as specified for the CONNECT THROUGH privilege used to make the trusted session ... |
|---|---|
| application | application name. |
| permanent | permanent user name. |

See *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for the definitions of application and permanent users.

11. The middle tier application returns the connection it had withdrawn to the connection pool.
12. The following housekeeping activities occur when either the session is terminated or Vantage receives a Cleanup parcel (flavor 80).
 - The proxy user is discarded.
 - Any session query bands are discarded.
 - Any transaction query bands are discarded.

Discretionary Access Control Security Issues With Trusted Sessions

Trusted sessions are designed to work with middle tier applications that submit SQL requests to Vantage on behalf of end users. If a middle tier application permits end users to submit SQL requests directly (this is sometimes referred to as *injected SQL*), then an end user could submit a SET QUERY_BAND request that switches the active session identity and role set to another proxy user who has discretionary access control privileges than that user is intended to have, thus enabling a potential security breach.

This security issue does not apply to trusted sessions when Vantage enforces row-level security for a table. For more information, see [Query Bands, Trusted Sessions, and Row-Level Security](#).

To prevent end users from changing the active session identity, the DBA can require that all SET QUERY_BAND requests submitted by a trusted user that set or remove a proxy user be a *trusted request*.

This accomplished using the following steps.

1. Grant the CONNECT THROUGH WITH TRUST_ONLY privilege to the trusted user. (see *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for the syntax and rules for GRANT CONNECT THROUGH requests).

When this privilege is granted to a trusted user, Vantage requires that any SET QUERY_BAND requests submitted by that trusted user that set or remove a proxy user must be designated as a trusted request or it rejects the request.

2. Using the available APIs, an application can designate an SQL request to be either trusted or not trusted.

When an application submits a SET QUERY_BAND request to set the ProxyUser, it can designate the request as *trusted*.

When an application submits SQL requests that are either created or modified by an end user, it can designate the requests as *not trusted* to prevent the client from injecting a SET QUERY_BAND request to change the proxy user or proxy role.

There is no SQL method for upgrading a request to the trusted designation.

When you either commit or roll back a transaction, Vantage removes both its query band and any proxy user who has been set with a query band for that transaction. This cannot be controlled by the TrustOnly flag. Therefore, you should use session query bands when you want to use trusted sessions.

The following table documents the behavior of SET QUERY_BAND requests associated with the different TrustOnly flag settings.

| IF an application specifies that the TrustOnly flag for a request is set to this value ... | THEN a SET QUERY_BAND request that sets or updates a proxy user can be performed ... |
|--|--|
| N This is the default. | as either of the following types of request. not trusted trusted |
| Y | <i>only</i> as a trusted request. When you have set a proxy connection with the TrustOnly flag set to Y, then you cannot submit a SET QUERY_BAND request that removes the Proxy User unless it is performed as a trusted request. |

The following table documents how a procedure that submits a SET QUERY_BAND request becomes trusted or not trusted.

| A SET QUERY_BAND request in this type of procedure ... | IS ... |
|--|--|
| SQL | trusted if the CALL request that invoked it is trusted. not trusted if the CALL request that invoked it is not trusted. |
| external | trusted if the Trusted flag in the Options parcel is set to Y. not trusted if the Trusted flag in the Options parcel is set to N. |

Query Bands, Trusted Sessions, and Connection Pooling

Connection pooling is a technique for sharing server resources among various requesting client applications. It enables multiple clients 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.

Trusted sessions do not change the handling of session parameters in connection pooling by default. Note the following restrictions for connection pooling with Vantage. Because it is not possible to reset a connection to Vantage, you must not change any of the following session parameters during a trusted session, because any changes to them are then inherited by the next user of the connection.

- Character set
- Collation
- Database
- Dateform
- Default Date Format
- Timezone
- Transaction Isolation Level

For information about how to change session parameters, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Using SET QUERY_BAND Requests to Make Proxy User Connections

To make a proxy user connection, a middle tier application that is connected as a trusted user issues a SET QUERY_BAND request that specifies the proxy user name and an optional proxy role for that user. The reserved query band names PROXYUSER and PROXYROLE are used to specify a trusted session user name and proxy role name in the SET QUERY_BAND request, respectively.

When making proxy user connections, a SET QUERY_BAND request performs the following actions:

1. If the query band specifies PROXYUSER, Vantage validates the current user has privileges to connect as the specified proxy user.
2. If the query band specifies PROXYROLE, Vantage validates the role can be set for the specified proxy user.
3. If the validation passes, Vantage sets the session to the specified proxy user name and proxy role name.

Once the proxy connection is made, Vantage uses the proxy user and the proxy role to determine the privileges for all subsequent requests in the session.

Note the following.

- The trusted session lasts for the life of the query band.
- The session query band remains set for the session and ends only when one of the following occurs.
 - The session ends.
 - You set the query band to NONE.
- The session query band is stored in *DBC.SessionTbl*, and the database recovers it after a system reset.
- The transaction query band is discarded when either of the following occurs.
 - The transactions ends (whether by commit, rollback, or abort)
 - The transaction query band is set to NONE and is not restored after a system reset.

A SET QUERY_BAND request returns an error whenever any of the following violations occur.

- The proxy user does not have CONNECT THROUGH privileges with the trusted user.

- The proxy user has not been granted privileges for the specified proxy role.
- The request attempts to set a PROXYUSER for a transaction when a session trusted session already exists.
- The request attempts to set a PROXYUSER for a session when a transaction proxy connection already exists.
- The request attempts to set a PROXYROLE when it is not in a trusted session.
- The request attempts to set a PROXYROLE to NONE or NULL when roles are defined for the trusted user in the GRANT CONNECT THROUGH privilege.

Query Bands, Trusted Sessions, and Multistatement Requests

The following rules apply to query bands, trusted sessions, and multistatement requests.

- As is true for non-trusted sessions, a transaction query band must be the first statement specified in a multistatement request.
- If PROXYUSER and PROXYROLE are set in a query band, then Vantage enforces the privileges defined for the trusted session to validate the privileges of the individual statements in a multistatement request.
- With the following exceptions, the trusted session privileges are not enforced in a multistatement request.
 - When the transaction query band is specified using a ? parameter in a Java or ODBC program.
 - In macros and procedures. See [Query Bands, Trusted Sessions, and Roles](#).

Query Bands, Trusted Sessions, and Roles

The following rules apply to the enforcement of CONNECT THROUGH privilege-defined roles in a trusted session.

- If a CONNECT THROUGH privilege specifies roles, then the following rules apply.
 - You cannot specify a PROXYROLE if you do not also specify a PROXYUSER.
 - You must use PROXYROLE to set the role in a trusted session because you cannot specify a SET ROLE request in a trusted session.
 - If PROXYROLE is not specified in the privilege definition, then all roles specified for the privilege are active.
 - PROXYROLE can be set to any role in the privilege. If you make this specification, then only that role is active.
 - PROXYROLE cannot be set to NONE or NULL.
- If a CONNECT THROUGH privilege specifies WITHOUT ROLE, then the following rules apply.
 - If PROXYROLE is not specified in the privilege definition, then the active role is the default role for the permanent proxy user.
 - PROXYROLE can be set to any role that has been granted to the permanent proxy user.
 - PROXYROLE can be set to NONE or NULL.

- If a CONNECT THROUGH privilege defines proxy roles, then the privileges for a trusted session that uses that privilege are those granted to.
 - Active proxy roles
 - PUBLIC
- If a CONNECT THROUGH privilege specifies WITHOUT ROLE for a permanent user, then the privileges for a trusted session that uses that privilege are those granted to.
 - The permanent user
 - Active roles
 - PUBLIC

Vantage enforces two exceptions to these rules. In these exceptional cases, Vantage does not enforce the privileges established for the proxy user, but instead enforces the privileges stated in the following table.

| FOR this database object type ... | THE following rules for privilege enforcement apply ... |
|-----------------------------------|---|
| macro | The immediately owning database or user must have all the appropriate privileges for executing the macro. |
| SQL procedure | The following check is made only if the procedure is created using SQL SECURITY INVOKER. Otherwise, the proxy user privileges are not used. Vantage checks the privileges of the immediate owner of the procedure for all statements specified in, and all objects referenced in, the procedure body during its execution. |

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. Note the following explain report, with the locking behavior highlighted in boldface type.

```
EXPLAIN SET QUERY_BAND = 'PROXYUSER=dg12345;' FOR SESSION;
```

```
*** Help information returned. 17 rows.
*** Total elapsed time was 1 second.
```

Explanation

```
-----
1) First, we lock DBC.TempStatistics for write on a RowHash,
```

```

and welock DBC.TempTables for write on a RowHash.
2) Next, we will apply the QUERY_BAND to the session.
3) We get the volatile and temporary table ids and delete the
   corresponding table headers and data.
4) We execute the following steps in parallel.
   1) We do a single-AMP DELETE from DBC.TempTables by way of
      the primary index with no residual conditions.
   2) We do a single-AMP DELETE from DBC.TempStatistics by way
      of the primary index with no residual conditions.
3) We do a single-AMP UPDATE from DBC.SessionTbl by way of the
   primary index with no residual conditions.
5) Finally, we send out an END TRANSACTION step to all AMPs
   involved in processing the request.
-> No rows are returned to the user as the result of statement 1.

```

Vantage does *not* remove the volatile and materialized temporary tables when you set a Proxy User in a SET QUERY_BAND FOR TRANSACTION request.

Query Bands and External Routines

External routines can access the query band information for both sessions and transactions. *External routine* is the generic term to describe external procedures, UDFs, and user-defined methods.

For example, SQL procedures can make SQL requests to set the query band of the transaction using a SET QUERY_BAND request. An SQL procedure can also retrieve the current query band using the QUERY_BAND built-in function.

An external procedure can set the transaction query band by calling an SQL procedure. You can also use an FNC call to retrieve the query band or to get the value of a name within it.

Note that there is no FNC interface to *set* a query band (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details).

Query Bands and Teradata Viewpoint/Database Query Log

Query band information is used by Teradata Viewpoint for workload classification and by the Database Query Log for logging. Vantage logs query bands in DBC.DBQLogTbl, as explained by the following table.

| IF you create ... | THEN the query band text contains ... |
|-------------------------------|--|
| no query band | an empty string. |
| only a transaction query band | the following string: =T> <i>transaction_queryband</i> where the <i>transaction_queryband</i> string element specifies the query band for the current transaction. |

| IF you create ... | THEN the query band text contains ... |
|--|---|
| only a session query band | the following string: <code>=S> session_queryband</code> where the <code>session_queryband</code> string element specifies the query band for the current session. |
| both a transaction query band and a session query band | the following concatenated string: <code>=T> transaction_queryband =S> session_queryband</code> where: <ul style="list-style-type: none"> • <code>transaction_queryband</code> specifies the query band for the current transaction. • <code>session_queryband</code> specifies the query band for the current session. For example, a combined session and transaction level query band might look like the following string: <code>=T> job=x1; =S> org=Finance;report=Fin123;</code> |

The state of a query band must be static once the query has been processed by those features; therefore, you cannot change a query band once its defining request has been dispatched.

When using Teradata Viewpoint, you can associate query band name:value pairs with filter rules and define them as workload classification attributes. In a trusted session, filter and throttle rules and workload classification are based on the trusted user. You can set a query band to create a filter rule or workload classification for a proxy user by doing any of the following.

- Associate a query band name:value pair with a filter rule.
- Set a query band name:value pair as part of the workload classification.
- Specify PROXYUSER as the name in a name:value pair, for example,

```
PROXYUSER=proxy_user_name
```

When Teradata Viewpoint compares rules and classification criteria to the query bands, it uses the first name:value pair found in the query bands in which the name in the name:value pair matches the name in the criteria to determine if it should use the rule or classification.

This enables Teradata Viewpoint to do the following things.

- Restrict access to a database object based on the request query band.
- Classify a request into a specific workload based on its query band.

If a query band specifies both a transaction query band and a session query band, and both have the same name, Teradata Viewpoint always uses the name:value pair in the transaction query band.

Vantage searches the query bands in the following order.

1. Transaction query band
2. Session query band

All comparisons are case insensitive.

You can specify a classification object to match query band name:value pairs. The definition contains the query band name with a values include or values exclude list.

See [Function of Query Bands](#) and *Teradata Vantage™ - Database Administration*, B035-1093 for details.

Query Bands and Load Utilities

The term *load utilities* include utilities or applications that use the Teradata FastLoad, MultiLoad, and FastExport protocols such as the Teradata Parallel Transport Load, Update, and Export operator, the FastLoad, MultiLoad, and FastExport utilities, and the embedded JDBC versions of FastExport and FastLoad.

You can classify these utilities based on session query bands. Because a transaction query band is discarded when a transaction ends, you should set a session query band when you use query banding with the data loading utilities if the query band is to be applied to all statements in the utility script.

Client load utilities can set several optional reserved query band names to control throttling. Vantage receives information about the name of the utility and the size of the data through these reserved query band names.

For example, a client utility can set the UtilityDataSize query band by means of a script. This sizing information is then used by Vantage as an aid in determining the number of AMP sessions to use for the load job.

The expected values for UtilityDataSize are SMALL, MEDIUM, and LARGE. These values provide Vantage with a simple way to influence the selection of the number of sessions needed to process the load job. The intent is to select a reasonable number of sessions for the default. You can use session configuration rules to provide a more precise selection.

Note that the client utilities do *not* set the UtilityDataSize query band automatically. To make use of UtilityDataSize, you must specify it in a script like the one in the following example.

```
.LOGTABLE Logtable002;
.LOGON tdpx/user,pwd;

SET QUERY_BAND='UtilityDataSize=LARGE;' UPDATE FOR SESSION;

.BEGIN IMPORT MLOAD TABLES Employee;
.....
.END MLOAD;
.LOGOFF;
```

See [Reserved Query Band Names and Values](#) for a list of reserved query band names and values for the load utilities and to the appropriate Teradata Tools and Utilities documentation for more information about client utilities.

Open API SQL Interfaces To Support Query Banding

Teradata provides several scalar and table UDFs and external procedures to support query banding.

For more information, see *Teradata Vantage™ - Application Programming Reference*, B035-1090.

Other Methods of Retrieving Query Band Information

Teradata supplies equivalent CLIV2 and SQL interfaces to return the query band for a specified session.

| Interface | Command or Function Name |
|-----------|--------------------------|
| CLIV2 | MONITOR QUERYBAND |
| SQL | MonitorQueryBand |

The MONITOR QUERYBAND PM/API interface is fully documented in *Teradata Vantage™ - Application Programming Reference*, B035-1090.

Teradata also supplies three library functions to enable UDFs, methods, and external procedures to retrieve query band names, query band values, or query band name:value pairs.

| To return this value from the current query band ... | Use this library function ... |
|--|-------------------------------|
| name | FNC_GetQueryBand |
| value | FNC_GetQueryBandValue |
| all name:value pairs | FNC_GetQueryBandPairs |

For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Reserved Query Band Names and Values

Note:

As a general rule, customer applications should not use reserved query band names. They are reserved for use by Teradata-written applications and applications written by third party Teradata partners.

Except for the temperature-based block-level compression query bands, the reserved query band names and values are documented only to make you aware that you should not use them. These reserved query band names and values are intended for use only by Teradata Enterprise Applications and by Teradata partner applications.

These categories of query bands are reserved for use by Teradata Enterprise Applications and by Teradata partner applications:

- Load utility query bands
- Storage management query bands
- Redrive query bands

Load Utility Reserved Query Bands

These reserved query bands are used by various Teradata load utilities. Do *not* use these query band names for your applications. They are only listed to make you aware that you should not use them in your own code. For more information about query bands and load utilities, see [Query Bands and Load Utilities](#).

Note:

As a general rule, customer-written applications should not use these query band names. They are reserved for use by Teradata-written applications and applications written by third party Teradata partners.

Following are the load utility reserved query bands:

| Query Band | Description |
|-----------------|---|
| DSAJobType | Specifies either backup or restore. Applicable to the session query band only. |
| UtilityAWT | Specifies the number of AWTs required for the current job. Applicable to the session query band only. |
| UtilityDataSize | Provides an estimate for the size of the data to be loaded, unloaded, backed up, or restored by a utility component. The following values are valid. <ul style="list-style-type: none"> • SMALL • MEDIUM • LARGE Customer-written applications should not use this query band name to prevent conflicts. Applications using embedded utilities can rely on those utilities to set the query band name appropriately. |
| UtilityName | Provides the name of the calling load utility. It represents the mechanism by which data is loaded, extracted, archived, or restored, either by a standalone utility such as FastLoad or by an embedded utility such as JDBC FastLoad. Customer-written applications should not use this query band name to prevent conflicts. Applications using embedded utilities can rely on those utilities to set the query band name appropriately. |

Reserved Query Band Load Utility Names

The following query band utility name values only work with load utilities and APIs that support the reserve load utility query band names. Do not use these reserved utility name values in your applications.

| UtilityName Value | Utility or API Supported |
|-------------------|--|
| CSPLOAD | DUL csp command For information about the csp command, see <i>Teradata Vantage™ - Database Utilities</i> , B035-1102 and pdehelp. |
| FASTEXP | FastExport |

| UtilityName Value | Utility or API Supported |
|----------------------|--|
| | For information about the CLlV2 version of the FastExport utility, see <i>Teradata® FastExport Reference</i> , B035-2410. |
| FASTLOAD | FastLoad For information about the CLlV2 version of the FastLoad utility, see <i>Teradata® FastLoad Reference</i> , B035-2411. |
| JDBCE | JDBC FastExport For information about using the embedded JDBC version of the FastExport utility, see <i>ODBC Driver for Teradata® User Guide</i> . |
| JDBCL | JDBC FastLoad For information about using the embedded JDBC version of the FastLoad utility, see <i>ODBC Driver for Teradata® User Guide</i> . |
| MULTLOAD | MultiLoad For information about using the CLlV2 version of the MultiLoad utility, see <i>Teradata® MultiLoad Reference</i> , B035-2409. |
| TPTEXP | Teradata Parallel Transporter EXPORT operator (FastExport emulation) For information about the Teradata Parallel Transporter Export operator, see <i>Teradata® Parallel Transporter Reference</i> , B035-2436 and <i>Teradata® Parallel Transporter User Guide</i> , B035-2445. |
| TPTLOAD | Teradata Parallel Transporter LOAD operator (FastLoad emulation) For information about the Teradata Parallel Transporter Load operator, see <i>Teradata® Parallel Transporter Reference</i> , B035-2436 and <i>Teradata® Parallel Transporter User Guide</i> , B035-2445. |
| TPTUPD | Teradata Parallel Transporter UPDATE operator (MultiLoad emulation) For information about the Teradata Parallel Transporter Update operator, see <i>Teradata® Parallel Transporter Reference</i> , B035-2436 and <i>Teradata® Parallel Transporter User Guide</i> , B035-2445. |

Reserved Storage Management Query Bands

Note:

Customer-written applications should not use these query band names. They are reserved for use by Teradata-written applications and applications written by third-party Teradata partners.

The following query band names are reserved for use by Teradata Enterprise Applications and partner applications for storage management resources because they are directives to Vantage.

- BlockCompression. For more information, see [BlockCompression Reserved Storage Management Query Bands](#).
- TVS Temperature. For more information, see [TVSTemperature Reserved Storage Management Query Bands](#).

The following SQL statements also support storage management query bands.

- ALTER TABLE (see [ALTER TABLE \(Basic Table Parameters\)](#))
- CREATE HASH INDEX (see [CREATE HASH INDEX](#))
- CREATE JOIN INDEX (see [CREATE JOIN INDEX](#))
- CREATE TABLE ... AS ... WITH DATA (see [CREATE TABLE \(AS Clause\)](#))

CREATE HASH INDEX, CREATE JOIN INDEX, and CREATE TABLE ... AS ... WITH DATA immediately populate the hash or join index subtable or new table with column values from the tables they reference, similar to a load operation.

- INSERT into an empty table (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146)
- INSERT ... SELECT into an empty table (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146)
- MERGE inserts into an empty table (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146)

Incremental Planning and Execution (IPE) Query Bands

You can set query bands to specify when incremental planning and execution is used to generate a dynamic plan.

DynamicPlan Query Band

You can specify the following settings.

| Value | Description |
|--------|---|
| SYSTEM | The system determines whether: <ul style="list-style-type: none"> • Incremental planning and execution (IPE) is used to generate a dynamic plan. • A static plan is generated and IPE is not used. This is the default. |
| OFF | Generates a static plan. Incremental planning and execution (IPE) is not used and a dynamic plan is not generated. |

SpecificPlan Query Band

You can specify the following settings.

| Value | Description |
|--------|---|
| SYSTEM | <p>For a parameterized query, the parameterized request cache (PRC) settings are used to determine whether a generic or specific plan is generated:</p> <ul style="list-style-type: none"> For a specific plan, the DynamicPlan query band setting is used. For a generic plan, a static plan is generated. Incremental planning and execution (IPE) is not used. <p>For a nonparameterized request:</p> <ul style="list-style-type: none"> On the first submission, the DynamicPlan query band setting is used. Subsequently, a static plan is generated. Incremental planning and execution (IPE) is not used. The PRC settings are used to determine whether to cache the plan or not. <p>This is the default.</p> |
| OFF | <p>For a parameterized request, a generic plan is generated. This is a static plan. IPE is not used.</p> <p>For a nonparameterized request, a static plan is generated. IPE is not used..</p> |
| ALWAYS | <p>For a parameterized or nonparameterized request, a specific plan is generated and the DynamicPlan query band setting is used.</p> |

BlockCompression Reserved Storage Management Query Bands

Note:

As a general rule, customer-written applications should not use these query band names. They are reserved for use by Teradata-written applications and applications written by third party Teradata partners.

The primary and fallback subtables of a table can have independent block-level compression attributes.

You can use the BlockCompression query band with tables whose effective BLOCKCOMPRESSION attribute is MANUAL. In that case, the query band indicates if the data being loaded into an empty primary or fallback table (or both) should be compressed at the data block level.

The specifications apply to permanent or global temporary table data, but do not apply to Spool data or Permanent Journal data.

The valid BlockCompression values apply, as noted, to primary data tables, fallback data tables, and work tables.

BlockCompression indicates if the data being loaded into an empty primary or fallback table (or both) should be compressed at the data block level. See [Adding, Modifying, or Dropping Block-Level Compression](#).

| Value | Description |
|-------|--|
| ALL | <p>Compress all new primary and fallback table data at the data block level.</p> <p>This option is equivalent to specifying YES.</p> |

| Value | Description |
|--|--|
| FALLBACK | Compress new fallback data, but do not compress primary table data. |
| FALLBACKANDCLOBS | Compress fallback data, including fallback and primary LOBs that are eligible for compression. |
| <ul style="list-style-type: none"> • NO • NONE | These options are equivalent and specify not to compress new primary and fallback table data at the data block level. |
| ONLYCLOBS | Compress only new LOB data that is eligible for compression. |
| WITHOUTCLOBS | Compress all data, except for LOB data. |
| YES | Compress all new primary and fallback table data at the data block level. This option is equivalent to specifying ALL. |

You can also control block-level compression using the FERRET utility and the compression fields of the DBS Control record. For more information about these DBS Control parameters, see *Teradata Vantage™ - Database Utilities*, B035-1102.

You can use the BlockCompression query band and the DBS Control flags in the following list, or both to control block-level compression.

- CompressPermPrimaryDBs
- CompressPermFallbackDBs
- CompressPermPrimaryCLOBDBs
- CompressPermFallbackCLOBDBs
- CompressGlobalTempPrimaryDBs
- CompressGlobalTempFallbackDBs
- CompressGlobalTempPrimaryCLOBDBs
- CompressGlobalTempFallbackCLOBDBs

If you use both, the BlockCompression query band setting overrides the DBS Control setting for any affected subtables unless the DBS Control setting is NEVER.

The following table summarizes the effects on various data of the BlockCompression query band settings.

| Value | Primary Table | Fallback Table | Primary Subtable Compressible LOBs | Fallback Subtable Compressible LOBs |
|--|----------------|----------------|------------------------------------|-------------------------------------|
| None | Default | Default | Default | Default |
| ALL | Compressed | Compressed | Compressed | Compressed |
| FALLBACK | Not compressed | Compressed | Not compressed | Compressed |
| <ul style="list-style-type: none"> • NO • NONE | Not compressed | Not compressed | Not compressed | Not compressed |

| Value | Primary Table | Fallback Table | Primary Subtable Compressible LOBs | Fallback Subtable Compressible LOBs |
|--------------|----------------|----------------|------------------------------------|-------------------------------------|
| ONLYCLOBS | Not compressed | Not compressed | Compressed | Compressed |
| WITHOUTCLOBS | Compressed | Compressed | Not compressed | Not compressed |
| YES | Compressed | Compressed | Compressed | Compressed |

TVSTemperature Reserved Storage Management Query Bands

In general, customer-written applications should not use reserved query band names, which are reserved for use by Teradata-written applications and applications written by third party Teradata partners. However, the TVSTemperature query bands are an exception to this rule because they support both Teradata Virtual Storage and temperature-based block-level compression.

Use the TVSTemperature query band for tables whose effective BLOCKCOMPRESSION option is AUTOTEMP. For AUTOTEMP to be active, both BLC and TempBLC must be enabled in DBS Control. In that case the TVSTemperature query band specifies the desired temperature for the table cylinders on which the table data is to be stored. This temperature determines the compressed or noncompressed state of blocks in the table. The DBS Control parameter TempBLCThresh determines which temperatures compress data and which decompress data.

Following are the temperature bands for block-level compression. The percentages are approximate.

| Temperature | Description |
|-------------|---|
| COLD | The 20% of data that is least frequently accessed. |
| WARM | The remaining 60% of data that falls between the COLD and HOT bands. |
| HOT | The 20% of data that is most frequently accessed. |
| VERY HOT | Data that you or Teradata Virtual Storage determines should be added to the Very Hot cache list and have its temperature set to very hot when it is loaded using the TVSTemperature query band. |

The following query band names are used for TVSTemperature.

- TVSTEMPERATURE_PRIMARY
- TVSTEMPERATURE_PRIMARYCLOBS
- TVSTEMPERATURE_FALLBACK
- TVSTEMPERATURE_FALLBACKCLOBS

You can specify either VERY HOT, HOT, WARM, or COLD for each of these query band names.

These query bands are valid for both sessions and transactions.

See *Teradata Vantage™ - Teradata® Virtual Storage*, B035-1179 for more information on these query band names.

Note:

Table data being added to an already allocated cylinder retains the temperature initially set for the cylinder, not the temperature you set for the query band.

Reserved Redrive Query Band

Note:

As a general rule, customer-written applications should not use the REDRIVE query band name. It is reserved for use by Teradata-written applications and applications written by third party Teradata partners.

The syntax for setting a Redrive query band is:

```
SET QUERY_BAND = 'redrive=off;' FOR SESSION;  
SET QUERY_BAND = 'redrive=on;' FOR SESSION;
```

The Redrive capability is assigned to a session at logon time. Once a session is enabled with Redrive, you can use the Redrive query band to toggle Redrive protection on and off during the life of the session.

Redrive set to OFF disables Redrive protection for subsequent SQL requests in the session. When Redrive is off, there is no automatic database recovery of submitted SQL requests after a database failure, and any recovery is controlled by the application.

Redrive set to ON enables Redrive protection for subsequent SQL requests in the session.

Note:

When subsequent query bands are processed, the previous query band is replaced completely unless the UPDATE option is specified in the SET QUERY_BAND statement. A session that wants to maintain a specific Redrive behavior must ensure that subsequent query bands include the desired Redrive attribute by specifying the Redrive *name:value* pair or using the UPDATE option to add to the existing query band. Redrive participation will return to the value specified in the RedriveDefaultParticipation DBS Control field if the Redrive attribute is omitted in subsequent query bands.

For details about Redrive functionality, see:

- *Teradata Vantage™ - Database Administration*, B035-1093
- The RedriveProtection and RedriveDefaultParticipation DBS Control fields in *Teradata Vantage™ - Database Utilities*, B035-1102

Related Information

For detailed information related to query bands and trusted sessions, see *Using Query Banding in Teradata Vantage Orange Book*, 541-0007069.

Also see the following for information about query banding and trusted sessions.

- SET QUERY_BAND in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

SET SESSION CALENDAR

Rules and Restrictions for SET SESSION CALENDAR

The following rules and restrictions apply to SET SESSION CALENDAR requests.

- You can use SET SESSION CALENDAR to specify one of the following system-defined business calendars:
 - Teradata
 - ISO
 - COMPATIBLE

You cannot specify a user-defined business calendar.

- The default calendar for all sessions is Teradata.

A new logon always sets the session to the system-defined business calendar Teradata by default, but you can change the default using a SET SESSION CALENDAR request.

You can also include a SET SESSION CALENDAR request in the startup string for a user to set the calendar to something other than the default Teradata calendar. Setting the session calendar in the STARTUP string may result in inconsistent behavior during connection pooling.

The following CREATE USER statement uses the STARTUP string option to set the default session calendar for user *abc* to ISO.

```
CREATE USER abc AS
  PERM=10E6,
  PASSWORD=abc,
  SPOOL = 1200000,
  FALLBACK PROTECTION,
  STARTUP='SET SESSION CALENDAR=ISO';
```

- The session-defined business calendar applies to the anchor name and system-defined calendar UDFs specified in the EXPAND ON clause of any DML requests executed in the session.

For example, if an expansion is by WEEK_BEGIN, the expansion is done by MONDAY for the ISO calendar and by SUNDAY for the Teradata calendar.

- If you do not specify a business calendar as the second argument for a calendar UDF, Vantage applies the calendar specified for the session.
- The system-defined business calendar set for the session survives database restarts.

The calendar name for the session is stored in *DBC.SessionTbl*, so if a restart occurs, the session can be reestablished with the same business calendar that was set before the restart occurred.

- You can change the system-defined business calendar at any time in the session.
- You can change the system-defined business calendar for a session any number of times during that session.

- The *Sys_Calendar.Calendar* and *Sys_Calendar.BusinessCalendar* view columns return values with respect to the calendar that is set for the session.

System-Defined Calendars

Vantage maintains several system-defined calendars in user DBC that you can use for a session. The supported calendars are based on the Gregorian calendar and have 365 days for normal years or 366 days for leap years.

The system-defined calendars have the following names and definitions.

- **Teradata**
All business calendar UDFs consider the CalendarPeriod for the Teradata calendar to be 1900-01-01 to 2100-12-31 and return an error message for any dates that are outside of this period.
The EXPAND ON clause returns results for any dates whether or not they are outside of the system calendar period.
This is the default calendar for all sessions.
- **ISO**
This calendar has the behavior of the ISO Calendar/European Calendar, with Monday as the WeekStart for any week.
All business calendar UDFs consider the CalendarPeriod to be 1900-01-01 to 2100-12-31, but not 0001-01-01 to 9999-12-31, and return an error message for any dates that are outside of this period.
The EXPAND ON clause returns results for any date whether they are outside the system calendar period or not.
- **COMPATIBLE**
The first week of every year for this calendar is January 1 to January 7.
All the business calendar UDFs consider the CalendarPeriod to be 1900-01-01 to 2100-12-31, and return an error message for any dates that are outside of this period.
The EXPAND ON clause returns results for any date whether they are outside the system calendar period or not.

See *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211 for complete documentation of these calendars.

Business Calendar UDFs

Vantage supports a number of business calendar UDFs that you can use to return various calendar values.

See *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211 for complete documentation of these UDFs.

SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL

Definition of Isolation Level

The ANSI SQL:2011 standard defines isolation level as follows: “The isolation level of an SQL-transaction defines the degree to which the operations on SQL-data or schemas in that SQL-transaction are affected by the effects of and can affect operations on SQL-data or schemas in concurrent SQL-transactions” (*International Standard ISO/IEC 9075-2:2011(E), Part 2: Foundation (SQL/Foundation)*, 2011, pages 133 and 134) Note that isolation level is a concept related to concurrently running transactions and how well their updates are protected from one another as a system processes their respective transactions. Also note that the definition for isolation used in the original definition of the ACID properties of transactions (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142) is synonymous with the definition for serializable, so it does not allow different *levels* of isolation for transactions: a transaction is either isolated or it is not isolated.

The standard defines three different phenomena that have different characteristics for various transaction isolation levels. These phenomena are defined in the following table, which is adapted from a list in the ANSI SQL:2011 standard (*International Standard ISO/IEC 9075-2:2011(E), Part 2: Foundation (SQL/Foundation)*, 2011, pages 134 and 135).

| Phenomenon | Abbreviation | Description |
|---------------------|--------------|---|
| Dirty Read | P1 | Transaction <i>t1</i> modifies a row. Transaction <i>t2</i> then reads that row before <i>t1</i> performs a COMMIT request. If <i>t1</i> then performs a ROLLBACK request, <i>t2</i> will have read a row that was never committed, so that row can be considered never to have existed. |
| Non-Repeatable Read | P2 | Transaction <i>t1</i> reads a row. Transaction <i>t2</i> then updates or deletes that row and performs a COMMIT request. If <i>t1</i> then attempts to read the row again, the database might return the updated value or the row might have been deleted. |
| Phantom Read | P3 | Transaction <i>t1</i> reads a set of rows <i>n</i> that satisfy some search condition. Transaction <i>t2</i> then executes SQL requests that generate one or more rows that also satisfy the search condition used by transaction <i>t1</i> . If transaction <i>t1</i> then repeats its initial read using the same search condition, the database returns a different set of rows. |

The following table, which is also adapted from the ANSI standard (Table 8, *International Standard ISO/IEC 9075-2:2011(E), Part 2: Foundation (SQL/Foundation)*, 2011, pages 134 and 135), explains how the behavior of these phenomena differs depending on the transaction isolation level.

| Isolation Level | P1 | P2 | P3 |
|------------------|--------------|--------------|--------------|
| READ UNCOMMITTED | possible | possible | possible |
| SERIALIZABLE | not possible | not possible | not possible |

A footnote to this table in the ANSI SQL:2011 standard notes the following: “The exclusion of these phenomena for SQL-transactions executing at isolation level SERIALIZABLE is a consequence of the requirement that such transactions be serializable.” While this statement borders on being a tautology, it can, and should, also be read as indicating that any critical database transactions should always be run under an isolation level of SERIALIZABLE.

Default Session Transaction Isolation Level

The initial default transaction isolation level for any session is SERIALIZABLE. In this context, that is equivalent to saying the default read-only locking severity for any session is READ. This applies to all SELECT operations, whether they are standalone SELECT requests, ordinary SELECT subqueries, or SELECT subqueries embedded within DELETE, INSERT, or UPDATE requests.

SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL permits you either to set the default read-only locking severity for SELECT operations embedded within DELETE, INSERT, or UPDATE requests for the current session to ACCESS or to set it back to READ after it has been set to ACCESS.

This means that changing the session-level isolation level to READ UNCOMMITTED affects the default read-only locking severity for any SELECT requests embedded within DELETE, INSERT, MERGE, or UPDATE requests, making the default lock severity ACCESS, but only when the value for the DBS Control flag AccessLockForUncomRead is set TRUE. When AccessLockForUncomRead is set FALSE, the default read-only locking severity for those SELECT operations is READ.

Note that you must both specify READ UNCOMMITTED and set the DBS Control flag AccessLockForUncomRead to TRUE for Vantage to downgrade the default locking severity for embedded SELECT requests within data manipulating requests from READ to ACCESS for the session. In this context, a data manipulating request is any DELETE, INSERT, MERGE, or UPDATE request.

| IF you set the isolation level to this value ... | AND set the value for the DBS Control flag AccessLockForUncomRead to this value ... | THEN ... |
|--|---|--|
| READ UNCOMMITTED RU | FALSE | the lock severity of embedded SELECT requests is not downgraded from READ to ACCESS. |
| SERIALIZABLE SR | TRUE | |

The following table presents this information in a slightly different way.

| IF the transaction isolation level is ... | AND the DBS Control AccessLockForUncomRead flag is set ... | THEN the default locking severity for outer SELECT and ordinary SELECT subquery operations is ... | AND the default locking severity for SELECT operations embedded within a DELETE, INSERT, MERGE, or UPDATE request is ... |
|---|--|---|--|
| SERIALIZABLE | FALSE | READ | READ |
| | TRUE | | READ |
| READ UNCOMMITTED | FALSE | | READ |
| | TRUE | | ACCESS |

Note that the Optimizer implicitly specifies the locking level (either row hash, view, or table) for any read-only operation, irrespective of whether the default session isolation level is SERIALIZABLE or READ UNCOMMITTED. Isolation levels affect locking *severities*, not locking levels.

You can only specify locking levels explicitly using the LOCKING request modifier. For details, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Why Default to ACCESS Locks For All SELECT Operations Embedded Within DELETE, INSERT, and UPDATE Requests in a Session?

Unlike business-critical applications such as financial, order entry, warehouse management, and others that require absolute certainty in the results of their read-only operations on tables, some applications, such as Customer Relationship Management (CRM), sometimes do not require the same level of absolute certainty in the data they retrieve from the database because they are interested only in a snapshot, “statistical” picture of the current state of the database.

Because of this, CRM and other similar applications can tolerate dirty read operations, so they can afford to relinquish a degree of accuracy in their database read operations in exchange for enhanced transaction concurrency (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for details).

A *dirty read* is an operation that might read data written to the database by transactions that are not yet committed. There is always a chance that any transaction will fail for some reason, and if this occurs, the system rolls back any updates it might have made. If those updates include data that has been read by another concurrently running transaction, then that read is said to have been dirty, because as far as the database is concerned, it never really existed. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for details about transaction integrity and some of the problems that relaxing it can bring.

It can be beneficial to the performance of applications that can tolerate dirty reads to be able to specify a default session-wide locking level that permits them rather than downgrading locks on a request-by-request basis, which can be burdensome. ACCESS-level locking achieves better performance than READ-level locking by enhancing transaction concurrency.

This is a very important consideration, and it should not be taken lightly. The overall qualitative workload of the session must be examined carefully before making the determination of whether to default to ACCESS-level locking for read-only operations or not. For example, consider a session in which a MultiLoad import job is running. Because of the way MultiLoad updates table rows during its acquisition phase (see *Teradata® MultiLoad Reference*, B035-2409), using ACCESS locks to query the target table of the MultiLoad job during an acquisition phase can produce extremely inaccurate result sets. In this case, the results probably would not provide even a reasonable impression of the table data.

The SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement provides a mechanism for doing that, allowing you to indirectly substitute ACCESS locks for READ locks for selected read-only SELECT operations in the current session (restricted to SELECT subqueries embedded within DELETE, INSERT, or UPDATE requests). The initial default for any session remains SERIALIZABLE (meaning READ level severity locking), but you can toggle the default read-only locking severity to READ UNCOMMITTED (meaning ACCESS level severity locking) either interactively or, more persistently, within your application code.

Correspondence Between ANSI SQL Transaction Isolation Levels and Database Locks

The correspondence between ANSI SQL transaction isolation levels and database locks is not 1:1 because the authors of the isolation levels standard intended to define its rules in terms that were independent of their implementation. In other words, they wanted to define a standard for transaction isolation levels that could apply to many different implementations, not just those based on locking, in particular the two-phase locking protocol (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for information about 2PL).

The only direct correspondence is between the isolation level SERIALIZABLE and the standard commercial database management system implementation of two-phase locking (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for a description of two-phase locking). All other isolation levels in the standard are defined in terms of what are called *phenomena*, of which the dirty read is one example. Because the database permits dirty reads through the use of ACCESS locking, there is a seemingly good correspondence between the ANSI isolation level READ UNCOMMITTED and Vantage ACCESS locks. Because the ANSI definition of READ UNCOMMITTED is written in the English language rather than formally defined, however, its details are open to interpretation.

Related Information

See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for information about transaction processing in general and for a description of the difference between locking levels and locking severities.

See SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for the syntax of this statement and examples of its use.

SET SESSION FUNCTION TRACE

Trace Library Function Calls

Teradata provides several library functions to enable programmers to debug their functions and external procedures, or for any other useful purposes.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for further information about these UDF- and external procedure-related trace library function calls.

Possible Uses for SET SESSION FUNCTION TRACE

The intended use of this statement is to enable or disable the output capability of UDFs and external procedures. As such, its targeted use is for diagnostic analysis when you are developing a new UDF or external procedure.

You can also use SET SESSION FUNCTION TRACE for any other trace output that either is not possible, or is inconvenient to perform, using other means.

SET SESSION FUNCTION TRACE Performance Overhead

When function trace processing is enabled, additional performance overhead is incurred for each request that uses UDFs or external procedures. This occurs because Vantage saves the output of the function in the specified materialized global trace table.

Function Trace Output Table

You must declare a global temporary trace output table to receive trace data when you enable function tracing. The specified table name must be the name of an existing global temporary trace table. See [CREATE GLOBAL TEMPORARY TRACE TABLE](#) for the required definition of this table.

SET TIME ZONE

Setting the Permanent Default Time Zone for Your System

You can set the system default time zone for your system using the DBS Control utility flags `SystemTimeZoneHour` and `SystemTimeZoneMinute`. If you are using time zone strings to convert to Daylight Saving Time and back automatically, you might also need to change the setting for the DBS Control flag `TimeDateWZControl`.

For information on using DBS Control to set the default time zone for your site, see *Teradata Vantage™ - Database Utilities*, B035-1102.

UTC Definition

UTC represents the Coordinated Universal Time, formerly referred to as Greenwich Mean Time, or GMT.

Formally, the value for a local time with respect to UTC is defined as follows:

$UTC = \text{local time} - \text{time_zone_displacement}$

where the value for *time_zone_displacement* is the time that must be subtracted from local time to produce the value for UTC, which is arbitrarily defined as zero hours displacement from the Greenwich meridian.

General Rules for Using SET TIME ZONE

The following rules apply to using SET TIME ZONE.

- You can set the default time zone for a session or user by specifying either a numeric offset from UTC or by specifying a time zone string literal.
- The range of time zone displacements permissible is -12:59 through +14.00 inclusive.
- The form for time zone displacement values is always INTERVAL HOUR TO MINUTE.
- The time zone string is associated with a session or user, not a time zone displacement. When a time zone displacement is needed in a request, that string plus a `TIMESTAMP` value or a `TIME` value, depending on which is appropriate, is used to determine the time zone displacement needed for a particular usage.

Rules and Restrictions for Specifying Simple Expressions for Time Zones

The following rules and restrictions apply to specifying a simple expression as a time zone value.

- The simple expression you specify must be a constant; otherwise, the system aborts the request and returns an error to the requestor.
- The current session time zone can be either a time zone displacement or time zone string.

| IF the current session time zone is ... | THEN Vantage ... |
|---|--|
| a time zone displacement | uses that time zone displacement. |
| a time zone string | determines the time zone displacement based on the specified time zone string and the UTC TIMESTAMP value for the source expression. See Daylight Saving Time and Time Zone Strings Specified As Time Zone Strings for a comprehensive list of valid time zone strings. |

- Vantage implicitly converts the simple expression you specify as needed and if allowed, to a time zone displacement or time zone string depending on its data type as defined in the following table.

| This expression data type ... | Is implicitly converted as follows ... |
|--|--|
| BIGINT BYTEINT INTEGER SMALLINT | CAST(CAST (<i>expression</i> AS INTERVAL HOUR(2)) AS INTERVAL HOUR(2) TO MINUTE) |
| DECIMAL NUMERIC where scale = 0 | CAST(CAST (<i>expression</i> AS INTERVAL HOUR(2)) AS INTERVAL HOUR(2) TO MINUTE) |
| DECIMAL NUMERIC where scale > 0 | CAST(CAST((<i>expression</i>)*60 AS INTERVAL MINUTE(4)) AS INTERVAL HOUR(2) TO MINUTE) |
| INTERVAL HOUR(2) TO MINUTE | no conversion |
| INTERVAL HOUR(<i>n</i>) TO MINUTE, where $n \geq 2$ | CAST(<i>expression</i>) AS INTERVAL HOUR(2) TO MINUTE) |
| INTERVAL HOUR INTERVAL DAY If the INTERVAL DAY value you specify is anything other than INTERVAL '0' DAY or INTERVAL '-0' DAY, it eventually fails because it becomes too large to be a time zone displacement. INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL HOUR INTERVAL HOUR TO SECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL SECOND | CAST(<i>expression</i>) AS INTERVAL HOUR(2) TO MINUTE) |

| This expression data type ... | Is implicitly converted as follows ... |
|--|--|
| CHARACTER with CHARACTER SET UNICODE | Attempt to CAST(CAST(se AS INTERVAL HOUR(2)) AS INTERVAL HOUR(2) TO MINUTE). If an error occurs for the casts, attempt to CAST(<i>expression</i> AS INTERVAL HOUR(2) TO MINUTE). If still unable to cast expression, Vantage treats the character value as a time zone string. |
| CHARACTER with CHARACTER SET ¼ UNICODE | TRANSLATE(expression USING <i>source_repertoire_name</i> _TO_Unicode) where <i>source_repertoire_name</i> is the server character set of expression. If an error occurs with the translated value, it is then handled as in the previous row for a character value with CHARACTER SET UNICODE replacing expression with the translation of expression. |
| anything else | The system aborts the request and returns an error to the requestor. |

- Specifying a simple expression without first specifying TIME ZONE is valid syntax and has the same effect as specifying TIME ZONE *simple_expression*.

Daylight Saving Time and Time Zone Strings Specified As Time Zone Strings

The ANSI SQL:2011 concept of a time zone does not properly handle the time changes that occur when Daylight Saving Time begins and ends. To deal with the Daylight Saving Time concept, Teradata defines a time zone as a set of rules for converting between UTC and the time in the local time zone. Because the rules and application of Daylight Saving Time vary widely, significant flexibility is required to maintain them correctly.

Daylight Saving Time is the term used in American English. British English uses the equivalent term *Summer Time*.

The database uses a time zone string submitted using an SQL SET TIME ZONE request or by using the `tdlocaledef` utility (for more information, see *Teradata Vantage™ - Database Utilities*, B035-1102). The time zone string names a rule set to be applied for a session or user. The SET TIME ZONE statement and the `tdlocaledef` utility use the same time zone strings except that `tdlocaledef` must also specify a rule that is to be associated with the time zone string.

The database uses the time zone rule string as an input parameter to the system-defined UDF named `GetTimeZoneDisplacement`, which is contained in the SYSLIB database. For information about the `GetTimeZoneDisplacement` UDF, including the rules for adding and modifying time zone strings and their associated rules, see *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211. The string is interpreted to determine the hour and minute offsets for a session or for specific users when they log onto the database. For each such name, the UDF accesses the rules and information it requires to

know what the displacement is depending on the date and time on which it changes from Daylight Saving Time and back.

Note:

Only those time zone strings that are *not* defined in terms of GMT enable automatic adjustments for Daylight Saving Time. The GMT time zone strings are designed to be used for regions and time zones that do not follow Daylight Saving Time.

The UDF is defined with the `tzs` parameter and returns its time zone displacement result with an INTERVAL HOUR TO MINUTE data type.

See SET TIME ZONE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for a complete list of the valid time zone strings.

The following table documents how the system-defined UDF returns values to the database.

| IF the ... | THEN the database ... |
|---|---|
| result is valid | returns a set of rules for every time zone string that describes a valid standard time zone displacement, daylight saving time zone displacement, start time for daylight savings time, and an end timestamp. |
| value for <code>tzs</code> is not valid | aborts the request and returns an error to the requestor. |
| date is old enough that the system has no information about Daylight Saving Time for it | |

Time Zone Strings

The following list presents the valid time zone string entries alphabetically, with the exception of the Greenwich Mean Time (GMT/Coordinated Universal Time (UTC) parameters, which are presented last. Regions that do not follow Daylight Savings Time are only represented by GMT values.

| Time Zone Strings | | |
|---|--|--|
| <ul style="list-style-type: none"> • Africa Egypt • Africa Morocco • Africa Namibia • America Alaska • America Aleutian • America Argentina • America Atlantic • America Brazil • America Central • America Chile • America Cuba | <ul style="list-style-type: none"> • Asia Omsk • Asia Syria • Asia Vladivostok • Asia West Bank • Asia Yakutsk • Asia Yekaterinburg • Australia Central • Australia Eastern • Australia Western • Europe Central • Europe Eastern | <ul style="list-style-type: none"> • GMT-5 • GMT-4 • GMT-3 • GMT-2 • GMT-1 • GMT • GMT+1 • GMT+2 • GMT+3 • GMT+3:30 • GMT+4 |

| Time Zone Strings | | |
|---|--|--|
| <ul style="list-style-type: none"> • America Eastern • America Mountain • America Newfoundland • America Pacific • America Paraguay • America Uruguay <ul style="list-style-type: none"> • Argentina • Asia Gaza • Asia Iran • Asia Iraq • Asia Irkutsk • Asia Israel • Asia Jordan • Asia Kamchatka • Asia Krasnoyarsk <ul style="list-style-type: none"> • Asia Lebanon • Asia Magadan | <ul style="list-style-type: none"> • Europe Kaliningrad • Europe Moscow • Europe Samara • Europe Western • Indian Mauritius • Mexico Central • Mexico Northwest • Mexico Pacific • Pacific New Zealand • Pacific Samoa • GMT-11 • GMT-10 • GMT-9 • GMT-8 • GMT-7 • GMT-6:30 • GMT-6 | <ul style="list-style-type: none"> • GMT+4:30 • GMT+5 • GMT+5:30 • GMT+5:45 • GMT+6 • GMT+6:30 • GMT+7 • GMT+8 • GMT+8:45 • GMT+9 • GMT+9:30 • GMT+10 • GMT+11 • GMT+11:30 • GMT+12 • GMT+13 • GMT+14 |

SQL HELP Statements

Teradata SQL provides several powerful statements for system administrators, database and security administrators, and application programmers.

These statements fall into three categories:

- Help about database object definitions
- Help about SQL statement and utility command syntax
- Summaries of database object definition statement text

For HELP and SHOW statement descriptions, syntax, and examples, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *SQL Quick Reference*, B035-1510.

About HELP Statements

HELP statements include:

- SQL HELP statements
- Online HELP statements

SQL HELP Statement Reports

The various HELP statements return reports about attributes for named database objects and processes. The reports returned by these statements can be useful to database designers and administrators who need to fine tune object definitions, such as indexes or columns, for example, when revising data typing to eliminate the necessity of ad hoc conversions.

HELP (Online)

The HELP online statements return information about SQL statements and client utility command syntax. See [HELP \(Online Form\)](#).

HELP Usage Notes

You use the HELP statements to view the attributes of a database object or session.

The output of HELP results usually are too wide to be shown on a single screen, and are sometimes wider than 132 characters. To adjust for this, you can format the output of your HELP reports using BTEQ formatting commands. For information about these formatting commands, see *Basic Teradata® Query Reference*, B035-2414.

Various examples of HELP queries are given in the following pages. The contents of HELP reports are explained in the following subsections.

For more information, refer to the appropriate HELP statement:

- [HELP CAST](#)
- [HELP COLUMN](#)
- [HELP CONSTRAINT](#)
- [HELP DATABASE](#)
- [HELP ERROR TABLE](#)
- [HELP FUNCTION](#)
- [HELP HASH INDEX](#)
- [HELP INDEX](#)
- [HELP JOIN INDEX](#)
- [HELP MACRO](#)
- [HELP METHOD](#)
- [HELP PROCEDURE](#)
- [HELP SESSION](#)
- [HELP STATISTICS \(Optimizer Form\)](#)
- [HELP STATISTICS \(QCD Form\)](#)
- [HELP TRIGGER](#)
- [HELP TYPE](#)
- [HELP VOLATILE TABLE](#)

Formatting HELP Report Results

You can format HELP reports using the formatting commands available for the API from which you make a HELP request.

Note:

Unless otherwise noted for a specific HELP statement type, report fields follow a standard format consistent with the field data type. For example, UDT Name VARCHAR(61) fields are formatted as X(61).

For example, this BTEQ input returns session information in the format demonstrated by the following report.

```
Teradata BTEQ 14.00.00.04 for LINUX.
Copyright 1984-2012, Teradata Corporation. ALL RIGHTS RESERVED.
Enter your logon or BTEQ command:
.logon dbc

.logon dbc
Password:
```

```
.foldline
.sidetitles
HELP SESSION;
*** Help information returned. One row.
*** Total elapsed time was 1 second.

*** Warning: Report has more than 100 column(s).
        Only the first 100 column(s) will be displayed.
```

```

        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-DDDBHH: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 ?

```

Object Name and Title Data in HELP Reports

Translating HELP Report Data into the Session Character Set

Attribute data reported as the result of a HELP request is translated from the data dictionary into the client session character set according to the following general guidelines, except as noted otherwise for individual HELP statements.

Most HELP attribute data is translated by the system from UNICODE into the session character set. For example:

- Names and lists of names
- Titles
- Text fields, for example, Comment
- Numeric values (translated from binary to UNICODE to the session character set)
- The Format attribute

For some HELP attribute data, the system does not translate from UNICODE:

- CHARACTER(1) and CHARACTER(2) fields, such as TVM Code or Data Type are translated from CHARACTER SET LATIN into the session character set.

- Attributes that return one of a fixed group of values, for example State, which returns a value using UTF8(24).
- Untranslatable character data that is returned as the UEscape character followed by a hex representation.
- Legacy names that are not changed and remain in the form of their language support mode.

Rules for HELP Report Names and Titles

When you execute a HELP statement, the system returns the following name and title information for database objects.

For names, such as column names:

- Primary name, for example, Column Name, based on object definitions that exist in the data dictionary. CHAR(30) name format for support of legacy applications.
- Column Dictionary Name
- Column SQL Name
- Column Name UEscape

For titles of objects which support a title, such as tables and columns:

- Title, up to VARCHAR(256)
- Dictionary Title
- SQL Title
- Title UEscape

These fields help you to manage the differences between the data dictionary representation of name or title and the representation in the session (client) character set. You can use SQL Name and SQL Title values directly in subsequent SQL statements.

The system determines the contents of HELP name fields based on the rules in the following table.

| Field | Description |
|--|---|
| <ul style="list-style-type: none"> • <i>object_type</i> Dictionary Name • Dictionary Title | <p>The data dictionary name or title, translated to the session character set.</p> <p>Object names: VARCHAR(128)</p> <p>Titles: VARCHAR(256)</p> <p>If any character in the name or title is not translatable into the session character set, the system converts it into the replacement character for the session character set. For example, when the session character set is ASCII, the system replaces each untranslatable character with the SUBSTITUTE character 0x1A:</p> <p style="text-align: center;">^Z</p> <p>where the ^Z is used to represent an unprintable control character.</p> |
| <ul style="list-style-type: none"> • <i>object_type</i> SQL Name • SQL Title | <p>The name or title as it exists in the data dictionary, converted to a string that can be directly used in an SQL request, expressed in the shortest form (with matching case) for the current session character set.</p> <p>Object Names: VARCHAR(644)</p> |

| Field | Description |
|--|---|
| | <p>If all characters in an object name are translatable into the session character set, the SQL Name for the object name is the same as the Dictionary Name. If SQL statement syntax requires quotation marks, the SQL Name field begins and ends with QUOTATION MARK (U+0022) characters.</p> <p>For information on using quotation mark characters with object names, see SQL Fundamentals.</p> <p>If the object name contains characters that are not translatable into the current session character set, the system expresses the SQL Name as a UNICODE delimited identifier, although without the normal closing UEscape clause, starting with LATIN CAPITAL LETTER U (U+0055), followed by AMPERSAND (U+0026), followed by QUOTATION MARK (U+0022), and ends with a QUOTATION MARK (U+0022).</p> <p>For example:</p> <pre>U&"\7A0E\91D1"</pre> <p>Titles: VARCHAR(1284)</p> <p>The system expresses the SQL Title similarly to the SQL Name, except that a title is a string, not a name, so:</p> <ul style="list-style-type: none"> • If all the characters in the title are translatable into the current session character set, the SQL Title field value begins and ends with an APOSTROPHE (U+0027). • If a title contains characters that are not translatable into the session character set, the system expresses the SQL Title as a UNICODE delimited literal, which places a LATIN CAPITAL U (U+0055) and then an AMPERSAND (U+0026) before the beginning APOSTROPHE. <p>The system replaces each character that is not translatable into the session character set with the corresponding UNICODE identifier. Each UNICODE identifier is preceded by the default delimiter character for the session character set.</p> <p>For example, when specifying the Polish currency as 'Polish_Zloty,' the LATIN SMALL LETTER L WITH STROKE, shown as an "ł" in "zloty" is expressed as \0142, for example:</p> <pre>U&'Polish_Z\0142oty '</pre> |
| <ul style="list-style-type: none"> • object_type UEscape • Title UEscape | <p>VARCHAR(1)</p> <p>Indicates the delimiter character that precedes each UNICODE identifier substituted for an untranslatable character in the SQL Name/SQL Title. For example:</p> <pre>Table UEscape: \</pre> <p>If all characters in the SQL Name/SQL Title are translatable into the session character set, the SQL Name/SQL Title field does not include any delimiters and the UEscape field is NULL.</p> <pre>Table UEscape: ?</pre> <p>where the ? character represents NULL.</p> |

Example Object Name in HELP Output

For example, you have a Japanese language table named テーブル.

If you submit a HELP DATABASE statement for the database that contains the table, the HELP output includes object names/titles, followed by the 3 associated name/title fields, the content of which differs based on the session character set.

For a UTF8 or UTF16 session, where the table name translates into the session character set, the table portion of the HELP output appears similar to the following, with some output lines omitted for clarity:

Table Name: テーブル

Table Dictionary Name: テーブル

Table SQL Name: テーブル

Table UEscape: ?

Title: BigTable1

Dictionary Title: BigTable1

SQL Title: 'BigTable1'

Title UEscape: ?

where the ? value for UEscape represents NULL, which is used because all SQL Name and SQL Title characters are translatable into the session character set and no delimiter is required.

For an ASCII session, where the table name does not translate into the session character set, the table portion of the HELP output appears similar to the following:

Table Name: ^Z^Z^Z^Z

Table Dictionary Name: ^Z^Z^Z^Z

Table SQL Name: U&"\FF83\FF70\FF8C\FF9E\FF99"

Table UEscape: \

Table Title: BigTable1

Dictionary Title: BigTable1

SQL Title: 'BigTable1'

Title UEscape: ?

where:

| Output | Description |
|--|--|
| Table Dictionary Name: ^Z^Z ^Z^Z | Each ^Z represents an ASCII replacement character, 0x1A, used to replace the 4 untranslatable characters, テーブル. Note: The system uses ^ to represent unprintable control characters. |
| Table SQL Name: U&"\FF83\FF70\FF8C\FF9E\FF99" | The SQL Name begins with U& to indicate that it is a UNICODE delimited identifier, that is, it contains untranslatable characters. The sequence \FF83\FF70\FF8C\FF9E\FF99 is the set of UNICODE identifiers for the 4 untranslatable characters, preceded by the default delimiter character, in this case, \. The string is enclosed in quotation marks so that the delimiter characters it contains can be used in an SQL request. Note: The system does not return the UEscape clause, which normally closes a UNICODE delimited identifier. If you want to use a UNICODE delimited identifier in an SQL request, you need to add the UEscape clause. See Using an SQL Name or SQL Title Value in an SQL Request . |
| Table UEscape: \ | Indicates the delimiter character used to separate the UNICODE identifiers in the SQL Name. |

Note:

Output for title entries is similar, for example, Dictionary Title and SQL Title. SQL Titles are enclosed in single quotation marks.

Determining the SQL Name or SQL Title Delimiter Character

When a HELP statement returns an SQL Name or SQL Title, the system replaces any character that is not translatable into the client session character set with the UNICODE identifier for the character. Each UNICODE identifier is preceded with a delimiter character.

The system uses one of the following delimiter characters in SQL Name and SQL title, depending on availability in the session character set, in the order shown:

- BACKSLASH (U+005C)
- The YEN SIGN (U+00A5) or WON SIGN (U+20A9), depending on which is present in the session character set at 0x5C.
- NUMBER SIGN (U+0023), which is present in all supported session character sets.

The UEscape field that follows each SQL Name field identifies the delimiter used.

Using an SQL Name or SQL Title Value in an SQL Request

You can use the SQL Name or SQL Title of an object in an SQL request when you cannot directly use the data dictionary representation because not all characters are available in the session character set.

For example, you can use `HELP DATABASE` to look in the `mydb` database for the name of a table you want to drop:

```
HELP DATABASE mydb;
... Table SQL Name U&"table_\4E00_name"
      Table UEscape          \
```

The SQL Name returned by the `HELP DATABASE` statement is a UNICODE delimited identifier, which includes an untranslatable character that the system expresses as `\4E00`.

If you want to use a UNICODE delimited identifier from the SQL Name as part of an SQL request, you must:

1. Add the closing `UEScape` clause to the table name taken from the SQL Name field.
2. Specify the delimiter character (`\`)

For example:

```
Drop Table U&"table_\4E00_name" UEscape '\';
```

If the SQL Name is not a UNICODE delimited identifier, you can use the name in an SQL request as it appears in the `HELP` output, without specifying the `UEScape` phrase and delimiter.

Data Type Codes

The following table lists the data type codes reported by `HELP` statements and the data types they represent.

| Data Type Code | Data Type |
|----------------|--------------------------|
| ++ | TD_ANYTYPE |
| A1 | ARRAY (one dimensional) |
| AN | ARRAY (multidimensional) |
| I8 | BIGINT |
| BO | BINARY LARGE OBJECT |
| BF | BYTE |
| BV | BYTE VARYING |
| I1 | BYTEINT |

| Data Type Code | Data Type |
|----------------|---|
| CF | CHARACTER (fixed) |
| CV | CHARACTER (varying) |
| CO | CHARACTER LARGE OBJECT |
| D | DECIMAL |
| DA | DATE |
| F | DOUBLE PRECISION DOUBLE PRECISION, FLOAT, and REAL are different names for the same data type. |
| F | FLOAT FLOAT, DOUBLE PRECISION, and REAL are different names for the same data type. |
| I | INTEGER |
| DY | INTERVAL DAY |
| DH | INTERVAL DAY TO HOUR |
| DM | INTERVAL DAY TO MINUTE |
| DS | INTERVAL DAY TO SECOND |
| HR | INTERVAL HOUR |
| HM | INTERVAL HOUR TO MINUTE |
| HS | INTERVAL HOUR TO SECOND |
| MI | INTERVAL MINUTE |
| MS | INTERVAL MINUTE TO SECOND |
| MO | INTERVAL MONTH |
| SC | INTERVAL SECOND |
| YR | INTERVAL YEAR |
| YM | INTERVAL YEAR TO MONTH |
| N | NUMBER |
| D | NUMERIC |
| PD | PERIOD(DATE) |
| PT | PERIOD(TIME(n)) |
| PZ | PERIOD(TIME(n) WITH TIME ZONE) |
| PS | PERIOD(TIMESTAMP(n)) |

| Data Type Code | Data Type |
|----------------|---|
| PM | PERIOD(TIMESTAMP(n) WITH TIME ZONE) |
| F | REAL REAL, DOUBLE PRECISION, and FLOAT are different names for the same data type. |
| I2 | SMALLINT |
| AT | TIME |
| TS | TIMESTAMP |
| TZ | TIME WITH TIME ZONE |
| SZ | TIMESTAMP WITH TIME ZONE |
| UT | USER-DEFINED TYPE (all types) |
| XM | XML |

TVM Kind Codes

The following table lists and defines the TVM Codes returned in HELP statement reports.

| Code | Description |
|------|---|
| A | Aggregate function. |
| B | Combined aggregate and ordered analytical function. |
| C | Table operator parser contract function |
| D | JAR. |
| E | External procedure. |
| F | Standard function |
| G | Trigger |
| H | Instance method or constructor method. |
| I | Join index. |
| J | Journal |
| L | User-defined table operator |
| M | Macro |
| N | Hash index. |
| O | Nonpartitioned table with no primary index |
| P | SQL procedure. |

| Code | Description |
|------|---|
| Q | Queue table. |
| R | Table function. |
| S | Ordered analytical function. |
| T | Table with a primary index, partitioning, or both |
| U | User defined data type |
| V | View. |
| X | Authorization object. |
| Y | GLOP set. |

HELP CAST

HELP CAST Attributes

The following table lists the attributes reported by the HELP CAST statement:

| Attribute | Data Type | Nullable? | Description |
|-------------------|--------------|-----------|--|
| Source | VARCHAR(31) | No | The name of the source type. Because the containing database is always <i>SYSUDTLIB</i> , it is not reported. |
| Target | VARCHAR(31) | No | The name of the target type. Because the containing database is always <i>SYSUDTLIB</i> , it is not reported. |
| Cast Routine | VARCHAR(31) | No | If UDT_name is a structured UDT, this is the specific name of the external routine that provides the casting functionality. If UDT_name is a distinct UDT and its casting functionality was system-generated, this is the word <i>System</i> . Because the containing database is always <i>SYSUDTLIB</i> , it is not reported. |
| As Assignment | VARCHAR(3) | No | <ul style="list-style-type: none"> • NO specifies that the AS ASSIGNMENT option was not specified for this cast definition. If the code is NO, implicit casting is <i>not</i> supported for this external routine. • YES specifies that the AS ASSIGNMENT option was specified for this cast definition. If the code is YES, implicit casting is supported for this external routine if the value for the DisableImplCastForSysFuncOp DBS Control flag is set to 00x0. For details, see Using the AS ASSIGNMENT Clause To Make a Cast Operation Implicitly Invokable . |
| Dictionary Source | VARCHAR(128) | No | The attributes shown in this section are based upon and supersede corresponding older attributes, Source, Target, and Cast Routine, while providing additional functionality. The older attributes are retained for compatibility with legacy applications. For details, see the topics beginning with Object Name and Title Data in HELP Reports . |
| SQL Source | VARCHAR(644) | No | |
| Source Uescape | VARCHAR(1) | Yes | |
| Dictionary Target | VARCHAR(128) | No | |
| SQL Target | VARCHAR(644) | No | |
| Target Uescape | VARCHAR(1) | Yes | |

| Attribute | Data Type | Nullable? | Description |
|-------------------------|--------------|-----------|-------------|
| Dictionary Cast Routine | VARCHAR(128) | No | |
| SQL Cast Routine | VARCHAR(644) | No | |
| Cast Routine Uescape | VARCHAR(1) | Yes | |

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 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, which you can tell 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 never system-generated, while the system always 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](#) for details

Example: All Castings For A UDT That Has Multiple Casting Pairs

The following example extends [CREATE CAST and REPLACE CAST](#) to show what the output of running `HELP CAST euro` (see [CREATE CAST and REPLACE CAST](#) and [CREATE CAST and REPLACE CAST](#)) 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 (which you can tell because 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-to-us_dollar* cast because it was not defined with the AS ASSIGNMENT option.

Related Information

See the documentation for the following statements and manual for additional information about creating castings for UDTs.

- [CREATE CAST and REPLACE CAST](#)
- [SHOW object](#)

- *Teradata Vantage™ - SQL External Routine Programming, B035-1147*

HELP COLUMN

HELP COLUMN With a FROM Clause

If a HELP COLUMN statement contains a FROM clause, you can reference one or more tables, join indexes, and hash indexes in the same statement, as long as each column name is fully qualified (that is, preceded by the table name in the form “tablename.columnname”).

This does not apply when the referenced object is a data table name or an error table name.

HELP COLUMN For a Data Table Or Error Table

If you use either the FROM ERROR TABLE FOR *data_table_name* syntax or the example for the FROM *error_table_name* syntax, then the information returned is for the column you specify in the error table associated with *data_table_name*, not from the base table specified by *data_table_name*.

The HELP COLUMN information and attributes reported by either of these syntaxes is identical to that returned by the other HELP COLUMN syntaxes, only for an error table rather than for a base data table.

See [CREATE ERROR TABLE](#) for more information about error tables.

Also see [HELP ERROR TABLE](#).

HELP COLUMN and Indexes

If the column for which information is requested defines a single-column index, then that is reported along with the uniqueness and type of index. HELP COLUMN returns index information for single-column indexes *only*. For multiple-column indexes, the columns are reported as not being used in an index.

HELP COLUMN does not provide information about whether a column is used in the partitioning expression of a partitioned primary index. Use SHOW TABLE (see [SHOW object](#)) or the system view *DBC.IndexConstraintsV* (see *Teradata Vantage™ - Data Dictionary*, B035-1092 for further information) to obtain the partitioning information.

HELP COLUMN returns the information indicated in [HELP COLUMN Information and Attributes](#).

HELP COLUMN Information and Attributes

All HELP return columns are subject to formatting using the rules described for their Export Width specification in Field Mode.

All CHAR and VARCHAR fields are affected, although the effects on VARCHAR in Record mode may not be noticeable.

HELP COLUMN and Column-Partitioned NoPI Tables and Join Indexes

Use HELP COLUMN requests to obtain partitioning information for a table or join index. You can also find information about indexes by using the system views *DBC.Indices[V][X]* and the data dictionary table *DBC.Indices*.

HELP COLUMN Attributes

The following table lists the attributes reported by a HELP COLUMN request.

| Attribute | Data Type | Nullable? | Description |
|---|---------------|-----------|--|
| Column Name | CHARACTER(30) | No | The name of a column for which HELP information is being returned. |
| Type See Data Type Codes for a list of the data type codes and descriptions. | CHARACTER(2) | Yes | <p>The data type for data in the column. With an explicit format, DATE is imported and exported according to that format, as indicated in the following bullets.</p> <ul style="list-style-type: none"> • With an Implicit format created in INTEGERDATE mode, the export format is 'YY/MM/DD'. • With an Explicit format created in ANSIDATE mode, the export format is 'YYYY-MM-DD'. <p>The character data types Latin, Unicode, Graphic, KanjiSJIS, and Kanji1 are distinguished with respect to CF and CV by their respective value for the Character Type column.</p> <p>The type codes GF and GV are no longer returned for columns defined as CHARACTER(n) CHARACTER SET GRAPHIC.</p> |
| Nullable | CHARACTER(1) | Yes | <p>Indicates whether the column specified in Column Name is nullable.</p> <ul style="list-style-type: none"> • N specifies that the column cannot be null. • Y specifies that the column can be null. |
| Format | CHARACTER(30) | Yes | <p>Display format for the data type. For a UDT, the displayed format is the format associated with the external type of the UDT. For data type display formats, see <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143.</p> |
| Max Length | INTEGER | Yes | <p>Maximum amount of storage in bytes. MaxLength is the length of the <i>external</i> CharFix representation of the column, defined as CHARACTER(MaxLength) in USING clauses and any other place this needs to be specified.</p> |

| Attribute | Data Type | Nullable? | Description |
|---------------------------|-----------|-----------|--|
| | | | <p>Note that while MaxLength is usually expressed as the internal size, presentation of the external size seems more appropriate for these data types.</p> <p>See <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143 for each individual DateTime and Interval to determine the MaxLength values.</p> <p>For CHARACTER data, Max Length is the length of the column declaration in bytes.</p> <p>For column names, the maximum length is 256 bytes; however, the enforced maximum column name length is 60 bytes.</p> <p>For a UDT, Max Length is the maximum length of the external type associated with the UDT formatted as -(10)9.</p> <p>When DATEFORM=ANSIDATE, this column should be described externally as CHARACTER(10).</p> <p>Host data characteristics for these data types should be interpreted as the equivalent of CF, or CHARACTER(n) data type.</p> <p>You can use the StatementInfo parcel to return the description of items within the Data Field of the corresponding indicator mode Record parcels.</p> <p>For example, the FastLoad .TABLE command can be executed in Indicator Mode to get the additional information of data type and length of all table attributes. That information can then be used to fetch the data correctly in any database release.</p> |
| Decimal Total Digits | SMALLINT | Yes | <p>If type is DECIMAL.</p> <p>For INTERVAL types, returns a value from 1 - 4 representing the number of digits in the leading field of the interval (number of non-fractional digits for INTERVAL SECOND).</p> <p>This column is null for TIME and TIMESTAMP data types.</p> |
| Decimal Fractional Digits | SMALLINT | Yes | <p>If type is DECIMAL.</p> <p>For TIME and TIMESTAMP types and all INTERVAL types with a SECOND field, this column returns a value of 0 - 6, indicating the fractional precision of seconds. For INTERVAL types without a SECOND field, the column returns null.</p> |
| Range Low | FLOAT | Yes | Always null. BETWEEN clause now included in column constraint. |
| Range High | FLOAT | Yes | Always null. BETWEEN clause now included in column constraint. |

| Attribute | Data Type | Nullable? | Description |
|-------------|--------------|-----------|---|
| Uppercase | CHARACTER(1) | Yes | <p>Returned if type is CHARACTER or VARCHAR.</p> <ul style="list-style-type: none"> • B specifies that column attributes for both UPPERCASE and CASESPECIFIC are defined. • C specifies that the column attributes for CASESPECIFIC is defined. • N specifies that column attributes for neither UPPERCASE nor CASESPECIFIC are defined. • U specifies that the column attributes for UPPERCASE is defined. |
| Table/View? | CHARACTER(1) | No | <p>Indicates whether the column specified by Column Name is from a table or a view.</p> <ul style="list-style-type: none"> • T specifies that the column is from a table definition. • V specifies that the column is from a view definition. |
| Indexed? | CHARACTER(1) | Yes | <p>Identifies if the column is single-column index.</p> <ul style="list-style-type: none"> • N specifies that the column is not used as a single-column index. <p>You can define multiple indexes on the same columns as long as they are different in some respect (value-ordered versus hash-ordered or nonunique primary index on some set of columns and a USI on the same columns).</p> <p>In this case, the codes reported for Unique? and Primary? are for the index with the lowest index id. For a nonunique partitioned primary index, a single column that also has a USI on that column displays N for Unique? and Y for Primary?</p> <ul style="list-style-type: none"> • Y specifies that the column is used as a single-column index. <p>A column can be a component of more than one index.</p> <p>Whether the value for this attribute is Y or N, the column might be a member of a column set that defines a composite index.</p> <p>Use HELP INDEX to see a list of all the primary and secondary indexes for a table.</p> |
| Unique? | CHARACTER(1) | Yes | <p>If the column defines a single-column index, indicates whether the index is unique or nonunique.</p> <ul style="list-style-type: none"> • N specifies that the index is not unique. • Y specifies that the index is unique. <p>If INDEXED? attribute is N, then returns null.</p> |

| Attribute | Data Type | Nullable? | Description |
|--|--------------|-----------|--|
| Primary? | CHARACTER(1) | Yes | <p>If the column defines a single-column index, is that index a primary index or a secondary index?</p> <ul style="list-style-type: none"> • P specifies that the column is a component of the primary index definition for the table. • S specifies that the column is a component of the secondary index definition for the table. <p>Multiple-column indexes return a null result. If INDEXED? attribute is N, then returns null.</p> |
| Title | VARCHAR (60) | Yes | The title (if a title exists) for the column about which HELP information is being returned. |
| Column Constraint | VARCHAR(255) | Yes | The text of the constraint clause, if a constraint is specified for the column. |
| Character Type | SMALLINT | Yes | <p>Returns the character data type for a CHAR or VARCHAR column.</p> <ul style="list-style-type: none"> • 1 specifies LATIN server character data type columns. • 2 specifies UNICODE server character data type columns. • 3 specifies KANJISJIS server character data type columns. • 4 specifies GRAPHIC server character data type columns. • 5 specifies KANJI1 server character data type columns. <p>If the data type of the column is not character, returns a null.</p> |
| IDCol Type | CHARACTER(2) | Yes | <p>Returns a code for the type of identity column.</p> <ul style="list-style-type: none"> • GA specifies Generated Always. • GD specifies Generated by Default. <p>If the column is not an identity column, returns a null.</p> |
| UDT Name | VARCHAR (61) | Yes | Returns the qualified type name for the UDT. |
| Temporal Column This attribute only applies to temporal tables. | CHARACTER(1) | Yes | <p>Returns the temporal qualification, if any, for the column formatted as X(1) with the title Temporal Column.</p> <ul style="list-style-type: none"> • Null specifies that the column is non-temporal. • R specifies that the column is part of a temporal relationship constraint. • T specifies that the column has TRANSACTIONTIME column. • V specifies that the column has VALIDTIME column. <p>If the column is not temporal, returns null. For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table</i></p> |

| Attribute | Data Type | Nullable? | Description |
|---|--------------|-----------|--|
| | | | <i>Support, B035-1186 and Teradata Vantage™ - Temporal Table Support, B035-1182.</i> |
| Current ValidTime Unique This attribute only applies to temporal tables. | CHARACTER(1) | Yes | <p>Reports whether a VALIDTIME column is Current and a UNIQUE constraint or not. If the value is:</p> <ul style="list-style-type: none"> • Null, the column is non-temporal. • N, the column is not VALIDTIME and UNIQUE. • Y, the column is VALIDTIME and UNIQUE. <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support, B035-1186 and Teradata Vantage™ - ANSI Temporal Table Support, B035-1186.</i></p> |
| Sequenced ValidTime Unique This attribute only applies to temporal tables. | CHARACTER(1) | Yes | <p>Reports whether a VALIDTIME column is Sequenced and a UNIQUE constraint or not. If the value is:</p> <ul style="list-style-type: none"> • Null, the column is non-temporal. • N, the column is not Sequenced VALIDTIME UNIQUE. • Y, the column is Sequenced VALIDTIME UNIQUE. <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support, B035-1186 and Teradata Vantage™ - Temporal Table Support, B035-1182.</i></p> |
| Nonsequenced ValidTime Unique This attribute only applies to temporal tables. | CHARACTER(1) | Yes | <p>Reports whether a VALIDTIME column is Nonsequenced and a UNIQUE constraint or not. If the value is:</p> <ul style="list-style-type: none"> • Null, the column is non-temporal. • N, the column is not Nonsequenced VALIDTIME UNIQUE. • Y, the column is Nonsequenced VALIDTIME UNIQUE. <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support, B035-1186 and Teradata Vantage™ - Temporal Table Support, B035-1182.</i></p> |
| Current TransactionTime Unique This attribute only applies to temporal tables. | CHARACTER(1) | Yes | <p>Reports whether a TRANSACTIONTIME column is Current and a UNIQUE constraint or not. If the value is:</p> <ul style="list-style-type: none"> • Null, column is non-temporal. • N, column is not Current TRANSACTIONTIME UNIQUE. • Y, column is Current TRANSACTIONTIME UNIQUE. <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table</i></p> |

| Attribute | Data Type | Nullable? | Description |
|---|---|-----------|--|
| | | | <i>Support, B035-1186 and Teradata Vantage™ - Temporal Table Support, B035-1182.</i> |
| Partitioning Column This attribute only applies to column-partitioned tables and join indexes. | CHARACTER(1) LATIN UPPERCASE NOT CASESPECIFIC NOT NULL | No | Identifies whether the column is the partitioning column. <ul style="list-style-type: none"> • N, column is not a partitioning column. • Y, column is a partitioning column in a partitioning expression. |
| Column Partition Number This attribute applies to column-partitioned tables and join indexes. | BIGINT NOT NULL | No | When a column or column expression is column-partitioned, identifies the number of the column partition to which the column belongs. <ul style="list-style-type: none"> • 0 means the column or column expression is not column-partitioned. • Anything other than 0 means the value describes the number of the column partition to which the column belongs. The format for Column Partition Number is 'ZZZZZZZZZZZZZZZZZZZZ9'. Note: Columns of a column-partitioned table or join index that have the same column partition number belong to the same column partition. |
| Column Partition Format This attribute applies to column-partitioned tables and join indexes. | CHARACTER(2) LATIN UPPERCASE NOT CASESPECIFIC NOT NULL | No | The format for a column partition. <ul style="list-style-type: none"> • CS means system-determined COLUMN format. • CU means user-specified COLUMN format. • NA means not applicable. The attribute describes a column expression or the column is not a member of a column partition. <ul style="list-style-type: none"> • RS means system-determined ROW format. • RU means user-specified ROW format. |
| Column Partition AC This attribute applies to column-partitioned tables and join indexes. | CHARACTER(2) LATIN UPPERCASE NOT CASESPECIFIC NOT NULL | No | Indicates whether a column that is part of a column partition is auto-compressed. <ul style="list-style-type: none"> • AC means auto-compressed. • NA means not applicable. The attribute describes a column expression or the column is not a member of a column partition. <ul style="list-style-type: none"> • NC means not auto-compressed. |
| Derived_UDT This attribute applies to derived Period columns. | CHARACTER(2) LATIN UPPERCASE NOT | No | This attribute applies to derived Period columns. <ul style="list-style-type: none"> • Null means that the column is not a derived Period column or a component of a derived Period column. • PB means derived Period column BEGIN. |

| Attribute | Data Type | Nullable? | Description |
|---------------------------------|--------------------------|-----------|--|
| | CASESPECIFIC NOT NULL | | <ul style="list-style-type: none"> • PE means derived Period column END. • PP means derived Period column. |
| Derived_ UDTFieldID | SMALLINT | Yes | <p>This attribute applies to derived Period columns and either reports the fieldid of the derived UDT column or null.</p> <p>Null means one of two possible things.</p> <ul style="list-style-type: none"> • The column is a derived Period column. • The column is not a component of a derived Period column. |
| Column Dictionary Name | VARCHAR(128) | No | <p>The attributes shown in this section are based upon and supersede the corresponding older attributes, Column Name, Title, UDT Database and UDT Name, while providing additional functionality.</p> <p>The older attributes are retained for compatibility with existing applications.</p> <p>For details, see the topics beginning with Object Name and Title Data in HELP Reports.</p> |
| Column SQL Name | VARCHAR(644) | | |
| Column Name UEscape | VARCHAR(1) | Yes | |
| Dictionary Title | VARCHAR(256) | Yes | |
| SQL Title | VARCHAR(1256) | ? | |
| Title UEscape | VARCHAR(1) | Yes | |
| UDT Database Dictionary Name | VARCHAR(128) | ? | |
| UDT Database SQL Name | VARCHAR(644) | ? | |
| UDT Database Name UEscape | VARCHAR(1) | Yes | |
| UDT Dictionary Name | VARCHAR(128) | No | |
| UDT SQL Name | VARCHAR(644) | | |
| UDT Name UEscape | VARCHAR(1) | Yes | |

HELP CONSTRAINT

HELP CONSTRAINT and Row-Level Security Constraints

The HELP CONSTRAINT statement has nothing to do with the row-level security constraints used by Teradata Row-Level Security. It does not report information for those constraints. Instead, submit a HELP SESSION request and read what the request reports for the Constraint attribute.

HELP CONSTRAINT and Non-Updatable Views

You cannot execute a HELP CONSTRAINT request against a non-updatable view.

Supported Named Constraint Types

There are three main types of named constraints:

- CHECK
- Referential
 - PRIMARY KEY
 - FOREIGN KEY
 - REFERENCES
- UNIQUE

CHECK Constraint Attributes

The following table lists the attributes reported as the result of a HELP CONSTRAINT request for a check constraint.

| Attribute | Data Type | Nullable | Description |
|-----------------|--------------|----------|--|
| Name | VARCHAR (30) | No | The name of the constraint about which HELP information is being reported. |
| Type | VARCHAR(15) | No | The type of constraint. For check constraints, the value is CHECK. |
| Constraint | VARCHAR(255) | No | The text of the constraint clause, |
| Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are derived from and supersede the corresponding older Name attribute, while providing additional functionality. The Name attribute is retained for compatibility with existing applications. For details on the new attributes, see the topics beginning with Object Name and Title Data in HELP Reports . |
| SQL Name | VARCHAR(644) | | |
| UEScape | VARCHAR(1) | Yes | |

Referential Constraint Attributes

The following table lists the attributes reported as the result of a HELP CONSTRAINT request for a referential constraint.

| Attribute | Data Type | Nullable | Description |
|----------------------|--------------|----------|---|
| Name | VARCHAR (30) | No | The name of the constraint about which HELP information is being reported. |
| Type | VARCHAR(15) | No | The type of constraint. For referential constraints, the value is REFERENCE. |
| State | CHAR(12) | No | Identifies any inconsistencies in the parent-child relationship defined in the constraint. If the State Text is: <ul style="list-style-type: none"> • INCONSISTENT, then the constraint is inconsistent. • INVALID, then the constraint is not valid. • UNRESOLVED, then the constraint is Unresolved. • VALID, then the constraint is valid. The following additional values may be returned if Batch or Soft RI are used: <ul style="list-style-type: none"> • InconsisBatch • InvalidBatch • UnresolvBatch • ValidBatch • InconsisSoft • InvalidSoft • UnresolvSoft • ValidSoft |
| Index Id | SMALLINT | No | Internal identifier for the index assigned by the system, which implements unique constraints as indexes. |
| Foreign Key Columns | VARCHAR(512) | Yes | Lists the names of the foreign key columns for the constraint. Note: The system converts each column name, including substitution characters, to its full length. If the constructed list of column names exceeds the maximum length of the Column Names field, the list is truncated after the last comma separator that can fit within the field. |
| Parent Database Name | VARCHAR (30) | Yes | The name of the database that contains the parent table for the constraint. |
| Parent Table Name | VARCHAR (30) | Yes | The name of the table that contains the constraint. |

| Attribute | Data Type | Nullable | Description |
|--|--------------|----------|---|
| Parent Index ID (except if the State is UNRESOLVED) | SMALLINT | Yes | Identifies the parent index for the constraint. |
| Parent Key Columns | VARCHAR(512) | Yes | Lists the names of the parent key columns for the constraint. Note: The system converts each column name, including substitution characters, to its full length. If the constructed list of column names exceeds the maximum length of the Column Names field, the list is truncated after the last comma separator that can fit within the field. |
| Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are derived from and supersede the corresponding older Name, Parent DB, and Parent Table attributes, while providing additional functionality. The older attributes are retained for compatibility with existing applications. For details on the new attributes, see the topics beginning with Object Name and Title Data in HELP Reports . |
| SQL Name | VARCHAR(644) | | |
| UEScape | VARCHAR(1) | Yes | |
| Parent DB Dictionary Name | VARCHAR(128) | No | |
| Parent DB SQL Name | VARCHAR(644) | | |
| Parent DB UEScape | VARCHAR(1) | Yes | |
| Parent Table Dictionary Name | VARCHAR(128) | No | |
| Parent Table SQL Name | VARCHAR(644) | | |
| Parent Table UEScape | VARCHAR(1) | Yes | |

UNIQUE Constraint Attributes

The following table lists the attributes reported as the result of a HELP CONSTRAINT request for a unique constraint. You can also issue a HELP CONSTRAINT against an index, which the system considers as a constraint.

| Attribute | Data Type | Nullable | Description |
|-----------|-------------|----------|--|
| Name | VARCHAR(30) | No | The name of the constraint about which HELP information is being reported. |
| Type | VARCHAR(35) | No | The type of constraint. |

| Attribute | Data Type | Nullable | Description |
|-----------------|--------------|----------|--|
| | | | <p>PRIMARY INDEX specifies that the constraint is a primary index.</p> <p>Other values that include VALIDTIME and TRANSACTIONTIME refer to constraints on temporal tables.</p> <p>See <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182 for information about constraint types for temporal tables.</p> |
| Unique | CHARACTER | No | <p>Reports whether the index or constraint is unique.</p> <ul style="list-style-type: none"> • Yes indicates the constraint or index is unique • No indicates that the constraint or index is not unique |
| Index ID | SMALLINT | No | Internal identifier for the index assigned by the system, which implements unique constraints as indexes. |
| Column Names | VARCHAR(512) | No | <p>Lists the names of columns for the constraint.</p> <p>Note:</p> <p>The system converts each column name, including substitution characters, to its full length. If the constructed list of column names exceeds the maximum length of the Column Names field, the list is truncated after the last comma separator that can fit within the field.</p> |
| Dictionary Name | VARCHAR(128) | No | <p>The attributes shown in this section are derived from and supersede the corresponding older Name attribute, while providing additional functionality.</p> <p>The Name attribute is retained for compatibility with existing applications.</p> |
| SQL Name | VARCHAR(644) | | |
| UEScape | VARCHAR(1) | Yes | <p>For details on the new attributes, see Object Name and Title Data in HELP Reports.</p> |

HELP DATABASE

HELP DATABASE Attributes

The following table lists the attributes reported as the result of a HELP DATABASE request. For each object in the database (not just tables, views, and macros), the system returns the attributes listed in the table below.

Note:

HELP does not differentiate between user and database objects, so HELP USER and HELP DATABASE commands that specify the same object name return the same information.

| Attribute | Data Type | Nullable | Description |
|-----------------------|---------------|----------|--|
| Table/View/Macro name | CHARACTER(30) | No | The name of an of an object in the database specified in the HELP request. |
| Kind | CHARACTER | No | Identifies the object type for the object identified by Table/View/Macro name. See TVM Kind Codes . |
| Comment | VARCHAR(255) | Yes | The contents of the optional comment field, if a comment was created. for the database. |
| Protection | CHARACTER | No | Indicates whether the database uses fallback protection. <ul style="list-style-type: none"> • F is Fallback. • N is None. |
| Creator Name | CHARACTER(30) | No | The name of the creator for the database. |
| Commit Option | CHARACTER | Yes | Indicates whether rows are preserved upon COMMIT of the transaction. <ul style="list-style-type: none"> • D represents Delete rows on COMMIT. • P represents Preserve rows on COMMIT. <p>Note: This attribute applies to temporary tables only.</p> |
| Transaction Log | CHARACTER | No | Indicates whether a transaction log is kept. <ul style="list-style-type: none"> • N means no log is kept. • Y means transactions are logged. <p>Note: This attribute applies to temporary tables only.</p> |
| Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are derived from and supersede the corresponding older Table/View |

| Attribute | Data Type | Nullable | Description |
|-------------------------|--------------|----------|--|
| SQL Name | VARCHAR(644) | | <p>/Macro Name and Creator Name attributes, while providing additional functionality.</p> <p>The system returns these attributes for each object in the specified database.</p> <p>The older attributes are retained for compatibility with legacy applications.</p> <p>For details on the new attributes, see the topics beginning with Object Name and Title Data in HELP Reports.</p> |
| UEScape | VARCHAR(1) | Yes | |
| Creator Dictionary Name | VARCHAR(128) | No | |
| Creator SQL Name | VARCHAR(644) | | |
| Creator UEScape | VARCHAR(1) | Yes | |

HELP ERROR TABLE

System-Defined Attribute Characteristics for the Error Table-Specific Columns

For more information about these columns, see [System-Defined Attributes for Error Table-Specific Columns](#).

The system-defined and generated error table-specific columns have the following characteristics:

| Column Name | Data Type | Nullable? |
|-----------------|---------------|-----------|
| ETC_DBQL_QID | DECIMAL(18,0) | N0 |
| ETC_DMLType | CHARACTER(1) | Yes |
| ETC_ErrorCode | INTEGER | N0 |
| ETC_ErrSeq | INTEGER | N0 |
| ETC_IndexNumber | SMALLINT | Yes |
| ETC_IdxErrType | CHARACTER(1) | Yes |
| ETC_RowId | BYTE(10) | Yes |
| ETC_TableId | BYTE(6) | Yes |
| ETC_FieldId | SMALLINT | Yes |
| ETC_RITableId | BYTE(6) | Yes |
| ETC_RIFieldId | SMALLINT | Yes |
| ETC_TimeStamp | TIMESTAMP(2) | N0 |
| ETC_Blob | BLOB | Yes |

Information Returned By The Different Syntax Forms Is Identical

The HELP ERROR TABLE information and attributes reported by either syntax are identical to one another and to that returned by HELP TABLE, only for an error table rather than for a base data table.

See [CREATE ERROR TABLE](#) for more information about error tables.

See [HELP MACRO](#) for information about the information and attributes returned by the various syntaxes for the HELP ERROR TABLE statement. Also see [CREATE TABLE](#).

HELP FUNCTION

User-Defined Function Attributes

The following attributes and format information are returned for each parameter in a user-defined function (UDF). The report returned for external functions and SQL functions is the same.

Note:

For a table function defined with a dynamic row result specification, HELP FUNCTION returns only input parameters because the maximum number of output parameters for the function cannot be known before it runs. Because of this, the names and data types for the output parameters also cannot be defined until run time.

| Attribute | Data Type | Nullable? | Description |
|----------------------|----------------|-----------|--|
| Parameter Name | VARCHAR(30) | No | The name of a parameter defined for the UDF specified in the HELP Function request. A parameter name and set of attributes is returned for each parameter in the specified UDF. Note: The word <i>RETURN0</i> is a system-generated return parameter name. |
| Type | CHARACTER(2) | No | The data for the reporting parameter. See Data Type Codes for a list of the data type codes and their meanings. |
| Comment | CHARACTER(255) | Yes | The contents of the optional comment field for the function, if a comment was created. |
| Nullable | CHARACTER(1) | No | Specifies whether the parameter is nullable. • N specifies NOT NULL. • Y specifies nullable. |
| Format | CHARACTER(30) | Yes | Display format for the data type. For a UDT, the displayed format is the format associated with the external type of the UDT. For data type display formats, see <i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143. |
| Max Length | INTEGER | No | The column format is -(10)9. When the parameter data type is a UDT, this column is 0. |
| Decimal Total Digits | SMALLINT | Yes | If type is DECIMAL. |

| Attribute | Data Type | Nullable? | Description |
|---------------------------|--------------|-----------|---|
| Decimal Fractional Digits | SMALLINT | Yes | If type is DECIMAL. |
| Table/View? | CHARACTER(1) | No | The function type: <ul style="list-style-type: none"> • A specifies Aggregate function. • B specifies a combined Aggregate and Statistical function. • E specifies external procedure. • F specifies Scalar function. • M specifies Function used as a method. • R specifies Table function. • S specifies Statistical function. |
| Character Type | SMALLINT | Yes | Returns a code identifying the character data type for the parameter. <ul style="list-style-type: none"> • 1 specifies LATIN server character data type columns. • 2 specifies UNICODE server character data type columns. • 3 specifies KANJISJIS server character data type columns. • 4 specifies GRAPHIC server character data type columns. • 5 specifies KANJI1 server character data type columns. If the data type of the column is not character, returns null. |
| Parameter Type | CHARACTER(2) | No | The type of parameter specified by the Parameter Name field. <ul style="list-style-type: none"> • C specifies a Return column of a function table. • E specifies an External OUT parameter. • I specifies an IN parameter. • O specifies an OUT parameter. • UT specifies UDT parameter |
| UDT Name | VARCHAR(61) | Yes | Returns the unqualified name of the UDT for the Parameter Name. If a UDT does not exist, the parameter is NULL. |
| Parameter Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are derived from and supersede the corresponding older Parameter Name and UDT Name attributes, while providing additional functionality. The older attributes are retained for compatibility with legacy applications. |
| Parameter SQL Name | VARCHAR(644) | | |

| Attribute | Data Type | Nullable? | Description |
|---------------------------|--------------|-----------|---|
| Parameter Name UEScape | VARCHAR(1) | Yes | For details on the new attributes, see the topics beginning with Object Name and Title Data in HELP Reports . |
| UDT Dictionary Name | VARCHAR(128) | Yes | |
| UDT SQL Name | VARCHAR(644) | Yes | |
| UDT Name UEScape | VARCHAR(1) | Yes | |

HELP HASH INDEX

Hash Index Attributes

The attributes and display format for a hash index are identical to that of the HELP TABLE statement.

See [HELP MACRO](#) for information about the attributes and display format for HELP HASH INDEX.

HELP INDEX Usage Restriction

HELP HASH INDEX is considered obsolete. Although HELP HASH INDEX is still supported for the current database release, Teradata recommends that you instead use the DBC.IndicesV[X] views. For information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

The system does not return the Dictionary Name, SQL Name and Uescape attributes, which are returned for other HELP statements.

HELP INDEX

HELP INDEX Usage Restriction

HELP INDEX is considered obsolete. Although HELP INDEX is still supported for the current database release, Teradata recommends that you instead use the DBC.IndicesV[X] views. For information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

The system does not return the Dictionary Name, SQL Name and Uescape attributes, which are returned for other HELP statements.

HELP INDEX and Non-Updatable Views

You cannot execute a HELP INDEX request against a non-updatable view.

HELP INDEX and Partitioned Primary Indexes

The report for a table with a partitioned primary index is identical to the report produced for a table with a nonpartitioned primary index. No information about the partitioning expression or partitioning columns is provided. SHOW TABLE reports the partitioning expression for the primary index (see [SHOW object](#)).

HELP INDEX and Unindexed Tables

If you submit a HELP INDEX request for an unindexed table, the system returns a message explaining that there is no help information to be returned.

For example, suppose you define the following table.

```
CREATE TABLE t1 (
  a INTEGER
  b INTEGER)
NO PRIMARY INDEX;
```

You then submit the following HELP INDEX request on *t1*.

```
HELP INDEX t1;
```

The system returns the following message.

```
*** Empty HELP information returned.
```


HELP INDEX and NoPI Tables, Column-Partitioned Tables, and Column-Partitioned Join Indexes

A table or join index with column partitioning cannot have a primary index, so Vantage does not return a row for a primary index for a NoPI table, a column-partitioned table, or a column-partitioned join index.

Because of this, a HELP INDEX request cannot be used to determine whether or not a table or join index is column partitioned.

HELP INDEX and Hash, Join and Covering Indexes

HELP INDEX does *not* report information about hash or join indexes.

HELP INDEX *does* report on the indexes defined on hash and join indexes, as explained in the following table:

| FOR this type of index ... | Information is reported on the following index characteristics ... |
|----------------------------|--|
| Hash | <ul style="list-style-type: none"> Primary index defined by default based on the primary index of its underlying table. Primary index defined explicitly by means of the BY and ORDER BY clauses of the CREATE HASH INDEX statement. <p>See Specifying the Primary Index for a Hash Index for information about these primary indexes.</p> |
| Join | <p>Primary index defined for the join index.</p> <p>Any secondary indexes defined on the join index.</p> |

HELP INDEX does not report the ordering columns for a covering index or value-ordered join index. SHOW TABLE reports these columns (see [SHOW object](#)).

HELP INDEX Attributes

The following attributes are reported when you submit a HELP INDEX request.

| Attribute | Data Type | Description |
|-----------------------|--------------|--|
| Unique? | CHARACTER(1) | <p>Defines whether the index is unique or nonunique.</p> <ul style="list-style-type: none"> N specifies a nonunique index. Y specifies a unique index. |
| Primary or Secondary? | CHARACTER(1) | <p>Defines whether the index is primary or secondary.</p> <ul style="list-style-type: none"> P specifies a Primary index. S specifies a Secondary index. <p>This code does not discriminate between USIs and NUSIs: both types of secondary index are reported as S.</p> |

| Attribute | Data Type | Description |
|-------------------------|-------------------------------------|--|
| Column Names | VARCHAR(512) | <p>Lists the names of the columns defined for the index.</p> <p>Note:</p> <p>The system converts each column name, including extended-length names and substitution characters, to its full length. If the constructed list of column names exceeds the maximum length of the Column Names field, the result is truncated after the last comma separator that can fit within the field. If the field ends in a comma, the list of names is truncated.</p> |
| Index ID | INTEGER | <p>System-defined.</p> <p>The primary index for a table is always index number 1. Secondary indexes are numbered in increments of 4 beginning with index number 4, which is assigned to the first secondary index defined for a table.</p> |
| Approximate Count | FLOAT | Defines the approximate number of rows in the index subtable. |
| Index Name | CHARACTER(30) | <p>Lists the index name, if there is one. This column is set to NULL if the index is unnamed.</p> <p>See “Column Names” earlier in this table for information about the CHARACTER SET attribute the Vantage assigns to INDEX NAME, depending on the language support for your system.</p> |
| Ordered or Partitioned? | CHARACTER(1) CHARACTER SET LATIN | <p>Defines whether the index is ordered or partitioned.</p> <ul style="list-style-type: none"> • H specifies a hash-ordered index. • P specifies a row-partitioned index. • V specifies a value-ordered index. |
| CDT Index Type | CHARACTER(1) CHARACTER SET LATIN | <p>Defines the type of complex data type NUSI this is.</p> <ul style="list-style-type: none"> • G specifies a Hilbert R-tree geospatial NUSI index. • N specifies a non-Hilbert R-tree geospatial index. |

HELP JOIN INDEX

Join Index Attributes

The attributes and display format for a join index are identical to that of the HELP TABLE statement. In fact, the report produced from a HELP TABLE statement performed against a join index is identical to that produced by a HELP JOIN INDEX statement performed against the same join index.

See [HELP TABLE](#) for information about the attributes and display format for HELP JOIN INDEX.

HELP MACRO

Macro Attributes

The columns or parameters are listed in the order in which they were defined.

The attributes are returned as shown in the following table.

| Attribute | Data Type | Nullable? | Description |
|----------------|---------------|-----------|--|
| Parameter Name | VARCHAR(30) | No | <p>The name of a parameter in the macro.</p> <p>The system returns the set of attributes shown in this attribute table for each column or parameter.</p> <p>Note:</p> <p>The following characteristics also apply:</p> <p>Host Data Characteristics: Interpret as the equivalent of CF, or CHARACTER(n) data type.</p> <p>Host Data Characteristics: When DATEFORM=ANSIDATE, this column should be described externally as CHARACTER(10).</p> <p>Format Characteristics: With an explicit format, DATE.</p> <p>With an implicit format created in ANSIDATE mode, 'YYYY/MM/DD'.</p> |
| Data Type | CHARACTER(2) | Yes | <p>The data type for the Parameter Name.</p> <p>See Data Type Codes for a list of the data type codes and their meanings</p> <p>The server character data types LATIN, UNICODE, GRAPHIC, KANJISJIS, and KANJI1 are distinguished with respect to CF and CV by their respective value for the Character Type column.</p> |
| Comment | VARCHAR (255) | Yes | The contents of the optional comment field for the macro, if a comment was created. |
| Nullable | CHARACTER | Yes | <p>Indicates whether the macro parameter specified in the Parameter Name field is nullable.</p> <ul style="list-style-type: none"> • N specifies nullable. • Y specifies not nullable. |
| Format | CHARACTER(30) | Yes | <p>Display format for the data type.</p> <p>For a UDT, the displayed format is the format associated with the external type of the UDT.</p> <p>For data type display formats, see <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143.</p> |
| Title | VARCHAR (60) | Yes | The title for Parameter Name, if used. |
| Max Length | INTEGER | Yes | Return the length in bytes for the macro parameters specified in Parameter Name. |

| Attribute | Data Type | Nullable? | Description |
|---------------------------|-----------|-----------|--|
| | | | <p>MaxLength is the length of the <i>external</i> CharFix representation of the parameter, defined as CHARACTER(MaxLength) in USING clauses and any other place this needs to be specified.</p> <p>Note that while MaxLength is usually expressed as the internal size, presentation of the external size is more appropriate for these data types.</p> <p>For the DATE type MaxLength is 4, representing the internal storage size.</p> <p>For CHARACTER data, Max Length is the length of the parameter declaration in bytes.</p> <p>For HELP MACRO, requests return the length of the macro parameter.</p> <p>For UDTs, Max Length is the maximum length of the external type associated with the UDT formatted as -(10)9.</p> <p>See the external representation for each individual DateTime and Interval type in <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143 to determine the MaxLength values.</p> <p>You can use the StatementInfo parcel to return the description of items within the Data Field of the corresponding indicator mode Record parcels.</p> <p>For example, the FastLoad .TABLE command can be executed in Indicator Mode to get the additional information of data type and length of all attributes. That information can then be used to fetch the data correctly in any database release.</p> |
| Decimal Total Digits | SMALLINT | Yes | <p>Indicates the total number of digits for decimal data.</p> <p>For INTERVAL types, returns a value from 1–4 representing the number of digits in the leading field of the interval (number of non-fractional digits for INTERVAL SECOND).</p> <p>This column is null for other TIME and TIMESTAMP data types.</p> |
| Decimal Fractional Digits | SMALLINT | Yes | <p>Indicates the decimal precision of the parameter in Parameter Name.</p> <p>If the Type is TIME or TIMESTAMP types, or any INTERVAL type with a SECOND field, this column returns a value of 0–6, indicating the fractional precision of seconds.</p> <p>For INTERVAL types without a SECOND field, the column returns null.</p> <p>Null for other data types.</p> |
| Range Low | FLOAT | Yes | <p>Always null. BETWEEN clause is included in column constraint.</p> |

| Attribute | Data Type | Nullable? | Description |
|-----------------|---------------|-----------|---|
| Range High | FLOAT | Yes | Always null. BETWEEN clause is included in column constraint. |
| Uppercase | CHARACTER | Yes | Applies if the type is CHARACTER or VARCHAR. <ul style="list-style-type: none"> • B specifies that column attributes for both UPPERCASE and CASESPECIFIC are defined. • C specifies that the column attributes for CASESPECIFIC is defined. • N specifies that column attributes for neither UPPERCASE nor CASESPECIFIC are defined. • U specifies that the column attributes for UPPERCASE is defined. |
| Table/View? | CHARACTER | No | The object type. See TVM Kind Codes . HELP MACRO requests always return a code of M for macro. |
| Default Value | VARCHAR (255) | Yes | Returns the default value (as text) for the parameter. The value is null if there is no default value. |
| Character Type | SMALLINT | Yes | Returns a code identifying the character data type for the parameter shown in Parameter Name. <ul style="list-style-type: none"> • 1 specifies LATIN server character data type columns. • 2 specifies UNICODE server character data type columns. • 3 specifies KANJISJIS server character data type columns. • 4 specifies GRAPHIC server character data type columns. • 5 specifies KANJI1 server character data type columns. If the data type of the column is not character, returns null. |
| IDCol Type | CHARACTER(2) | Yes | Returns a code for the type of identity column. <ul style="list-style-type: none"> • GA specifies Generated Always. • GD specifies Generated by Default. |
| UDT Name | VARCHAR(61) | Yes | Returns the unqualified name of the UDT. If a UDT does not exist, the parameter is NULL. |
| Temporal Column | CHARACTER(2) | Yes | Reports whether a column is temporal or not and if it is part of a Temporal Relationship constraint. <ul style="list-style-type: none"> • Null specifies that the column is non-temporal. • R specifies that the column is part of a temporal relationship constraint. |

| Attribute | Data Type | Nullable? | Description |
|---------------------------------|---------------|-----------|--|
| | | | <ul style="list-style-type: none"> • T specifies that the column has TRANSACTIONTIME column. • V specifies that the column has VALIDTIME column. <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Current ValidTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Current VALIDTIME column or not.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Sequenced ValidTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Sequenced VALIDTIME column or not.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| NonSequenced ValidTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Nonsequenced VALIDTIME column or not.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Current TransactionTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Current TRANSACTIONTIME column or not.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Parameter Dictionary Name | VARCHAR(128) | No | <p>The attributes shown in this section are derived from and supersede the corresponding older Parameter Name, Title, and UDT Name attributes, while providing additional functionality.</p> <p>These attributes are returned for each parameter in the specified method.</p> <p>The older attributes are retained for compatibility with legacy applications.</p> <p>For details on the new attributes, see the topics beginning with Object Name and Title Data in HELP Reports.</p> |
| Parameter SQL Name | VARCHAR(644) | No | |
| Parameter Name Uescape | VARCHAR(1) | Yes | |
| Dictionary Title | VARCHAR(256) | Yes | |
| SQL Title | VARCHAR(1256) | Yes | |
| Title Uescape | VARCHAR(1) | Yes | |
| UDT Dictionary Name | VARCHAR(128) | Yes | |

| Attribute | Data Type | Nullable? | Description |
|---------------------|--------------|-----------|-------------|
| UDT SQL Name | VARCHAR(644) | Yes | |
| UDT Name Uescape | VARCHAR(1) | Yes | |

HELP METHOD

HELP METHOD Only Valid For User-Defined Methods

You can only submit a HELP METHOD request against a user-defined method. This is why you cannot specify MUTATOR or OBSERVER as method types, because mutator and observer methods are both system-generated.

If you attempt to perform the statement by specifying the name of a system-generated method, the request aborts and the system returns an error to the requestor.

SELF Parameter

The first parameter listed in a HELP METHOD report is the first parameter, named SELF, that the system generates internally by default.

HELP METHOD Attributes

The following table lists the attributes reported by the HELP METHOD statement:

| Attribute | Data Type | Nullable? | Description |
|----------------|----------------|-----------|--|
| Parameter Name | VARCHAR(30) | No | The name of a parameter for the method specified in the HELP METHOD request. The system returns the set of attributes shown in this table for each parameter. |
| Type | CHARACTER(2) | Yes | The data type for the Parameter Name. See Data Type Codes for a list of the data type codes and their meanings. When DATEFORM=ANSIDATE, this column should be described externally as CHARACTER(10). With an explicit format, DATE. With an implicit format created in INTEGERDATE mode, 'YY/MM/DD'. With an implicit format created in ANSIDATE mode, 'YYYY/MM/DD'. Interpret as the equivalent of CF, or CHARACTER(n) data type. |
| Comment | CHARACTER(255) | Yes | The contents of the optional comment field for the method, if a comment was created. |
| Nullable | CHARACTER(1) | Yes | Defines whether the parameter in Parameter Name can be null. |
| Format | CHARACTER(30) | Yes | Returns null if the parameter data type is UDT. This is because the system passes UDTs to routines as handles, not directly as UDT values. |

| Attribute | Data Type | Nullable? | Description |
|---------------------------|--------------------------------|-----------|--|
| | | | The character set for FORMAT is <i>always</i> Unicode regardless of the language support mode for your system and the session character set. <i>See Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147 for details. |
| Max Length | INTEGER Formatted as -(10)9 | Yes | Returns the length in bytes of the parameter specified by Parameter Name. MaxLength is the length of the <i>external</i> CharFix representation of the column, defined as CHARACTER(MaxLength) in USING clauses and any other place this needs to be specified. Note that while MaxLength is usually expressed as the internal size, presentation of the external size is more appropriate for these data types. MaxLength is 4, representing the internal storage size. See the external representation for each individual DateTime and Interval type in <i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143 to determine the MaxLength values. |
| Decimal Total Digits | SMALLINT Formatted as -(5)9 | Yes | If the parameter is decimal. For INTERVAL types, returns a value from 1–4 representing the number of digits in the leading field of the interval (number of non-fractional digits for INTERVAL SECOND). This column is null for other TIME and TIMESTAMP data types. |
| Decimal Fractional Digits | SMALLINT Formatted as -(5)9 | Yes | If type is DECIMAL. For TIME and TIMESTAMP types and all INTERVAL types with a SECOND field, this column returns a value of 0–6, indicating the fractional precision of seconds. For INTERVAL types without a SECOND field, the column returns null. The server character data types LATIN, UNICODE, GRAPHIC, KANJISJIS, and KANJI1 are distinguished with respect to CF and CV by their respective value for the Character Type column. |
| Table/View? | CHARACTER(1) | No | Object type. The code M defines the object as a method. For other object codes, see TVM Kind Codes . |
| Char Type | SMALLINT Formatted as -(5)9 | Yes | Returns a code identifying the character data type for the parameter shown in Parameter Name. <ul style="list-style-type: none"> • 1 specifies LATIN server character data type columns. • 2 specifies UNICODE server character data type columns. • 3 specifies KANJISJIS server character data type columns. |

| Attribute | Data Type | Nullable? | Description |
|---------------------------|--------------|-----------|--|
| | | | <ul style="list-style-type: none"> • 4 specifies GRAPHIC server character data type columns. • 5 specifies KANJI1 server character data type columns. <p>If the data type of the column is not character, returns null.</p> |
| Parameter Type | CHARACTER(2) | No | <p>Defines the parameter type.</p> <ul style="list-style-type: none"> • C specifies a Return column of a function table. • E specifies an External OUT parameter. • I specifies an IN parameter. • O specifies an OUT parameter. • UT specifies UDT parameter <p>For a HELP METHOD request, the parameter type is always UDT.</p> |
| UDT Name | VARCHAR(61) | Yes | <p>Returns UDT name if the parameter is a UDT.</p> <p>If the parameter is not a UDT, returns null.</p> |
| Parameter Dictionary Name | VARCHAR(128) | No | <p>The attributes shown in this section are derived from and supersede the corresponding older Parameter Name and UDT Name attributes, while providing additional functionality.</p> <p>These attributes are returned for each parameter in the specified method.</p> <p>The older attributes are retained for compatibility with legacy applications.</p> <p>For details on the new attributes, see the topics beginning with Object Name and Title Data in HELP Reports.</p> |
| Parameter SQL Name | VARCHAR(644) | No | |
| Parameter Name UEscape | VARCHAR(1) | Yes | |
| UDT Dictionary Name | VARCHAR(128) | Yes | |
| UDT SQL Name | VARCHAR(644) | Yes | |
| UDT Name UEscape | VARCHAR(1) | Yes | |

HELP PROCEDURE

Procedure Parameter Attributes

The following attribute and format information is returned by HELP PROCEDURE for each parameter in the specified procedure.

If the specified procedure has no parameters, the system returns the “Empty help” message.

| Attribute | Data Type | Nullable? | Description |
|---------------------------|--------------------------------|-----------|---|
| Parameter Name | VARCHAR(30) | No | The name of a parameter in the specified procedure. |
| Data Type | CHARACTER(2) | Yes | The data type for the parameter. Always null for UDTs. See Data Type Codes for a list of the data type codes and their meanings. |
| Comment | CHARACTER(255) | Yes | The contents of the optional comment field for the procedure, if a comment was created. |
| Nullable | CHARACTER | No | Indicates whether the parameter can be NULL. • N specifies not null. • Y specifies can be null. |
| Format | CHARACTER(30) | Yes | Display format for the data type. For a UDT, the displayed format is the format associated with the external type of the UDT. For data type display formats, see <i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143. |
| Title | VARCHAR(60) | Yes | The title (if used) for the Parameter Name |
| Max Length | INTEGER | Yes | Default output format is -(10)9. Returns null for a UDT. |
| Decimal Total Digits | SMALLINT Formatted as -(5)9 | Yes | If type is DECIMAL. |
| Decimal Fractional Digits | SMALLINT Formatted as -(5)9 | Yes | If type is DECIMAL. |
| Uppercase | CHARACTER | Yes | Returned if type is CHARACTER or VARCHAR. • B specifies that column attributes for both UPPERCASE and CASESPECIFIC are defined. • C specifies that the column attributes for CASESPECIFIC is defined. • N specifies that column attributes for neither UPPERCASE nor CASESPECIFIC are defined. |

| Attribute | Data Type | Nullable? | Description |
|---------------------------|----------------|-----------|---|
| | | | <ul style="list-style-type: none"> • U specifies that the column attributes for UPPERCASE is defined. |
| TVM Kind | CHARACTER | No | Indicates the type of procedure. <ul style="list-style-type: none"> • E specifies External procedure. • P specifies SQL procedure. |
| Default Value | CHARACTER(255) | Yes | Returns the default value (as text) for the parameter. The value is null if there is no default value. |
| Character Type | SMALLINT | Yes | Returns a code specifying the character data type for each Parameter Name. <ul style="list-style-type: none"> • 1 specifies LATIN server character data type columns. • 2 specifies UNICODE server character data type columns. • 3 specifies KANJISJIS server character data type columns. • 4 specifies GRAPHIC server character data type columns. • 5 specifies KANJI1 server character data type columns. |
| Parameter Type | CHARACTER(2) | No | Indicates the type of parameter for the Parameter Name. <ul style="list-style-type: none"> • B specifies an INOUT parameter. • E specifies an External OUT parameter. • I specifies an IN parameter. • O specifies an OUT parameter. • UT specifies UDT parameter |
| UDT Name | VARCHAR(61) | Yes | Returns the qualified name of the UDT formatted X(61). |
| Parameter Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are derived from and supersede the corresponding older Parameter Name, Title, and UDT Name attributes, while providing additional functionality. These attributes are returned for each parameter in the specified procedure. The older attributes are retained for compatibility with legacy applications. For details on the new attributes, see the topics beginning with Object Name and Title Data in HELP Reports . |
| Parameter SQL Name | VARCHAR(644) | No | |
| Parameter Name UEscape | VARCHAR(1) | Yes | |
| Dictionary Title | VARCHAR(256) | Yes | |
| SQL Title | VARCHAR(1256) | Yes | |
| Title UEscape | VARCHAR(1) | Yes | |

| Attribute | Data Type | Nullable? | Description |
|---------------------|--------------|-----------|-------------|
| UDT Dictionary Name | VARCHAR(128) | No | |
| UDT SQL Name | VARCHAR(644) | No | |
| UDT Name UEscape | VARCHAR(1) | Yes | |

Procedure Attributes

| Attribute | Description |
|-----------------------------|---|
| Transaction Semantics | <ul style="list-style-type: none"> ANSI specifies ANSI transaction semantics. TERADATA specifies Teradata transaction semantics. |
| Platform | LINUX specifies the Linux operating system |
| Character Set | <ul style="list-style-type: none"> ASCII specifies the ASCII character set. EBCDIC specifies the EBCDIC character set. KANJISJIS specifies the Kanji Shift Japanese Industrial Standard Code (DOS Kanji) character set. KANJIEBCDIC specifies the Kanji EBCDIC character set. |
| Default Character Data Type | <ul style="list-style-type: none"> GRAPHIC specifies the IBM DB2 Graphic server character data type. KANJI1 specifies the non-canonical Kanji server character data type. KANJISJIS specifies the canonical Kanji: Shift Japanese Industrial Standard server character data type. LATIN specifies the Teradata Latin server character data type. UNICODE specifies the Unicode server character data type. |
| Collation | <p>The following collations are supported. See CREATE USER or <i>Teradata Vantage™ - Advanced SQL Engine International Character Set Support</i>, B035-1125 for more information about the various collations supported by Vantage.</p> <ul style="list-style-type: none"> ASCII CHARSET_COLL EBCDIC JIS_COLL MULTINATIONAL |
| Stored Procedure Text | <ul style="list-style-type: none"> N specifies that statements defining the procedure are not stored. Y specifies that statements defining the procedure are stored in the procedure object. |
| Version Number | <ul style="list-style-type: none"> 00 specifies that the procedure was created in V2R4.0. 01 specifies that the procedure was created in V2R4.1. 02 specifies that the procedure was created in V2R4.2. |

| Attribute | Description |
|--------------------------|---|
| | <ul style="list-style-type: none"> • 03 specifies that the procedure was created in V2R5.0. • 04 specifies that the procedure was created in V2R6.0. • 05 specifies that the procedure was created in V2R6.2. • 06 specifies that the procedure was created in V2R7.0. • 07 specifies that the procedure was created in Teradata Database 13.0. • 08 specifies that the procedure was created in Teradata Database 13.10. • 09 specifies that the procedure was created in Teradata Database 14.0. • 10 specifies that the procedure was created in Teradata Database 14.10. • 11 specifies that the procedure was created in Teradata Database 15.0. • 12 specifies that the procedure was created in Teradata Database 15.10. |
| Default Database | The default database name for the procedure is the creator's default database. |
| Warning Option | <ul style="list-style-type: none"> • N specifies that all compilation warnings are suppressed. • Y is the default. <p>It specifies that all compilation warnings generated during procedure compilation or recompilation are returned to the user.</p> |
| Creation Timezone | The time zone offset when the procedure was created or last altered. |
| Creation Timezone String | <p>The time zone string used to set the time zone offset for the procedure when it was created or last altered.</p> <p>If the procedure was created or altered using a GMT time zone specification, nothing is returned.</p> |

HELP SESSION

Session Attributes

The following table describes the attributes reported by a HELP SESSION request.

| Attribute | Data Type | Nullable | Description |
|------------------|-------------|----------|--|
| User Name | VARCHAR(30) | No | Reports the name of the owner of the session. |
| Account Name | VARCHAR(30) | No | Reports the account name currently in effect for the session. See SET SESSION in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144. |
| Logon Date | CHAR(8) | No | Reports the date of logon to the database, in the format YY/MM/DD. |
| Logon Time | CHAR(8) | No | Reports the time of logon to the database, in the format HH:MI:SS. |
| Current Database | VARCHAR(30) | No | Reports the database currently defined as the default for the session (see DATABASE and CREATE USER). |
| Collation | VARCHAR(15) | No | <p>Reports the sort sequence defined for the session. See CREATE USER and SET SESSION in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</p> <ul style="list-style-type: none"> • ASCII specifies the collation order is as it would appear if converted for an ASCII session, and a binary sort performed. • CHARSET_COLL specifies the collation order is as it would appear if converted to the current client character set and then sorted in binary order. • EBCDIC specifies the collation order is as it would appear if converted for an EBCDIC session and a binary sort performed. • JIS_COLL specifies the collation order is Japanese Industrial Standards (JIS) in this order. <ul style="list-style-type: none"> JIS X 0201-defined characters in standard order. JIS X 0208-defined characters in standard order. JIS X 0212-defined characters in standard order. KanjiEBCDIC-defined characters <i>not</i> defined in JIS X 0201, JIS X 0208, or JIS X 0212 in standard order. All remaining characters in Unicode standard order. |

| Attribute | Data Type | Nullable | Description |
|------------------------|-------------|----------|---|
| | | | <ul style="list-style-type: none"> MULTINATIONAL specifies the collation order is two-level based on the Unicode collation standard. <p>For backward compatibility, the following are true.</p> <p>MULTINATIONAL collation of KANJI1 data is single level.</p> <p>Single byte character collation is redefineable by the system administrator.</p> |
| Character Set | VARCHAR(30) | No | Reports the character set currently defined for the session. |
| Transaction Semantics | VARCHAR(8) | No | <p>Reports the whether the session is in ANSI or Teradata session mode.</p> <ul style="list-style-type: none"> ANSI specifies that the session is in ANSI mode. Teradata specifies that the session is in Teradata mode. |
| Current Dateform | VARCHAR(11) | No | <p>Reports the current mode for export and import of DATE values.</p> <p>For information about date formats, see <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143.</p> <ul style="list-style-type: none"> IntegerDate specifies that the session uses four-byte integer values. ANSIDate specifies that the session uses CHARACTER(10) ANSI data format. |
| Session Time Zone | VARCHAR(6) | No | <p>Reports the current time zone value in use by the session, displayed in INTERVAL HOUR TO MINUTE format.</p> <p>For example, -08:00. This is an offset from UTC.</p> |
| Default Character Type | VARCHAR(30) | No | Reports the default character type for the user logged onto the current session. |
| Export Latin | CHAR(1) | No | <p>Reports the export width value for each character data type for the session (see <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143).</p> <p>Note:</p> <p>No export width is reported for Kanji1 character data type because the export width for such data is always one.</p> |
| Export Unicode | CHAR(1) | No | |
| Export Unicode Adjust | CHAR(1) | No | |
| Export KanjiSJIS | CHAR(1) | No | |
| Export Graphic | CHAR(1) | No | |
| Default Date Format | VARCHAR(30) | No | Reports the setting of the default date output format as set in the Specification for Data Formatting (SDF) file by the database administrator. This custom date format serves as the default output format for dates when DATEFORM is set to IntegerDate. |

| Attribute | Data Type | Nullable | Description |
|--------------------------|-------------|----------|--|
| | | | <p>See “User Name” earlier in this table for information about the data type and system-defined character sets for DEFAULT DATE FORMAT.</p> <p>For information about date formats, see <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143.</p> <p>For information about changing the default date output format in the SDF and the definition of date formatting characters, see <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143.</p> |
| Radix Separator | CHAR(1) | No | <p>The character that separates the integer and fractional part of monetary numeric data.</p> <p>The system copies the value into the output string of numeric data whenever a D appears in the corresponding output FORMAT phrase.</p> |
| Group Separator | CHAR(1) | No | <p>The character that separates groups of digits in the integer part of monetary strings.</p> <p>The value of Currency Grouping Rule determines when the system copies the value of Currency Group Separator into the output string of numeric monetary data.</p> |
| Grouping Rule | VARCHAR(30) | No | <p>The number of digits to group together before inserting the value of Currency Group Separator in the output of monetary numeric data.</p> <p>The grouping rule applies to the output of numeric data that uses a FORMAT phrase containing a G formatting character.</p> |
| Currency Radix Separator | CHAR(1) | No | <p>The character that separates the integer and fractional part of monetary numeric data.</p> <p>The system returns a value if the format string contains the format element for the currency symbol or name.</p> |
| Currency Group Separator | CHAR(1) | No | <p>The character that separates groups of digits in the integer part of monetary strings.</p> <p>The system returns a value if the format string contains the format element for the currency symbol or name.</p> |
| Currency Grouping Rule | VARCHAR(30) | No | <p>The number of digits to group together before inserting the value of Currency Group Separator in the output of monetary numeric data.</p> <p>The system returns a value if the format string contains the format element for the currency symbol or name.</p> |
| Currency Name | VARCHAR(30) | No | <p>The local currency as a completely spelled-out currency name. The system copies the value into the output string of monetary numeric data</p> |

| Attribute | Data Type | Nullable | Description |
|--------------------------|-------------|----------|---|
| | | | whenever an N appears in the corresponding output FORMAT phrase. |
| Currency | VARCHAR(30) | No | The local currency symbol. The system copies the value into the output string of monetary numeric data whenever an L appears in the corresponding output FORMAT phrase. |
| ISOCurrency | VARCHAR(30) | No | The local currency as an uppercase, three-character code from ISO 4217. The system copies the value into the output string of monetary numeric data whenever a C appears in the corresponding output FORMAT phrase. |
| Dual Currency Name | VARCHAR(30) | No | The dual currency as a completely spelled-out currency name. The system copies the value into the output string of monetary numeric data whenever an A appears in the corresponding output FORMAT phrase. |
| Dual Currency | VARCHAR(30) | No | The dual currency symbol. The system copies the value into the output string of monetary numeric data whenever an O appears in the corresponding output FORMAT phrase. |
| Dual ISOCurrency | VARCHAR(30) | No | The dual currency as an uppercase, three-character code from ISO 4217. The system copies the value into the output string of monetary numeric data whenever a U appears in the corresponding output FORMAT phrase. |
| Default ByteInt Format | VARCHAR(30) | No | The default output format applied to BYTEINT data types. |
| Default Integer Format | VARCHAR(30) | No | The default output format applied to INTEGER data types. |
| Default SmallInt Format | VARCHAR(30) | No | The default output format applied to SMALLINT data types. |
| Default Numeric Format | VARCHAR(30) | No | The default output format applied to NUMERIC and DECIMAL data types. |
| Default Real Format | VARCHAR(30) | No | The default output format applied to REAL, DOUBLE PRECISION, and FLOAT data types. |
| Default Time Format | VARCHAR(30) | No | The default output format applied to TIME and TIME WITH TIME ZONE data types |
| Default Timestamp Format | VARCHAR(30) | No | The default output format applied to TIMESTAMP and TIMESTAMP WITH TIME ZONE data types. |
| Current Role | VARCHAR(30) | No | Reports the current role for the session. If no role applies, the system returns an empty string. |

| Attribute | Data Type | Nullable | Description |
|-------------------------|---------------|----------|---|
| Logon Account | VARCHAR(30) | No | Reports the account name used to log on to the database. |
| Profile | VARCHAR(30) | No | Reports the name of the profile associated with this session. See "User Name" earlier in this table for information about the data type and system-defined character sets for Profile. |
| LDAP | CHARACTER(1) | No | Reports whether the session is authenticated by an LDAP directory and whether the directory user is mapped to a database user. <ul style="list-style-type: none"> • N specifies that the session is not LDAP-based. • P specifies the session of an LDAP user mapped to a database user. • X specifies the session of an LDAP user not mapped to a database user. |
| Audit Trail ID | VARCHAR(30) | No | Reports the name used for access logging, regardless of whether the user is Directory-managed or database-managed. The system stores values for external users who do not have a corresponding permanent user name in the User Name column. |
| Current Isolation Level | CHARACTER(2) | No | Reports the default read-only isolation level for the current session. <ul style="list-style-type: none"> • RU specifies that the default read-only isolation level for the current session is Read Uncommitted. The default read-only locking severity for the session is ACCESS. • SR specifies that the default read-only isolation level for the current session is Serializable. The default read-only locking severity for the session is READ. |
| Default BigInt Format | VARCHAR(30) | No | Reports the default output format for the BIGINT data type. The value is set in the Specification for Data Formatting (SDF) file by the database administrator. If the attribute is Default BIGINT Format, the default output format applies to BIGINT data type For information about changing the default data type output format in the SDF and the definition of formatting characters, see <i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143. |
| Query Band | VARCHAR(8201) | No | If there is no query band, the Query Band text contains an empty string. If there is only a transaction query band, the Query Band text contains the following string. |

| Attribute | Data Type | Nullable | Description |
|--------------------|---------------|----------|--|
| | | | <p>=T> <i>transaction_queryband</i></p> <p>where <i>transaction_queryband</i> is the query band for the transaction.</p> <p>If there is only a session query band, the Query Band text contains the following string.</p> <p>=S> <i>session_queryband</i></p> <p>where <i>session_queryband</i> is the query band for the current session.</p> <p>If there is both a transaction query band and a session query band, the Query Band text contains the following concatenated string.</p> <p>=T> <i>transaction_queryband</i> =S> <i>session_queryband</i></p> <p>where <i>transaction_queryband</i> is the query band for the transaction and <i>session_queryband</i> is the query band for the current session.</p> |
| Proxy User | VARCHAR(30) | No | <p>If there is a proxy connection in effect for the session, reports the name of the Proxy User.</p> <p>If there is no proxy connection in effect for the session, the system returns an empty string.</p> |
| Proxy Role | VARCHAR(30) | No | <p>If there is a proxy connection in effect for the session and a role has been assigned to the Proxy User, Proxy Role reports the name of that role.</p> <p>If there is no proxy connection in effect for the session, or if no roles have been assigned to the Proxy User, Proxy Role returns an empty string.</p> |
| Constraint1Name | VARCHAR(30) | No | <p>The name of a row level security constraint.</p> <p>If no row level security constraint applies, the <i>Constraint_n_Name</i> is null.</p> <p>If the session has more than 1 security constraint, the system displays each with a unique identifying number, for example:</p> <pre>Constraint1Name Constraint1Value Constraint2Name Constraint2Value...</pre> |
| Constraint1Value | CHAR(68) | No | <p>The value code for the default session label for the Constraint <i>n</i>Name.</p> <p>If no row level security constraint applies, the <i>Constraint_n_Value</i> is null.</p> |
| Temporal Qualifier | VARCHAR(1024) | No | <p>Reports the default temporal attribute for the current session.</p> |

| Attribute | Data Type | Nullable | Description |
|----------------------------------|--------------|----------|--|
| | | | For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i> , B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i> , B035-1182. |
| Calendar Name | VARCHAR(128) | No | Reports the default calendar for the current session. |
| Export Width Rule Set | CHAR(40) | No | The export width rule in effect for the session. For details, see <i>Teradata Vantage™ - Advanced SQL Engine International Character Set Support</i> , B035-1125. |
| Default Number Format | VARCHAR(30) | No | The default output format for the NUMBER data type. |
| TTGranularity | VARCHAR(15) | No | Indicates the temporal transaction time granularity for the session. For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i> , B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i> , B035-1182. |
| Redrive Participation | VARCHAR(60) | No | Reports whether Redrive protection is enabled or disabled for the current session. |
| User Dictionary Name | VARCHAR(128) | No | <p>The attributes shown in this section are based upon and supersede the corresponding older attributes:</p> <ul style="list-style-type: none"> • User Name • Account Name • Current Database • Current Role • Logon Account • Profile • Audit Trail ID • Proxy User • Proxy Role • Constraint <p>The system reports up to 8 numbered constraint attributes if constraints apply to the session user.</p> <p>The older attributes are retained for compatibility with existing applications.</p> <p>For details, see the topics beginning with Object Name and Title Data in HELP Reports.</p> |
| User SQL Name | VARCHAR(644) | No | |
| User UEscape | VARCHAR(1) | Yes | |
| Account Dictionary Name | VARCHAR(128) | No | |
| Account SQL Name | VARCHAR(644) | No | |
| Account UEscape | VARCHAR(1) | Yes | |
| Current Database Dictionary Name | VARCHAR(128) | No | |
| Current Database SQL Name | VARCHAR(644) | No | |
| Current Database UEscape | VARCHAR(1) | Yes | |
| Current Role Dictionary Name | VARCHAR(128) | Yes | |
| Current Role SQL Name | VARCHAR(644) | Yes | |

| Attribute | Data Type | Nullable | Description |
|--------------------------------|--------------|----------|-------------|
| Current Role UEscape | VARCHAR(1) | Yes | |
| Logon Account Dictionary Name | VARCHAR(128) | No | |
| Logon Account SQL Name | VARCHAR(644) | No | |
| Logon Account UEscape | VARCHAR(1) | Yes | |
| Profile Dictionary Name | VARCHAR(128) | Yes | |
| Profile SQL Name | VARCHAR(644) | Yes | |
| Profile UEscape | VARCHAR(1) | Yes | |
| Audit Trail Id Dictionary Name | VARCHAR(128) | No | |
| Current Database SQL Name | VARCHAR(644) | No | |
| Current Database UEscape | VARCHAR(1) | Yes | |
| Audit Trail Id SQL Name | VARCHAR(644) | No | |
| Audit Trail Id UEscape | VARCHAR(1) | Yes | |
| Proxy User Dictionary Name | VARCHAR(128) | Yes | |
| Proxy User SQL Name | VARCHAR(644) | Yes | |
| Proxy User UEscape | VARCHAR(1) | Yes | |
| Proxy Role Dictionary Name | VARCHAR(128) | Yes | |
| Proxy Role SQL Name | VARCHAR(644) | Yes | |
| Proxy Role UEscape | VARCHAR(1) | Yes | |
| Constraint _n Dictionary Name | VARCHAR(128) | Yes | |

| Attribute | Data Type | Nullable | Description |
|--------------------------------------|---------------|----------|--|
| Constraint_ <i>n</i> SQL Name | VARCHAR(644) | Yes | |
| Constraint_ <i>n</i> Name UEscape | VARCHAR(1) | Yes | |
| Zone Name | VARCHAR(128) | N | Name of zone for the session. |
| SearchUIFDBPath | | Yes | |
| Transaction QueryBand | VARCHAR(4096) | Yes | List of the query bands set for the transaction. |
| Session QueryBand | VARCHAR(4096) | Yes | List of the query bands set for the session. |
| Profile QueryBand | VARCHAR(4096) | Yes | List of the query bands set for the profile. |
| Unicode Pass Through | CHAR(1) | No | The Unicode Pass Through (UPT) setting for the session: <ul style="list-style-type: none"> • S for session level UPT • F for UPT off, the default. |

HELP SESSION CONSTRAINT

Session Constraint Attributes

The HELP SESSION CONSTRAINT command is similar to the HELP SESSION command except that it returns additional information about row-level security constraints.

The following table shows the attributes returned for each row-level security constraint that applies to the session in addition to the normal HELP SESSION attributes.

| Additional Attributes | Data Type | Nullable? | Description |
|--|--------------|-----------|--|
| Constraint_ <i>n</i> _ Name | VARCHAR(30) | Yes | The name of a row-level security constraint that applies to a session. If no row-level security constraint applies, the Constraint_ <i>n</i> _Name is null. |
| Constraint_ <i>n</i> _ Value | | Yes | The constraint value for the constraint name that immediately precedes it. If no row-level security constraint applies, the Constraint_ <i>n</i> _Value is null. |
| Constraint_ <i>n</i> _ Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are based upon and supersede the corresponding older Constraint_ <i>n</i> _Name attribute, while providing additional functionality. The older attribute is retained for compatibility with existing applications. The system returns this set of attributes for each constraint applicable to the session. For details, see the topics beginning with Object Name and Title Data in HELP Reports . |
| Constraint_ <i>n</i> _ SQL Name | VARCHAR(644) | No | |
| Constraint_ <i>n</i> _ Uescape | VARCHAR(1) | Yes | |

HELP STATISTICS (Optimizer Form)

HELP STATISTICS Usage Restriction

HELP STATISTICS is considered obsolete. Although HELP STATISTICS is still supported for the current Vantage release, Teradata recommends that you instead use other statistics reporting options.

| For... | Instead of HELP STATISTICS, use... |
|--------------------|---|
| Column statistics | SHOW STATISTICS |
| Summary statistics | Statistics views, for example: <ul style="list-style-type: none"> • StatsV • TablesStatsV • TempTableStatsV • ExpStatsV • IndexStatsV • MulticolumnStatsV • MultiExpStatsV • ReconfigTableStatsV For information, see <i>Teradata Vantage™ - Data Dictionary</i> , B035-1092. |

Locks and Concurrency

Vantage places rowhash-level READ or ACCESS locks on *DBC.StatsTbl* to retrieve statistics information.

HELP STATISTICS and Non-Updatable Views

You cannot execute a HELP STATISTICS request against a non-updatable view.

Support for Global Temporary and Volatile Tables

If you specify the keyword TEMPORARY then the statistics for the local instance of a materialized global temporary table or volatile table in the current session are displayed; else, the statistics for the base global temporary table are displayed.

How HELP STATISTICS Behaves When No Statistics Exist

If no statistics have been collected for the specified table, the system returns a warning message to the requestor.

Summary Optimizer Statistics Attributes

The data types for the table attributes reported by a summary HELP STATISTICS request are as follows:

| Attribute | Description | Data Type |
|---------------|--|--|
| Date | The date on which statistics were last collected. | CHARACTER(8) CHARACTER SET UNICODE |
| Time | The time at which statistics were last collected. | CHARACTER(8) CHARACTER SET UNICODE |
| Unique Values | An estimate of the number of unique values for the specified column set. | VARCHAR(20) CHARACTER SET UNICODE |
| Column Names | The names of the columns for which summary statistics are reported. | <p>VARCHAR(2000)</p> <p>The requirement of a 2,000 character maximum derives from the maximum number of columns per index and the maximum length of a column name plus overhead: $(64 * 30) + 63 = 1983$ characters.</p> <p>On a standard language support system, the data type for Column Names is VARCHAR(2000) CHARACTER SET LATIN.</p> <p>On a Japanese language support system, the data type for Column Names is VARCHAR(2000) CHARACTER SET UNICODE if a multibyte site-defined session character set such as UTF-8, UTF-16, Chinese, or Korean is used.</p> <p>For any other language support system, the data type for Column Names is VARCHAR(2000) CHARACTER SET UNICODE.</p> <p>The server character set for Column Names is Unicode, which gets translated to the client character set.</p> <p>If there is multibyte character data, Column Names uses the UTF-8, UTF-16, or some other character sets that supports multibyte client character sets.</p> |

Related Information

For information about collecting and dropping Optimizer statistics, see the following:

- [COLLECT STATISTICS \(Optimizer Form\)](#)
- [DROP STATISTICS \(Optimizer Form\)](#)
- COLLECT STATISTICS and DROP STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

For information about reporting detailed Optimizer statistics, see the following:

- [SHOW STATISTICS](#)
- SHOW STATISTICS in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

Also see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 and *Teradata Vantage™ - Database Design*, B035-1094 for information about statistics, their collection, and their recollection.

HELP STATISTICS (QCD Form)

HELP STATISTICS and Non-Updatable Views

You cannot execute a HELP STATISTICS request against a non-updatable view.

How HELP STATISTICS Behaves When No Statistics Exist

If no statistics have been collected for the specified table, the system returns a warning message to the requestor.

How To Report Summary and Detailed Statistics HELP

HELP STATISTICS returns only summary interval statistics for columns and indexes of the specified table for which statistics have been collected.

To report detailed interval statistics, you must use the SHOW STATISTICS statement.

Summary QCD Statistics Attributes

The data types for the table attributes reported by a summary HELP STATISTICS request are as follows:

| Attribute | Description | Data Type |
|---------------|--|---|
| Date | The date on which statistics were last collected. | CHARACTER(8) CHARACTER SET UNICODE |
| Time | The time at which statistics were last collected. | CHARACTER(8) CHARACTER SET UNICODE |
| Unique Values | An estimate of the number of unique values for the specified column set. | VARCHAR(20) CHARACTER SET UNICODE |
| Column Names | The names of the columns for which summary statistics are reported. | VARCHAR(2000) The requirement of a 2,000 character maximum derives from the maximum number of columns per index and the maximum length of a column name plus overhead: $(64 * 30) + 63 = 1983$ characters. On a standard language support system, the data type for Column Names is VARCHAR(2000) CHARACTER SET LATIN. On a Japanese language support system, the data type for Column Names is VARCHAR(2000) CHARACTER SET UNICODE if a multibyte site-defined session character set such as UTF-8, UTF-16, Chinese, or Korean is used. |

| Attribute | Description | Data Type |
|-----------|-------------|--|
| | | <p>For any other language support system, the data type for Column Names is VARCHAR(2000) CHARACTER SET UNICODE.</p> <p>The server character set for Column Names is Unicode, which gets translated to the client character set.</p> <p>If there is multibyte character data, Column Names uses the UTF-8, UTF-16, or some other character sets that supports multibyte client character sets.</p> |

Statistics Attribute Definitions

The following tables describe the attributes reported by the HELP STATISTICS (QCD Form) statement.

Inter-Interval Statistics

The following group of columns is reported once per HELP STATISTICS request.

| Attribute | Description |
|-------------------------|---|
| Date | The date on which statistics were last collected. |
| Time | The time at which statistics were last collected. |
| Number of Rows | An estimate of the cardinality of the table. |
| Number of Nulls | An estimate of the number of nulls for the specified column or index. |
| Number of All Nulls | An estimate of the number of rows containing nulls for all columns of the specified column set or index. |
| Average AMP RPV | <p>Overall average of the average rows per value from each AMP.</p> <p>Note that the system collects Average AMP Rows Per Value statistics only for NUSI columns. The value is used to improve nested join costing.</p> <p>If this is not a NUSI column, the system reports a null.</p> |
| Number of Intervals | The number of intervals in the frequency distribution histogram containing the column or index statistics. |
| Number of Unique Values | An estimate of the number of unique values for the column or index. |
| Numeric | <p>Identifies whether the data type of the column set reported on is numeric or non-numeric.</p> <ul style="list-style-type: none"> • 0 specifies that the data type is non-numeric. • Non-zero specifies that the data type is numeric. |
| Sampled | <p>Identifies whether the statistics were collected from all rows in the table or from a sampled subset.</p> <ul style="list-style-type: none"> • 0 specifies that statistics have been collected on all the rows for the specified table. |

| Attribute | Description |
|-----------------|--|
| | <ul style="list-style-type: none"> Nonzero specifies that statistics have been collected on a sampled subset of the rows for the specified table. |
| Sampled Percent | The approximate percentage of total rows in the table included in the sample. |
| Version | The version number of the statistics structure in effect when the statistics were collected. |
| Min Value | An estimate of the smallest value for the specified column or index in the specified table. |
| Mode Value | An estimate of the most frequently occurring value or values for the column or index in the specified table. |
| Mode Frequency | An estimate of the number of rows having the modal value for the specified column or index in the specified table. |

Intra-Interval Statistics, Equal-Height

The following group of columns is reported for equal-height intervals returned by a HELP STATISTICS report.

For more information about equal-height intervals, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

| Attribute | Description |
|------------------|---|
| Max Value | An estimate of the largest value for the column or index in the interval. |
| Mode Value | An estimate of the most frequently occurring value or values for the column or index in the interval. |
| Mode Frequency | An estimate of the number of rows in the interval having its modal value for the column or index. |
| Non-Modal Values | An estimate of the number of distinct non-modal values for the column or index in the interval. |
| Non-Modal Rows | <p>An estimate of the number of rows in the interval with values for the specified column or index.</p> <p>This is a measure of the skewness of the distribution of the index or column values within the interval.</p> |

Intra-Interval Statistics, High-Biased

The following group of columns is reported for high-biased intervals returned by a HELP STATISTICS report.

For more information about high-biased intervals, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

| Attribute | Description |
|------------------|--|
| Max Value | If two values are stored in the interval, then Max is the value for the second high frequency loner. |
| Mode Value | If two values are stored in the interval, then Mode is the value for the first high frequency loner. A maximum of two modal values is stored per interval for a skewed distribution. |
| Mode Frequency | The number of rows in the interval having the modal value. |
| Non-Modal Values | A code indicating the number of loner values for the interval. <ul style="list-style-type: none"> • -1 specifies that the interval has one loner value. • -2 specifies that the interval has two loner values. |
| Non-Modal Rows | The number of rows in the interval having the maximum value. |

Related Information

For information about collecting and dropping QCD statistics, see the following in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146:

- COLLECT STATISTICS (QCD Form)
- DROP STATISTICS (QCD Form)

Also see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for information about QCD statistics.

HELP TABLE

HELP TABLE and Column-Partitioned NoPI Tables and Join Indexes

A HELP TABLE request provides information about each of the columns in the table, and its output is the same as the output for an nonpartitioned table or join index.

Table, Hash Index, and Join Index Attributes

A HELP TABLE request returns a set of attributes for each column in the specified table, as shown in the following table of attributes.

The columns or parameters are listed in the order in which they were defined.

| Attribute | Data Type | Nullable? | Comments |
|-------------|---------------|-----------|--|
| Column Name | VARCHAR(30) | No | The name of a column in the specified table. Interpret as the equivalent of CF, or CHARACTER(<i>n</i>) data type. When DATEFORM=ANSIDATE, this column should be described externally as CHARACTER(10). |
| Type | CHARACTER(2) | No | The data type for Column Name. The character data types Latin, Unicode, Graphic, KanjiSJIS, and Kanji1 are distinguished with respect to CF and CV by their respective value for the Character Type column. Default output format for the code is X(2). See Data Type Codes for a complete list of data types and codes. |
| Comment | VARCHAR (255) | Yes | The contents of the optional comment field for the table, if a comment was created |
| Nullable | CHARACTER | Yes | Whether the column accepts nulls. <ul style="list-style-type: none"> • N specifies that the column does not accept nulls. • Y specifies that the column accepts nulls. |
| Format | VARCHAR(30) | Yes | The format for Column Name. The CHARACTER SET attribute for FORMAT is always Unicode regardless of the system language support mode and session character set. With an explicit format, DATE. The general rule for reporting column FORMAT attributes by HELP TABLE is that a format string is only reported when it does not match the default format for the column. |

| Attribute | Data Type | Nullable? | Comments |
|---------------------------|--------------------------------|-----------|---|
| | | | <p>The exception to this rule is columns that have the DATE data type. In this case, the system <i>always</i> reports the FORMAT string.</p> <p>With an implicit format created in INTEGERDATE mode, 'YY/MM/DD'.</p> <p>With an implicit format created in ANSIDATE mode, 'YYYY/MM/DD'.</p> |
| Title | VARCHAR(60) | Yes | The title for the column, if a title exists. |
| Max Length | INTEGER | Yes | <p>Maximum amount of storage in bytes.</p> <p>MaxLength is the length of the <i>external</i> CharFix representation of the column, defined as CHARACTER(MaxLength) in USING clauses and any other place this needs to be specified.</p> <p>MaxLength for the DATE type is 4, representing the internal storage size.</p> <p>Note that while MaxLength is usually expressed as the internal size, presentation of the external size is more appropriate for these data types.</p> <p>See the external representation for each individual DateTime and Interval type in <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143 to determine the MaxLength values.</p> <p>For UDTs, Max Length is the maximum length of the external type associated with the UDT formatted as -(10)9.</p> <p>You can use the StatementInfo parcel to return the description of items within the Data Field of the corresponding indicator mode Record parcels.</p> <p>For example, the FastLoad .TABLE command can be executed in Indicator Mode to get the additional information of data type and length of all table attributes. That information can then be used to fetch the data correctly in any release of Vantage.</p> |
| Decimal Total Digits | SMALLINT Formatted as -(5)9 | Yes | <p>If type is DECIMAL.</p> <p>For INTERVAL types, returns a value from 1–4 representing the number of digits in the leading field of the interval (number of non-fractional digits for INTERVAL SECOND).</p> <p>This column is null for TIME and TIMESTAMP data types.</p> |
| Decimal Fractional Digits | SMALLINT Formatted as -(5)9 | Yes | <p>If type is DECIMAL.</p> <p>For TIME and TIMESTAMP types and all INTERVAL types with a SECOND field, this column returns a value of 0–6, indicating the fractional precision of seconds.</p> <p>For INTERVAL types without a SECOND field, the column returns null.</p> |

| Attribute | Data Type | Nullable? | Comments |
|----------------|---------------|-----------|--|
| Range Low | FLOAT | Yes | Always null. BETWEEN clause is included in column constraint. |
| Range High | FLOAT | Yes | Always null. BETWEEN clause is included in column constraint. |
| Uppercase | CHARACTER | Yes | Reported if the column data type is CHARACTER or VARCHAR. <ul style="list-style-type: none"> • B specifies that column attributes for both UPPERCASE and CASESPECIFIC are defined. • C specifies that the column attributes for CASESPECIFIC is defined. • N specifies that column attributes for neither UPPERCASE nor CASESPECIFIC are defined. • U specifies that the column attributes for UPPERCASE is defined. Otherwise null. |
| Table/View? | CHARACTER | No | Object type. The value reported depends on whether the request is HELP HASH INDEX, HELP JOIN INDEX, or HELP TABLE. <ul style="list-style-type: none"> • I specifies join index. • N specifies hash index. • O specifies NoPI table. • Q specifies queue table. • T specifies base table. |
| Default Value | VARCHAR (255) | Yes | Returns the default value (as text) for the parameter. The value is null if there is no default value. |
| Character Type | SMALLINT | Yes | Returns a code for the server character set for the column. <ul style="list-style-type: none"> • 1 specifies LATIN server character data type columns. • 2 specifies UNICODE server character data type columns. • 3 specifies KANJISJIS server character data type columns. • 4 specifies GRAPHIC server character data type columns. • 5 specifies KANJI1 server character data type columns. If the data type of the column is not character, returns null. |
| IDCol Type | CHARACTER(2) | Yes | Returns a code for the type of identity column. <ul style="list-style-type: none"> • GA specifies Generated Always. |

| Attribute | Data Type | Nullable? | Comments |
|---------------------------------|--------------|-----------|--|
| | | | <ul style="list-style-type: none"> GD specifies Generated by Default. |
| UDT Name | VARCHAR(61) | Yes | Returns the unqualified name of the UDT. |
| Temporal Column | CHARACTER(2) | Yes | <p>Reports whether a column is temporal and if it is part of a Temporal Relationship constraint.</p> <ul style="list-style-type: none"> N specifies a non-temporal column. R specifies that the column is part of a temporal relationship constraint. T specifies a TRANSACTIONTIME column. V specifies a VALIDTIME column. <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Current ValidTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Current VALIDTIME column.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Sequenced ValidTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Sequenced VALIDTIME column.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| NonSequenced ValidTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Nonsequenced VALIDTIME column.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Current TransactionTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Current TRANSACTIONTIME column.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Column Dictionary Name | VARCHAR(128) | No | <p>The attributes shown in this section are based upon and supersede the corresponding older attributes, Column Name, Title, and UDT Name, while providing additional functionality.</p> <p>The older attributes are retained for compatibility with existing applications.</p> <p>The system returns this set of attributes for each column in the HELP TABLE.</p> |
| Column SQL Name | VARCHAR(644) | No | |
| Column Name UEscape | VARCHAR(1) | Yes | |

| Attribute | Data Type | Nullable? | Comments |
|---------------------|---------------|-----------|---|
| Dictionary Title | VARCHAR(256) | No | For details, see the topics beginning with Object Name and Title Data in HELP Reports . |
| SQL Title | VARCHAR(1284) | No | |
| Title UEscape | VARCHAR(1) | Yes | |
| UDT Dictionary Name | VARCHAR(128) | Yes | |
| UDT SQL Name | VARCHAR(644) | Yes | |
| UDT Name UEscape | VARCHAR(1) | Yes | |

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*.

HELP TRANSFORM

HELP TRANSFORM Attributes

The following table lists the attributes reported by HELP TRANSFORM:

| Attribute | Data Type | Nullable? | Description |
|------------------------------|--------------|-----------|--|
| Group | VARCHAR(30) | No | The name of the transform group for this UDT. |
| Predefined Type | VARCHAR(30) | No | The predefined data type that is associated with either the fromsql or tosql routine of this transform group. |
| from-sql | VARCHAR(61) | Yes | <ul style="list-style-type: none"> If the UDT is a structured type, this is the fully qualified specific name of the external routine that provides the fromsql transform functionality. If the UDT is a distinct type and its transform functionality is system-generated, this is the word <i>Local</i>. If FROM-SQL is null, then TO-SQL cannot be null. |
| to-sql | VARCHAR(61) | Yes | <ul style="list-style-type: none"> If the UDT is a structured type, this is the fully qualified specific name of the external routine that provides the tosql transform functionality. If the UDT is a distinct type and its transform functionality is system-generated, this is the word <i>Local</i>. If To-SQL is null, then FROM-SQL cannot be null. |
| Dictionary Group | VARCHAR(128) | No | <p>The attributes shown in this section are based upon and supersede the corresponding older attributes, Group Name, from-sql, and to-sql, while providing additional functionality.</p> <p>The older attributes are retained for compatibility with existing applications.</p> <p>The from-sql Database and to-sql Database attributes identify the database that contains the external routine that performs the transform function.</p> <p>The system returns this set of attributes for each column in the HELP TABLE.</p> <p>For details, see the topics beginning with Object Name and Title Data in HELP Reports.</p> |
| SQL Group | VARCHAR(644) | No | |
| Group UEscape | VARCHAR(1) | Yes | |
| from-sql Database Dictionary | VARCHAR(128) | No | |
| from-sql Database SQL | VARCHAR(644) | No | |
| from-sql Database UEscape | VARCHAR(1) | Yes | |
| from-sql Dictionary | VARCHAR(128) | No | |
| from-sql SQL | VARCHAR(644) | No | |

| Attribute | Data Type | Nullable? | Description |
|----------------------------------|--------------|-----------|-------------|
| from-sql UEScape | VARCHAR(1) | Yes | |
| to-sql Database Dictionary | VARCHAR(128) | No | |
| to-sql Database SQL | VARCHAR(644) | No | |
| to-sql Database UEScape | VARCHAR(1) | Yes | |
| to-sql Dictionary | VARCHAR(128) | No | |
| to-sql SQL | VARCHAR(644) | No | |
| to-sql UEScape | VARCHAR(1) | Yes | |

HELP TRIGGER

Trigger Attributes

| Attribute | Data Type | NULLABLE? | Description |
|-------------------------|--------------|-----------|--|
| Name | VARCHAR(30) | No | The name of the trigger. |
| Actiontime | CHAR(1) | No | <p>When the trigger fires.</p> <ul style="list-style-type: none"> • A indicates that the trigger fires after the execution of the triggering statement. • B indicates that the trigger fires before the triggering statement. <p>For triggers created in prior database releases with INSTEAD OF as the action time, the code I is displayed. Such triggers are not valid.</p> |
| Decimal Order Value | INTEGER | No | <p>The optional integer value used to specify the order in which the trigger fires, when multiple triggers are defined on the same table.</p> <p>This value overrides the ANSI-specified default order of execution.</p> <p>If no ORDER value has been specified with the trigger, the default value of 32,767 is displayed.</p> |
| Creation Timestamp | TIMESTAMP(0) | No | When the trigger was first created, or when its timestamp was last altered using the ALTER TRIGGER ... TIMESTAMP statement. |
| Event | CHAR(1) | No | <p>The triggering event, that is, the SQL DML statement type that causes the trigger to fire.</p> <ul style="list-style-type: none"> • D represents DELETE. • I represents INSERT, including INSERT... SELECT, Atomic Upsert, and MERGE inserts. • U represents UPDATE, including Atomic Upsert, and MERGE updates. |
| Kind | CHAR(1) | No | <p>The trigger type as defined by the FOR EACH clause.</p> <ul style="list-style-type: none"> • R is a row trigger. • S is a statement trigger. |
| Enabled | CHAR(1) | No | <p>Whether the trigger is activated or not.</p> <ul style="list-style-type: none"> • N specifies that the trigger is disabled. • Y specifies that the trigger is enabled. |
| Comment | VARCHAR(255) | Yes | Optional text commenting on the named trigger. This attribute can be null. |
| Trigger Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are based upon and supersede the corresponding older Name attribute, while providing additional functionality. |

| Attribute | Data Type | NULLABLE? | Description |
|------------------|--------------|-----------|--|
| Trigger SQL Name | VARCHAR(644) | No | The older attribute is retained for compatibility with existing applications. For details, see the topics beginning with Object Name and Title Data in HELP Reports . |
| Trigger UEscape | VARCHAR(1) | Yes | |

HELP TYPE

General Type Attributes

The following table lists the attributes reported by HELP TYPE with no options:

| Attribute | Data Type | Nullable? | Description |
|----------------------|-----------------------------------|-----------|--|
| Name | VARCHAR(61) | No | The name of the UDT or ARRAY/VARRAY type. |
| Internal Type | CHARACTER(2) | No | <p>A code for the UDT or ARRAY/VARRAY type.</p> <ul style="list-style-type: none"> A1 is a one-dimensional ARRAY/VARRAY. AN is a multidimensional ARRAY/VARRAY. UD is a distinct UDT. US is a structured UDT. UT is a Teradata internal UDT. <p>The external type for both A1 and AN internal types is always CV (VARCHAR).</p> |
| External Type | CHARACTER(2) | Yes. | <p>The external data type of the UDT or ARRAY/VARRAY as defined by its fromsql transform.</p> <p>See Data Type Codes for a list of the data type codes and their meanings.</p> <p>A null is reported if the UDT or ARRAY/VARRAY has no fromsql transform defined for it.</p> <p>With an explicit format, DATE is imported and exported according to that format, as indicated in the following bullets.</p> <ul style="list-style-type: none"> With an Implicit format created in INTEGERDATE mode, the export format is 'YY/MM/DD'. With an Explicit format created in ANSIDATE mode, the export format is 'YYYY-MM-DD'. <p>MaxLength is 4, representing the internal storage size.</p> |
| Max Length | INTEGER Formatted as -(10)9 | Yes | <p>Maximum length of the external type in bytes.</p> <p>For CHARACTER data, Max Length is the length of the column declaration in bytes.</p> <p>For UDTs and ARRAY/VARRAY types, Max Length is the maximum transformed length of the external type associated with the UDT formatted as -(10)9.</p> <p>You can use a CLIV2 StatementInfo parcel to return the description of items within the Data Field of the corresponding indicator mode Record parcels.</p> <p>For example, the FastLoad .TABLE command can be executed in Indicator Mode to get the additional information of data type and length of all table attributes. That information can then be used to fetch the data correctly in any database release.</p> |
| Decimal Total Digits | SMALLINT | Yes | <p>Only reported for numeric data types, otherwise is null.</p> <p>For INTERVAL types, returns a value from 1 - 4 representing the number of digits in the leading field</p> |

| Attribute | Data Type | Nullable? | Description |
|---------------------------|--------------|-----------|---|
| | | | <p>of the interval (number of non-fractional digits for INTERVAL SECOND).</p> <p>This column is null for other TIME and TIMESTAMP data types.</p> <p>Regardless of the underlying data type, ARRAY/VARRAY types always return null for this attribute.</p> |
| Decimal Fractional Digits | SMALLINT | Yes | <p>Only reported for numeric data types, otherwise is null. For TIME and TIMESTAMP types and all INTERVAL types with a SECOND field, this column returns a value of 0 - 6, indicating the fractional precision of seconds. For INTERVAL types without a SECOND field, the column returns null.</p> <p>The character data types Latin, Unicode, Graphic, KanjiSJIS, and Kanji1 are distinguished with respect to CF and CV by their respective value for the Character Type column.</p> <p>Regardless of the underlying data type, ARRAY/VARRAY types always return null for this attribute.</p> |
| Contains LOB | CHARACTER(1) | No | <p>Records whether the UDT contains BLOB or CLOB predefined data types.</p> <ul style="list-style-type: none"> • N specifies that the UDT does not contain a LOB data type. • Y specifies that the UDT contains a LOB data type. |
| Ordering | CHARACTER(1) | No | <p>Records whether the UDT has a defined ordering.</p> <ul style="list-style-type: none"> • E specifies that only equality and non-equality comparisons are supported for the UDT. This code is only supported for Teradata internal UDTs. • F specifies that all comparison operations are supported for the UDT. • N specifies that the UDT does not have a defined ordering. |
| Ordering Category | CHARACTER(1) | Yes | <p>Records the ordering category for the UDT.</p> <ul style="list-style-type: none"> • M specifies that the UDT is mapped, meaning it has a defined ordering. • R specifies that the UDT ordering is relative. This code is only supported for Teradata internal UDTs. • Null specifies that the UDT does not have a defined ordering. |
| Ordering Routine | VARCHAR(61) | Yes | <p>The type of ordering routine is one of the following:</p> <ul style="list-style-type: none"> • For user-defined routines, the fully qualified name of the external routine that provides ordering functionality in the form <i>database.object</i>. <p>If the value is not a fully qualified name, the system returns NULL for the 6 naming attributes beginning</p> |

| Attribute | Data Type | Nullable? | Description |
|-------------|--------------|-----------|--|
| | | | <p>with Dictionary Ordering Routine Database, shown later in this table.</p> <ul style="list-style-type: none"> • S for a system ordering routine • L for a local ordering routine <p>The column is null if a UDT does not have a defined ordering.</p> |
| Cast | CHARACTER(1) | No | <p>Records whether at least one cast is defined for the UDT.</p> <ul style="list-style-type: none"> • N specifies that no casts are defined for the UDT. • Y specifies that casts are defined for the UDT. |
| Transform | CHARACTER(1) | No | <p>Records whether a transform is defined for the UDT or not.</p> <ul style="list-style-type: none"> • N specifies that no transform is defined for the UDT. • Y specifies that a transform is defined for the UDT. |
| Method | CHARACTER(1) | Yes | <p>Indicates whether methods are associated with this UDT.</p> <ul style="list-style-type: none"> • N specifies that no methods are associated with the UDT. • Y specifies that at least one method is associated with the UDT. <p>Structured UDTs always report Y for this column because of their system-generated instance and mutator methods.</p> <p>No method names are reported with HELP TYPE.</p> <p>To report a complete list of the methods associated with a UDT, use the HELP METHOD statement (see HELP METHOD).</p> |
| Char Type | CHARACTER(1) | No | <p>Returns the character data type for a CHAR or VARCHAR column.</p> <ul style="list-style-type: none"> • 1 specifies LATIN server character data type columns. • 2 specifies UNICODE server character data type columns. • 3 specifies KANJISJIS server character data type columns. • 4 specifies GRAPHIC server character data type columns. • 5 specifies KANJI1 server character data type columns. <p>If the data type of the column is not character, returns a null.</p> |
| Array (Y/N) | CHARACTER(1) | No | <p>Whether the UDT is an ARRAY type.</p> <ul style="list-style-type: none"> • N specifies that the UDT is not an ARRAY type. • Y specifies that the UDT is an ARRAY type. |

| Attribute | Data Type | Nullable? | Description |
|--------------------------------------|-------------------------------------|-----------|---|
| Dimensions | BYTEINT formatted as --9 | Yes | Returns the number of dimensions the ARRAY type has. <ul style="list-style-type: none"> For a one-dimensional ARRAY type (Internal Type code A1), the Dimensions value is always 1. For a multidimensional ARRAY type (Internal Type code AN), the Dimensions value is an integer in the range 1-5, inclusive. |
| Element Type | CHARACTER(2) | Yes | Returns the data type for the ARRAY element. See Data Type Codes for a list of the possible data type codes that can be reported for Element . |
| UDT Name | VARCHAR(61) | Yes | Returns the qualified name of the UDT that is the element type of the ARRAY type if the element type of the array is a UDT. The server character set specification for the column is CHARACTER SET UNICODE UPPER CASE NOT CASESPECIFIC. |
| Array Scope | VARCHAR(3200) formatted as X(45) | Yes | Returns the scope of the array for an ARRAY type in the form $[n:m]$ for a one-dimensional array and $[n:m] \dots [n:m]$ for a multidimensional array, where n is the lower bound for each dimension of the type and m is the upper bound for each dimension of the type. To see more than 45 characters, use record mode. The server character set specification for the column is CHARACTER SET LATIN UPPER CASE NOT CASESPECIFIC. For a one-dimensional array, the lower bound always begins with 1. The bound ends with m , where m is the maximum cardinality of the array. For a multidimensional array, the string contains the scope information specified for the type when it was created. If explicit lower bounds were not specified for the multidimensional array when it was created, then the lower bound always begins with 1, and ends with m , where m is the maximum cardinality of the array. |
| Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are based upon and supersede the corresponding older Name, Ordering Routine, and UDT Name attributes, while providing additional functionality with several new attributes. The older attributes are retained for compatibility with existing applications. For details, see the topics beginning with Object Name and Title Data in HELP Reports . |
| SQL Name | VARCHAR(644) | No | |
| Uescape | VARCHAR(1) | Yes | |
| Dictionary Ordering Routine Database | VARCHAR(128) | Yes | |
| SQL Ordering | VARCHAR(644) | Yes | |

| Attribute | Data Type | Nullable? | Description |
|-----------------------------------|--------------|-----------|-------------|
| Routine Database | | | |
| Ordering Routine Database Uescape | VARCHAR(1) | Yes | |
| Dictionary Ordering Routine | VARCHAR(128) | Yes | |
| SQL Ordering Routine | VARCHAR(644) | Yes | |
| Ordering Routine Uescape | VARCHAR(1) | Yes | |
| UDT Database Dictionary Name | VARCHAR(128) | Yes | |
| UDT Database SQL Name | VARCHAR(644) | Yes | |
| UDT Database Name Uescape | VARCHAR(1) | Yes | |
| UDT Dictionary Name | VARCHAR(128) | Yes | |
| UDT SQL Name | VARCHAR(644) | Yes | |
| UDT Uescape | VARCHAR(1) | Yes | |

Attributes Reported by the ATTRIBUTE Option

The attributes reported by this option are generally identical to those reported by `HELP COLUMN` (see [HELP COLUMN](#) for the definitions of those attributes). The following table lists only those attributes that are unique to the `HELP TYPE ... ATTRIBUTE` option.

| Attribute | Data Type | Nullable? | Description |
|------------------------------|-----------------------------------|-----------|--|
| Attribute Name | VARCHAR(30) | Yes | <ul style="list-style-type: none"> If the UDT is a structured type, reports the name of one of its attributes. If the UDT is a distinct type, reports null. |
| Max Length | INTEGER formatted as -(10)9 | Yes | <ul style="list-style-type: none"> If the attribute is a UDT or ARRAY/VARRAY type, reports the maximum transformed length of its external data type. If no transform group is defined for the UDT, reports null. |
| Format | VARCHAR(30) | Yes | <ul style="list-style-type: none"> If the attribute is a UDT or ARRAY/VARRAY type, reports the format associated with its fromsql transform. If no fromsql transform is defined for the UDT, reports null. |
| UDT Name | VARCHAR(61) | Yes | <ul style="list-style-type: none"> If the attribute is a UDT or ARRAY/VARRAY type, reports its name. If the attribute is not a UDT or ARRAY/VARRAY type, reports null. |
| Decimal Total Digits | SMALLINT | Yes | <ul style="list-style-type: none"> If the attribute is a UDT or ARRAY/VARRAY type whose underlying data type is numeric, reports the total number of digits for the underlying type. If the attribute is not a UDT or ARRAY/VARRAY type, reports null. |
| Decimal Fractional Digits | SMALLINT | Yes | <ul style="list-style-type: none"> If the attribute is a UDT or ARRAY/VARRAY type whose underlying data type is numeric, reports the total number of fractional digits for the underlying type. If the attribute is not a UDT or ARRAY/VARRAY type, reports null. |
| Attribute Dictionary Name | VARCHAR(128) | No | <p>The attributes shown in this section are based upon and supersede the corresponding older attributes, Attribute Name and UDT Name, while providing additional functionality.</p> <p>The older attributes are retained for compatibility with existing applications.</p> <p>For details, see the topics beginning with Object Name and Title Data in HELP Reports.</p> |
| Attribute SQL Name | VARCHAR(644) | No | |
| Attribute Name Uescape | VARCHAR(1) | Yes | |
| UDT Database Dictionary Name | VARCHAR(128) | No | |

| Attribute | Data Type | Nullable? | Description |
|---------------------------|--------------|-----------|-------------|
| UDT Database SQL Name | VARCHAR(644) | No | |
| UDT Database Uescape Name | VARCHAR(1) | Yes | |
| UDT Dictionary Name | VARCHAR(128) | No | |
| UDT SQL Name | VARCHAR(644) | No | |
| UDT Name Uescape | VARCHAR(1) | Yes | |

Attributes Reported by the METHOD Option

The attributes reported by this option are different from those reported by HELP TYPE specified with no options or with the ATTRIBUTE option. The following table lists the options reported by HELP TYPE ... METHOD:

| Attribute | Data Type | Nullable? | Description |
|---------------|--------------|-----------|--|
| Method Name | VARCHAR(30) | No | The name used to invoke the associated method in an SQL request. |
| Specific Name | VARCHAR(30) | No | The Specific Name for the associated method. See "Method Name" earlier in this table for the character set specifications for SPECIFIC NAME. |
| Method Type | CHARACTER(2) | No | An encoded description of the associated method type. <ul style="list-style-type: none"> • C is a Constructor method. • I is an Instance method. • M is a Mutator method. • O is an Observer method. |
| Null Call | CHARACTER(1) | Yes | An encoded description of how the associated method deals with null parameters. This refers to the null call clause in the UDT definition (see CREATE TYPE (Distinct Form) and CREATE TYPE (Structured Form)). <ul style="list-style-type: none"> • N means do not invoke the method if an input parameter is null. Return a null to the requestor. • Y means always invoke the method. |

| Attribute | Data Type | Nullable? | Description |
|--------------------------|--------------|-----------|--|
| | | | Y corresponds to the CALLED ON NULL INPUT option. N corresponds to the RETURNS NULL ON NULL INPUT option. |
| Exec Mode | CHARACTER(2) | No | An encoded description of the execution mode for the associated method as defined by the ALTER METHOD statement (see ALTER METHOD). • NP means Non-protected. • P means Protected. NP corresponds to the EXECUTE NOT PROTECTED option. P corresponds to the EXECUTE PROTECTED option. |
| Deterministic | CHARACTER(1) | No | An encoded description of the deterministic characteristics clause for the associated method as defined by CREATE METHOD (see CREATE METHOD). • N means the method is not deterministic. • Y means the method is deterministic. Y corresponds to the DETERMINISTIC option. N corresponds to the NOT DETERMINISTIC option. |
| Method Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are based upon and supersede the corresponding older attributes, Method Name and Specific Name, while providing additional functionality. The older attributes are retained for compatibility with existing applications. For details, see the topics beginning with Object Name and Title Data in HELP Reports . |
| Method SQL Name | VARCHAR(644) | No | |
| Method Name Uescape | VARCHAR(1) | Yes | |
| Specific Dictionary Name | VARCHAR(128) | No | |
| Specific SQL Name | VARCHAR(644) | No | |
| Specific Name Uescape | VARCHAR(1) | Yes | |

HELP USER

HELP USER Attributes

The following table lists the attributes reported as the result of a HELP USER request.

Note:

HELP does not differentiate between user and database objects, so HELP USER and HELP DATABASE commands that specify the same object name return the same information.

| Attribute | Data Type | Nullable | Description |
|-----------------------|--------------|----------|--|
| Table/View/Macro name | VARCHAR(30) | No | The name of an object created in the user space. |
| Kind | CHARACTER | No | Returns the object type code for the object identified by Table/View/Macro name. See TVM Kind Codes . |
| Comment | VARCHAR(255) | Yes | The contents of the optional comment field for the user, if a comment was created. |
| Protection | CHARACTER | No | Indicates whether the user specifies default use of fallback protection. <ul style="list-style-type: none"> • F is Fallback. • N is None. |
| Creator Name | VARCHAR(30) | No | The name of the user that created the user specified in this HELP USER request. |
| Commit Option | CHARACTER | Yes | Indicates whether rows are preserved upon COMMIT of the transaction. <ul style="list-style-type: none"> • D represents Delete rows on COMMIT. • P represents Preserve rows on COMMIT. <p>Note: This attribute applies to temporary tables only.</p> |
| Transaction Log | CHARACTER | Yes | Indicates whether a transaction log is kept. <ul style="list-style-type: none"> • N means no log is kept. • Y means transactions are logged. <p>Note: This attribute applies to temporary tables only.</p> |
| Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are based upon and supersede the corresponding older attributes, Name and Creator Name, while providing additional functionality. The older attributes are retained for compatibility with existing applications. |
| SQL Name | VARCHAR(644) | | |
| UEScape | VARCHAR(1) | Yes | |

| Attribute | Data Type | Nullable | Description |
|-------------------------|--------------|----------|---|
| Creator Dictionary Name | VARCHAR(128) | No | For details, see the topics beginning with Object Name and Title Data in HELP Reports . |
| Creator SQL Name | VARCHAR(644) | | |
| Creator UEscape | VARCHAR(1) | Yes | |

HELP VIEW

View Attributes

The columns or parameters are listed in the order in which they were defined.

The attributes are returned as shown in the following table.

| Attribute | Data Type | Nullable? | Comments |
|----------------------|---------------|-----------|---|
| Column Name | VARCHAR(30) | No | <p>The name of a column in the view specified in the HELP VIEW request.</p> <p>Interpret as the equivalent of CF, or CHARACTER(n) data type.</p> <p>When DATEFORM=ANSIDATE, this column should be described externally as CHARACTER(10).</p> <p>Format characteristics</p> <ul style="list-style-type: none"> • With an explicit format, DATE. • With an implicit format created in ANSIDATE mode, 'YYYY/MM/DD'. <p>Note:</p> <p>The attributes listed in this table are returned for each column in the view.</p> |
| Type | CHARACTER(2) | Yes | <p>The data type for the Column Name.</p> <p>See Data Type Codes for a list of the data type codes and their meanings.</p> |
| Comment | VARCHAR (255) | Yes | <p>The contents of the optional comment field for the view, if a comment was created</p> |
| Nullable | CHARACTER | Yes | <p>Indicates whether the column name column is nullable.</p> <ul style="list-style-type: none"> • N specifies NOT NULL. • Y specifies nullable. |
| Format | CHARACTER(30) | Yes | <p>This column is always null for views.</p> |
| Title | VARCHAR (60) | Yes | <p>The title of the column, if a title exists.</p> |
| Max Length | INTEGER | Yes | <p>HELP VIEW requests do not return a Max Length value.</p> |
| Decimal Total Digits | SMALLINT | Yes | <p>If type is DECIMAL.</p> <p>For INTERVAL types, returns a value from 1–4 representing the number of digits in the leading field of the interval (number of non-fractional digits for INTERVAL SECOND).</p> <p>This column is null for TIME and TIMESTAMP data types.</p> |

| Attribute | Data Type | Nullable? | Comments |
|---------------------------|---------------|-----------|--|
| Decimal Fractional Digits | SMALLINT | Yes | This column is always null for views. |
| Range Low | FLOAT | Yes | Always null. BETWEEN clause is included in column constraint. |
| Range High | FLOAT | Yes | Always null. BETWEEN clause is included in column constraint. |
| Uppercase | CHARACTER | Yes | <p>If type is CHARACTER or VARCHAR, defines UPPERCASE and CASESPECIFIC attributes for the column.</p> <ul style="list-style-type: none"> • B specifies that column attributes for both UPPERCASE and CASESPECIFIC are defined. • C specifies that the column attributes for CASESPECIFIC is defined. • N specifies that column attributes for neither UPPERCASE nor CASESPECIFIC are defined. • U specifies that the column attributes for UPPERCASE is defined. |
| Table/View? | CHARACTER | No | Object type. HELP VIEW requests always return a value of V for view. |
| Default Value | VARCHAR (255) | Yes | <p>Returns the default value (as text) for the parameter.</p> <p>The value is null if there is no default value.</p> |
| Character Type | SMALLINT | Yes | <p>Returns the type of the different character data type columns.</p> <ul style="list-style-type: none"> • 1 specifies LATIN server character data type columns. • 2 specifies UNICODE server character data type columns. • 3 specifies KANJISJIS server character data type columns. • 4 specifies GRAPHIC server character data type columns. • 5 specifies KANJI1 server character data type columns. <p>If the data type for a column is not CHARACTER, this attribute returns null.</p> |
| IDCol Type | CHARACTER(2) | Yes | <p>Returns a code for the type of identity column.</p> <ul style="list-style-type: none"> • GA specifies Generated Always. • GD specifies Generated by Default. |
| UDT Name | VARCHAR(61) | Yes | This column is always null. |

| Attribute | Data Type | Nullable? | Comments |
|---------------------------------|---------------|-----------|---|
| Temporal Column | CHARACTER(2) | Yes | <p>Reports whether a column is temporal or not and if it is part of a Temporal Relationship constraint.</p> <ul style="list-style-type: none"> • N specifies that the column is non-temporal. • R specifies that the column is part of a temporal relationship constraint. • T specifies that the column has TRANSACTIONTIME column. • V specifies that the column has VALIDTIME column. <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Current ValidTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Current VALIDTIME column or not.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Sequenced ValidTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Sequenced VALIDTIME column or not.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| NonSequenced ValidTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Nonsequenced VALIDTIME column or not.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Current TransactionTime Unique? | CHARACTER(2) | Yes | <p>Reports whether a column is a UNIQUE Current TRANSACTIONTIME column or not.</p> <p>For information about temporal columns, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> |
| Column Dictionary Name | VARCHAR(128) | No | <p>The attributes shown in this section are based upon and supersede the corresponding older attributes, Column Name and Title, while providing additional functionality.</p> <p>The older attributes are retained for compatibility with existing applications.</p> <p>For details, see the topics beginning with Object Name and Title Data in HELP Reports.</p> <p>The system reports these attributes for each column in the view.</p> |
| Column SQL Name | VARCHAR(644) | No | |
| Column Uescape | VARCHAR(1) | Yes | |
| Dictionary Title | VARCHAR(256) | Yes | |
| SQL Title | VARCHAR(1256) | Yes | |

| Attribute | Data Type | Nullable? | Comments |
|---------------|------------|-----------|----------|
| Title Uescape | VARCHAR(1) | Yes | |

View Columns

If you perform `HELP VIEW` for a view or recursive view that has semantic errors in its definition, the system does not return an error message.

For details on defining views, see [CREATE VIEW](#) and [REPLACE VIEW](#).

HELP VOLATILE TABLE

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. Similarly, you can obtain information for the following volatile table attributes by performing a SHOW TABLE statement (see [SHOW object](#)).

- Protection
- Commit Option
- Transaction Log Option

| Attribute | Data Type | Nullable | Description |
|-----------------------|--------------|----------|---|
| Table Name | VARCHAR(30) | No | The name of the volatile table. |
| Table ID | CHAR(16) | No | The internal system-assigned identifier for the table. |
| Table Dictionary Name | VARCHAR(128) | No | The attributes shown in this section are based upon and supersede the corresponding older Table Name attribute, while providing additional functionality. The older attribute is retained for compatibility with existing applications. For details, see the topics beginning with Object Name and Title Data in HELP Reports . |
| Table SQL Name | VARCHAR(644) | No | |
| Table Uescape | VARCHAR(1) | Yes | |

HELP (Online Form)

Online HELP Compared with SQL HELP Statements

The online HELP statement facility is distinct from the SQL HELP statements.

- SQL HELP returns information on database objects and sessions. See the topics beginning with [About HELP Statements](#).
- Online HELP returns information about SQL statement and utility command syntax.

Notation for Online Help

The notation used to display SQL statement and utility command syntax is simple, as indicated by the following table:

| Notation Element | Description |
|------------------|--|
| [] | The item within the square brackets is optional. |
| { } | The item within the curly braces is required. |
| [...] | The item within the square brackets can be repeated. |
| | Exclusive OR. You can specify either the item to the right of or the item to its left, but not both. |

Stacked optional items indicate alternatives. For example, the following fragment indicates that you can choose only one of x, y, or z:

```
[x]
[y]
[z]
```

Sequential optional items indicate independent options. For example, the following fragment indicates that you can choose x, y, z, x and y, x and z, y and z, or x, y, and z:

```
[x] [y] [z]
```

Topic Keywords

Vantage provides online help for selected topical areas in addition to individual SQL statements. The online HELP topic keywords are the following.

- Archive
- BTEQ
- Dump

- FastExport
- FastLoad
- Help
- MultiLoad
- PMPC
- SPL (stored procedure control language statements)
- SQL
- TPCCONS

The topic and command name entries are not case-sensitive. For information, see the user documentation for the specific application.

Error on Request for Help on a Nonexistent Topic

The response for a HELP query on a nonexistent topic is the following:

```
Query completed. No rows found.
```

Response Format

The HELP response comprises a set of rows ordered by line number. The maximum length of each row of text is 80 characters. When accessed through BTEQ, no BTEQ formatting is needed to view the entire HELP message.

SQL SHOW Statements

Teradata SQL provides statements that summarize database object definition statement text.

For SHOW statement descriptions, syntax, and examples, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *SQL Quick Reference*, B035-1510.

About SHOW Statements

Using SHOW Statements

SHOW statements return the result of the last data definition statement performed against the named database object in the form of a CREATE *database_object* or ALTER *database_object* statement.

These statements are particularly useful for application developers who need to develop exact replicas of existing tables, views, triggers, hash and join indexes, stored procedures, user-defined functions, or macros for purposes of testing new software.

Some SHOW statements optionally return database object definition SQL text in XML format.

CHARACTER SET Support for SHOW Requests

Depending on the language support in use for the system, Vantage produces SHOW request reports using different character sets, as in the following table.

| Language Support | SHOW Request Output Server CHARACTER SET |
|---|--|
| Standard. | LATIN |
| Japanese with multibyte site-defined character set. | UNICODE |
| Anything else. | KANJI1 |

Unprintable Characters in Object Names and Literals Returned by SHOW

If an object name or literal returned by a SHOW request includes a character not in the repertoire of the session character set, or is otherwise unprintable, the name or literal is shown as a UNICODE delimited identifier or literal.

Each non-representable character is escaped to its hexadecimal equivalent.

The system uses BACKSLASH (U+005C), if it is available, as the escape character. If BACKSLASH is not available:

- If either YEN SIGN (U+00A5) or WON SIGN (U+20A9) replaces BACKSLASH in the session character set, the system uses it as the escape character.
- If none of the preceding characters is available, the system uses NUMBER SIGN (U+0023).

For example:

```
create table u&"#4E00" uescape'#' (i int);
show table u&"#4E00" uescape'#';
```

Object names that are translatable to the session character set, but are not lexically distinguishable as names are enclosed in double quotation marks.

Note:

For information on object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Using SHOW Output with Escape Characters in an SQL Statement

You can use SHOW output containing escape characters in another SQL statement. For example, using the SHOW output from the previous topic:

```
CREATE SET TABLE U&"\4E00" UESCAPE '\', NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO
(
    i INTEGER)
PRIMARY INDEX ( i );
```

SHOW request

Report Structure

The SHOW report lists the *DML_statement* DDL definitions in order of dependencies, first listing the definitions of independent objects and then listing dependent objects.

The rank ordering of the DDL definitions in the report is as follows:

1. Base tables
2. Referenced (parent) tables of base tables
3. Error tables
4. Views
5. Macros
6. Triggers
7. Join and hash indexes
8. UDTs

Restrictions

The following restrictions apply to SHOW:

- The statement cannot be used as part of a multistatement request. It can be issued only as a standalone request.
- The USING request modifier cannot precede a SHOW request.
- *DML_statement* is limited to the following SQL DML statements:
 - DELETE
 - INSERT
 - INSERT ... SELECT
 - MERGE
 - SELECT
 - UPDATE
- The objects referenced in *DML_statement* must have valid object names. For example, SHOW SELECT 1 is not a valid request and returns an error.
- If an object contains UDTs, the SHOW statement does not report the DDL for its casting, ordering, or transform functionality.

To view those DDL statements, you must use the SHOW TYPE statement (see [SHOW object](#)).

SHOW and Column Partitioning

See [SHOW TABLE](#), [SHOW JOIN INDEX](#), and [Column Partitioning](#) for information about how SHOW reports information about DML requests that access column-partitioned NoPI tables or join indexes.

SHOW and the IN XML Option

You can submit a SHOW request with the IN XML option to return the definitions of all the objects referenced by the specified DML request in XML format.

You cannot specify the IN XML option for the following database objects:

- CAST
- CONSTRAINT
- ERROR TABLE
- FUNCTION
- MACRO
- METHOD
- PROCEDURE
- SPECIFIC FUNCTION
- TRIGGER
- TYPE

The encoding of object names, SQL text, and literal character values is controlled by the default character for the session in which a SHOW IN XML request is executed.

Vantage sorts the objects in a topological order where objects in the lower order do not depend on objects in the higher order.

While Teradata provides the XML schemas required to interpret the XML files produced by the IN XML option, you must use a tool with a standard XML parser to interpret them.

Note:

Vantage returns the response to any SHOW IN XML request in a single result set in most cases. For multistatement requests that contain multiple SHOW IN XML statements, Vantage returns multiple result sets, with each result set containing one XML document that corresponds to the matching request.

Unlike the case for ordinary SHOW requests, you cannot submit the output of a SHOW IN XML request to Vantage to recreate the same database object.

SHOW *object*

SHOW and Embedded SQL

SHOW is a data returning statement. It returns a single row of a single column whose data type is VARCHAR(20480), and which contains new line characters embedded in the character string.

In the workstation-attached environment, SHOW returns carriage return characters in the string, which locate the cursor to the first position of the current line.

SHOW... INTO cannot be performed as a dynamic statement. Dynamic execution of data returning statements always requires a dynamic cursor.

SHOW CAST

SHOW CAST reports the user-submitted DDL request used to create a cast for the specified UDT.

SHOW CAST does not return the create text for any system-generated cast for a distinct type because no CREATE CAST request was executed to create it.

SHOW CONSTRAINT

The rules and restrictions for the SHOW CONSTRAINT statement are as follows.

- You must have either the CONSTRAINT DEFINITION privilege or the CONSTRAINT ASSIGNMENT privilege to execute this statement.
- The constraint you specify must currently exist in your system.

IN XML Option for SHOW Requests

You can export the definitions of tables or hash indexes from Vantage in XML format. Teradata provides the XML schemas required to interpret the XML files produced by this feature.

This enables the following applications.

- Displaying and altering table and hash index definitions using a GUI that need not be able to understand and write SQL DDL statements.
- Performing offline analyses of SQL workloads that require detailed definitions of the tables or hash indexes being referenced.
- Moving table and hash index definitions easily from one database version to another using a Teradata Tools and Utilities program like Teradata Data Mover that can parse XML object definitions and produce DDL requests to apply on the target database server, with the mapping of database features to database releases available on the Teradata client.

Teradata Tools and Utilities applications and third party tools can retrieve, display, and modify table and hash index definitions in XML format.

Teradata Tools and Utilities applications and third party tools can retrieve and display the XML formatted text for view and join index column definitions, but because the IN XML option does not report all of the definition constructs for views and join indexes, it is not possible to decompose and reconstruct their definitions from their XML format definitions.

The XML output for the SHOW IN XML VIEW and SHOW IN XML JOIN INDEX statements also contains a list in XML format of all the views and tables the view references, and all the tables the join index references.

The following SHOW *database_object_name* statements have the IN XML option, allowing you to produce reports in XML format in addition to their default SQL DDL code.

- SHOW HASH INDEX
- SHOW JOIN INDEX
- SHOW TABLE
- SHOW VIEW

You can also report the most recent create text for error tables in XML format if you use the SHOW IN XML TABLE *error_table_name* syntax.

Of these statements, only the XML output for SHOW IN XML HASH INDEX and SHOW IN XML TABLE statements contains complete information about the object definitions.

Because the output of statements using the IN XML option is in XML format, you must use a tool with a standard XML parser to interpret that output. Also be aware that unlike the non-XML forms of these SHOW statements, the statements that specify the IN XML option return their responses in a single result set for most cases. For multistatement requests that contain multiple SHOW IN XML statements, Vantage returns multiple result sets, with each result set containing one XML document that corresponds to the matching request.

Limitations in the Application of SHOW JOIN INDEX IN XML and SHOW VIEW IN XML

The IN XML option reports the SQL DDL, but not all of the definition constructs for join indexes and views, so it is not possible to decompose and reconstruct their definitions from their reported XML format definitions.

This means that you cannot use Teradata Tools and Utilities applications or third-party tools to retrieve join index or view definitions and their properties in XML format, nor can you export the definitions of those objects from the database in XML format.

Despite this, 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 or view definition in XML format.
You can then create global temporary or volatile tables to record the XML text view or join index definition data.
- A list of all of the referenced database objects in the join index or view definition.

This information is useful for validating that the base tables referenced in the join index or view definition are available, because Vantage only makes a semantic check when the object is referenced at run time.

Standard Form of Display for SHOW JOIN INDEX for Aggregate Join Indexes

By definition, the first column in the select list of an aggregate join index definition must be either a `COUNT(non_nullable_expression)` or `COUNT(*)` expression. When you create an aggregate join index that does not specify one of these COUNT expressions as the first column in its select list, then Vantage automatically moves the first such expression you specify to the first column in the definition. This also occurs whenever Vantage must add a `COUNT(*)` expression to the aggregate join index definition if you specify only a SUM operator.

This means that the column positions in the output of a SHOW JOIN INDEX request for an aggregate join index might not be identical to the column positions in the index definition that you originally typed.

SHOW MACRO, SHOW VIEW, and Semantic Definition Errors

If you perform SHOW MACRO or SHOW VIEW for a macro or view that has semantic errors in its definition, the system does not return an error message.

When the Actual User or Database Does Not Match the Definition Reported by SHOW MACRO or SHOW VIEW

When you copy or restore an archived macro or view definition to a different database or user than the one from which it had been archived, later SHOW MACRO or SHOW VIEW requests display the containing database or user for the *original* CREATE text for the macro or view stored in `DBC.TVM.RequestText`, not updated CREATE text that indicates the database or user into which the archived view or macro definition has been copied or restored.

This behavior is different from that for other SHOW statements, which generate the SQL text for the *current* definition of the specified object by building a parsing tree for the text from data in the table header and then working backward to return the SQL text for the current definition of the object. This reconstruction of definition text is not possible for macros and views because they do not have table headers from which current SQL definition text can be generated.

SHOW METHOD

This statement displays the create text for the SQL CREATE METHOD statement that created the body of the specified instance or constructor method.

The SHOW METHOD report does not show any method characteristics that are specified in the associated CREATE TYPE request create text. To view these characteristics, you must submit an appropriate SHOW TYPE request.

SHOW METHOD is not intended to be used to return the create text for any of the system-generated instance observer or mutator methods because no CREATE METHOD request was performed to create them. As a result, there is no create text that can be reported for system-generated methods.

If you submit a SHOW METHOD request on an observer or mutator method, the request aborts and the system returns an error to the requestor.

If a method signature was created as part of CREATE TYPE or ALTER TYPE statement, but a corresponding CREATE METHOD statement has not been submitted at the time you submit a SHOW METHOD request for it, the request aborts and the system returns an error to the requestor.

SHOW PROCEDURE For Java External Procedures

A SHOW PROCEDURE request displays the CREATE PROCEDURE or REPLACE PROCEDURE DDL text and the source code, if present. However, for Java external procedures, the source code is *not* present, and therefore cannot be shown. The JAVA LANGUAGE, JAVA PARAMETER STYLE and DATA ACCESS clauses are shown.

For example,

```
REPLACE PROCEDURE UT1.mysp (INOUT a INTEGER)
LANGUAGE JAVA
MODIFIES SQL DATA
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.mysp';
```

Standard Form of Display for SHOW TABLE

The following items describe the standard display form for SHOW TABLE results.

- SHOW TABLE always indicates whether the table was created as SET (no duplicate rows) or MULTISSET (duplicate rows if no uniqueness defined).
- SHOW TABLE displays fallback and journal status of the table.
- SHOW TABLE does not return the default FORMAT specifications for the columns in a table; it reports only those FORMAT specifications that differ from the defaults.

This means that SHOW TABLE returns an equivalent, but not necessarily identical, create text to that specified for the defining CREATE TABLE request.

- Named constraints become either table-level constraints or are mapped to unique indexes. The display reflects the implementation.

This means that regardless of the setting for DisableNoPI, when you declare an explicit PRIMARY KEY for a table, but do not declare either an explicit PRIMARY INDEX or NO PRIMARY INDEX, the

SHOW TABLE listing for that table does not report the specified primary key as a PRIMARY KEY constraint. Instead, Vantage reports what you had specified as the PRIMARY KEY as a UPI. Because *field_3* is neither declared as a PRIMARY KEY constraint nor as a UNIQUE constraint, the system does not change its definition in the SQL create text. See *Teradata Vantage™ - Database Design*, B035-1094 for the rules Vantage follows when making these conversions.

- If you do not define a MERGEBLOCKRATIO value for a table, a SHOW TABLE request returns the CREATE TABLE SQL text as if you had specified DEFAULT MERGEBLOCKRATIO, which is the default value for the option.
- SHOW TABLE displays character constant strings, including TITLE, COMPRESS, and DEFAULT values, as follows:

| FOR this type of constant... | The value is displayed in this format... |
|---------------------------------|--|
| printable single byte character | Teradata Latin. |
| anything else | internal Teradata hexadecimal. |

- The general rule for reporting column FORMAT attributes by SHOW TABLE is that a format string is only reported when it does not match the default format for the column.

The exception to this rule is columns that have the DATE data type. In this case, the system *always* reports the FORMAT string.

- If the table is a temporal table, the temporal columns are returned in the form AS VALIDTIME and AS TRANSACTIONTIME in the report.

SHOW TABLE If DATABLOCKSIZE or FREESPACE Has Been Modified

If the DATABLOCKSIZE or FREESPACE (or both) of the table has been modified using the ALTER TABLE or CREATE TABLE statement, then SHOW TABLE reports the currently defined data block size or percent freespace (or both).

If the data block size or freespace value is not listed, the default value is in effect.

Applications of SHOW TABLE IN XML and SHOW HASH INDEX IN XML

The IN XML option for SHOW TABLE and SHOW HASH INDEX enables Teradata Tools and Utilities applications and third-party tools to retrieve and display table and hash index definitions and their properties in XML format.

You can also export the definitions of tables or hash indexes from Vantage in XML format. This enables the following applications.

- Displaying and modifying table and hash index definitions using a GUI that does not need to be able to understand and write DDL statements.

- Offline analysis of SQL workloads that require detailed definitions of the database objects being referenced.
- Easily moving tables and hash indexes from one database version to another using a Teradata Tools and Utilities program like Teradata Data Mover that can parse XML object definitions and produce DDL statements to apply on the target database server, with the mapping of database features to database releases available on the Teradata client.

SHOW TABLE Support for Global Temporary and Volatile Tables

When you specify the keyword `TEMPORARY`, the table definition for the materialized global temporary table in the current session is displayed; else the definition for the base global temporary table is displayed.

The keyword `VOLATILE` is reported when the requested table is a volatile table.

SHOW TABLE, SHOW JOIN INDEX, and Column Partitioning

If the output for a `SHOW TABLE` or `SHOW JOIN INDEX` request includes a `PARTITION BY` clause, the partitioning follows. The output is the same for a column-partitioned table or join index except the `PARTITION BY` clause includes a `COLUMN` clause. Grouping, if any, is included in the `COLUMN` clause and not in the column definition list or select expression list. An `ADD` option is included if the level has column partitioning or if the number of defined partitions for a row partitioning level is less the maximum number of partitions for the level and it is not the first level that has row partitioning.

The following rules apply to the output of a `SHOW TABLE` or `SHOW JOIN INDEX` request for a column-partitioned object that has column grouping in a `COLUMN` clause.

- If all of the column partitions are single-column partitions with system-determined `COLUMN` or `ROW` format and without the `NO AUTO COMPRESS` option, Vantage does not return a column grouping following the `COLUMN` clause.
- If all of the column partitions are single-column partitions except for one that is multicolumn, and all of the column partitions have system-determined `COLUMN` or `ROW` format without an `NO AUTO COMPRESS` option, Vantage reports the shorter (in terms of its number of characters) of `COLUMN ALL BUT` or `COLUMN` following the `COLUMN` clause. For example,

```
COLUMN ALL BUT ((d, p, z))
```

```
COLUMN (a, b, c, g)
```

- If not the case documented by the second bullet in this list, and at least one, but not all, of the column partitions is a single-column partitions with system-determined `COLUMN` or `ROW` format without the `NO AUTO COMPRESS` option, Vantage reports `COLUMN ALL BUT` with any applicable options separated by COMMA characters 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 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 following the `COLUMN` clause.

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)
```

- If none of cases in the first 3 bullets apply, Vantage reports COLUMN with any applicable options, separated by COMMA characters 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 that specifies all of the column partitions following the COLUMN clause.

For example,

```
COLUMN (ROW d,(i,t), k NO AUTO COMPRESS, COLUMN(m,s,u,v))
```

SHOW TABLE and Language Support Mode of a System

You cannot submit a SHOW TABLE request against a dictionary table to determine the language support mode for your system because Vantage always returns the value *Unicode* as the response to such a request.

Instead, you must submit the following SELECT request to determine the language support mode used by your system.

```
SELECT infodata
FROM DBC.dbcinfoV
WHERE infokey='language support mode';
```

| IF your system uses this language support mode ... | THEN the InfoKey value the system returns is ... |
|--|--|
| standard | Standard |
| Japanese | Japanese |

SHOW TYPE

This statement reports all the user-submitted DDL statements used to create the specified type and define its behavior except CREATE CAST, not just the create text for the specified UDT. To report CREATE CAST DDL, you must use the SHOW CAST statement.

For ARRAY and VARRAY types, SHOW TYPE reports the most recent SQL create text for the ARRAY type in standard Teradata syntax, using the ARRAY keyword. The ordering and transform functionality for the ARRAY type are not reported, because it is system-generated and there is no create text to return.

The report includes information about the following related DDL statements for the type and its supporting semantics including the create text for the following definitions:

- CREATE TYPE
- CREATE ORDERING
- CREATE TRANSFORM

SHOW TYPE does not return the create text for any of the system-generated orderings, or transforms for a distinct type because no CREATE ORDERING or CREATE TRANSFORM statements were performed to create them. As a result, there is no create text that can be reported for system-generated casts, orderings, or transforms.

Because the specification of user-defined casts is optional, there might not be a cast definition for a given UDT. In this case, the system returns a “No rows returned” message to the requestor indicating that no casts have been defined for the specified UDT.

SHOW QUERY LOGGING

Alternative Method for Determining Query Logging Rules

You can also determine the query logging rules stored in *DBC.DBQLRuleTbl* by submitting the following SELECT request against the *DBQLRulesV* view:

```
SELECT *
FROM DBC.DBQLRulesV;
```

Rules Hierarchy for SHOW QUERY LOGGING

When you submit a SHOW QUERY LOGGING request, it 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 specific user and specific account. |
| 3 | A rule for this specific user and any account. |
| 4 | A rule for all users and this specific account. |
| 5 | A rule for all users 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 that exist in the rules cache and in *DBC.LogRuleTbl*. Note that these criteria handle 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 name and account name | the matching rule from the following set, searching in the order indicated: <ol style="list-style-type: none"> 1. specified user name:specified account name 2. specified user name:ALL account names 3. ALL user names:specified account name 4. ALL user names:ALL account names |
| a single user name, but no account name | the matching rule from the following set, searching in the order indicated: <ol style="list-style-type: none"> 1. specified user name:ALL account names 2. ALL user names:ALL account names |

| IF you specify ... | THEN the request searches for the following single best fit rule ... |
|---------------------------------|--|
| ALL and a specific account name | the matching rule from the following set, searching in the order indicated: 1. ALL user names:specified account name 2. ALL user names:ALL account names |
| ALL, but no account name | the matching ALL user names:ALL account names rule. |

SHOW STATISTICS

Applicability of SHOW STATISTICS Requests

You can use SHOW STATISTICS requests to report information on both Optimizer statistics and QCD statistics.

Rules and Guidelines for SHOW STATISTICS Requests

The following rules and guidelines apply to SHOW STATISTICS requests.

- Vantage 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, Vantage 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, Vantage reports detailed statistics for all of the columns in the database object.
- When you specify the SUMMARY option, Vantage 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 Vantage 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.
- Vantage 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 Vantage operations irrespective of the session time zone.

About the SEQUENCED Option

The output of the SHOW STATISTICS VALUES option is a spool with a set of output request rows. If the table-level SHOW STATISTICS VALUES, or if you specify multiple column or index sets, the response can contain output from multiple statistics. The SEQUENCED option, which is only valid if you also specify the

VALUES option, enables you to report sequence numbers for each such sequence of statistics data in a spool. The database uses the sequence number to identify the boundary of the each statistics output.

For example, assume you are interested in reported detailed statistics for 3 statistics on a table, and the individual detailed statistics have the following sizes.

| Statistics Group | Size (KB) |
|------------------|-----------|
| 1 | 150 |
| 2 | 30 |
| 3 | 100 |

The database inserts the output of the 3 statistics groups into a spool along with sequence numbers that can be used to maintain the correct order of the output request text. The spool for the output is chunked into sequences up to 64 KB each. The database inserts up to 64 KB of the output request text into the first sequence, and whatever remains into subsequent sequences.

The final output with sequence numbers for this example is as follows. Because statistics output 2 is only 30 KB, it all fits into a single sequence.

| Statistics Output | Sequence Number | Size (KB) |
|-------------------|-----------------|-----------|
| 1 | 1 | 64 |
| | 2 | 64 |
| | 3 | 22 |
| 2 | 1 | 30 |
| 3 | 1 | 64 |
| | 2 | 36 |

The following examples demonstrate different uses of the SEQUENCED option.

The following SHOW STATISTICS request displays all the available detailed statistics for *table_1* with sequence numbers.

```
SHOW STATISTICS VALUES SEQUENCED ON table_1;
```

The following SHOW STATISTICS request displays the detailed statistics on column *x1* from *table_1* with sequence numbers.

```
SHOW STATISTICS VALUES SEQUENCED COLUMN x1 ON table_1;
```

The following SHOW STATISTICS request displays the detailed statistics on columns *x1* and *y1* from *table_1* with sequence numbers.

```
SHOW STATISTICS VALUES SEQUENCED COLUMN x1, COLUMN y1 ON table_1;
```

The following SHOW STATISTICS request displays all of the available detailed statistics for *table_1* with sequence numbers in XML format.

```
SHOW IN XML STATISTICS VALUES SEQUENCED ON table_1;
```

Rules, Restrictions, and Usage Guidelines for the SEQUENCED Option

- You can specify the SEQUENCED option for both the Optimizer and QCD forms of SHOW STATISTICS.
- You can only specify SEQUENCED if you also specify VALUES.
- Client utilities can use sequence numbers to identify the boundaries of each statistics output.

Response Sequences for Detailed Statistics (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:

| Attribute | Description | Data Type |
|--|---|--------------|
| This set of information is reported once across all the returned histogram intervals. | | |
| 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 database software on your server. | VARCHAR(32) |
| SamplePct | If the statistics were sampled, the sampling percentage used to collect them. | DECIMAL(5,2) |

| Attribute | Description | Data Type |
|----------------------------|--|--|
| 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. | |
| 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. Vantage 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. Vantage 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, Vantage converts it to UNICODE. • If the minimum value is a non-case specific CHARACTER or VARCHAR type, Vantage converts it to UPPERCASE. • Vantage might truncate a minimum value if its size is greater than MAXVALUELENGTH. |

| Attribute | Description | Data Type |
|--------------------------|--|--|
| Mode Value | The data value that occurs the most frequently in a histogram interval. Vantage 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, Vantage converts it to UNICODE. • If the modal value is a non-case specific CHARACTER or VARCHAR type, Vantage converts it to UPPERCASE. • Vantage might truncate a modal value if its size is greater than MAXVALUELENGTH. |
| Max Value | The maximum data value in a histogram interval. Vantage 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, Vantage converts it to UNICODE. • If the maximum value is a non-case specific CHARACTER or VARCHAR type, Vantage converts it to UPPERCASE. • Vantage might 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. | Stored as the native type for the data value of the first biased value in the histogram with the following exceptions. <ul style="list-style-type: none"> • If the first biased value is a non-LATIN CHARACTER or VARCHAR type, Vantage converts it to UNICODE. • If the first biased value is a non-case specific CHARACTER or VARCHAR type, Vantage converts it to UPPERCASE. |

| Attribute | Description | Data Type |
|------------------------------------|---|--|
| | | <ul style="list-style-type: none"> Vantage might truncate a biased value if its size is greater than MAXVALUELENGTH. |
| Biased Freq1 | The frequency of occurrence of the first biased value. | REAL |
| ... | | |
| Biased Value n | The data value of the n^{th} biased value in the histogram. | <p>Stored as the native type for the data value of the n^{th} biased value in the histogram with the following exceptions.</p> <ul style="list-style-type: none"> If the n^{th} biased value is a non-LATIN CHARACTER or VARCHAR type, Vantage converts it to UNICODE. If the n^{th} biased value is a non-case specific CHARACTER or VARCHAR type, Vantage converts it to UPPERCASE. Vantage might truncate a biased value if its size is greater than MAXVALUELENGTH. |
| Biased Freq n | The frequency of occurrence of the n^{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. | |
| Max Value | <p>The maximum data value in the first equal height interval in the histogram.</p> <p>Vantage returns a separate column for each reported column value.</p> | <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, Vantage converts it to UNICODE. If the maximum value is a non-case specific CHARACTER or VARCHAR type, Vantage converts it to UPPERCASE. Vantage might truncate a maximum value if its size is greater than MAXVALUELENGTH. |
| Mode Value | <p>The modal data value in the first equal height interval in the histogram.</p> <p>Vantage returns a separate column for each reported column value.</p> | <p>Stored as the native type for the column with the following exceptions.</p> <ul style="list-style-type: none"> If the modal value is a non-LATIN CHARACTER or VARCHAR type, Vantage converts it to UNICODE. If the modal value is a non-case specific CHARACTER or VARCHAR type, Vantage converts it to UPPERCASE. Vantage might truncate a modal value if its size is greater than MAXVALUELENGTH. |

| Attribute | Description | Data Type |
|----------------------------------|---|--|
| 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. The Optimizer uses this value to determine the deviation of the values in an equal-height interval compared with its modal and lowest frequencies. | 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 |
| Equal-Height Interval [2] | The information that follows this heading is taken from the second equal-height interval in the histogram being reported. | |
| Max Value | The maximum data value in the second equal height interval in the histogram. Vantage 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, Vantage converts it to UNICODE. • If the maximum value is a non-case specific CHARACTER or VARCHAR type, Vantage converts it to UPPERCASE. • Vantage might 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. Vantage 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, Vantage converts it to UNICODE. • If the modal value is a non-case specific CHARACTER or VARCHAR type, Vantage converts it to UPPERCASE. • Vantage might 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 |

| Attribute | Description | Data Type |
|---------------------|--|-----------|
| 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 |

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 for information about Record and Indicator modes.

Locks and Concurrency

Vantage places rowhash-level READ or ACCESS locks on *DBC.StatsTbl* to retrieve statistics information.

Detailed Interval Statistics Can Change From One Release To The Next

Detailed QCD interval statistics represent an internal structure that is subject to change from one release to the next in terms of content, structure, number of intervals, or format (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for more information). Because of this, you should avoid including such SHOW STATISTICS requests in applications that cannot easily be modified.

The same is true for detailed Optimizer interval statistics, which are used internally by the Optimizer. Optimizer statistics represent an internal structure that is subject to change from one release to the next in terms of content, structure, number of intervals, or format (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for more information). Avoid including SHOW STATISTICS requests for detailed Optimizer statistics in applications that you cannot modify easily.

Detailed Interval Statistics Attributes (Field Mode)

When a request for a detailed SHOW STATISTICS report is made in Field Mode, Vantage returns the result as multiple rows (one per interval, including the master record) consisting of multiple columns. The statistics values retain the underlying data type of the column on which they were collected.

Response Sequences for Detailed Statistics (Record 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

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 |