Teradata
a division of NCR

Teradata® RDBMS

SQL Reference - Volume 4
Data Definition Statements

V2R5.0

B035-1101-122A
May 2003

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

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

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

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

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

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

or write:

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

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

Copyright © 1996-2002, NCR Corporation
All Rights Reserved

Preface

Supported Software Release

This book supports Teradata RDBMS V2R5.0.

Changes to This Book

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

Date	Description
May 2003	• Corrected syntax diagrams for CREATE JOIN INDEX. • Added note that ROWID can be specified only for CREATE JOIN INDEX statement and then only in the outermost SELECT statement in the join index definition.
December 2002	• Added new chapter for query and workload analysis SQL statements. With the exception of the optimizer forms of COLLECT STATISTICS and DROP STATISTICS, the statements in this chapter are all new to V2R5.0 • Added ALTER PROCEDURE statement. • Added SET SESSION DATABASE statement. • Extended ALTER TABLE for partitioned primary index, batch referential integrity, referential constraints, and multivalued compression. • Extended CREATE TABLE for partitioned primary index, multivalued compression, batch referential integrity, referential constraints, and identity column. • Moved the following statements from *Teradata RDBMS SQL Reference, Volume 6* into this book: – DUMP EXPLAIN – INSERT EXPLAIN • Added new statements for database query logging: – BEGIN QUERY LOGGING – END QUERY LOGGING • Added new statements to support Teradata Index and Statistics wizards: – COLLECT DEMOGRAPHICS – COLLECT STATISTICS (QCD Form) – DROP STATISTICS (QCD Form) – INITIATE INDEX ANALYSIS – RESTART INDEX ANALYSIS

Date	Description
December 2002 (continued)	• Added new statements for roles: – CREATE ROLE – DROP ROLE – SET ROLE – GRANT (Role Form) • REVOKE (Role Form)Added new statements for profiles: – CREATE PROFILE – DROP PROFILE – MODIFY PROFILE • Extended GRANT (SQL Form), REVOKE (SQL Form), COMMENT, CREATE USER, MODIFY USER, BEGIN LOGGING, and END LOGGING for roles and profiles • Extended HELP SESSION for new data formatting attributes • Removed references to DefaultDateFormat, which is no longer in DBSControl • Moved former Chapter 4 to *Teradata RDBMS SQL Reference, Volume 6* • Performed DR fixes and enhancement requests
June 2001	• Added new CREATE/DROP HASH INDEX statements • Extended CREATE JOIN INDEX statement for direct base table access and single-table join index compression • Revised limits on number of volatile and global temporary tables per session • Extended SHOW statement • Performed DR fixes and enhancement requests
September 2000	• Usability enhancements
June 2000	• General reorganization and rewriting of material to enhance usability • Addition and extensive rewriting and reorganization of embedded SQL material formerly found in *Teradata RDBMS Application Programming with C, COBOL and PL/I* • New stored procedure DDL and development tool statements: — CREATE PROCEDURE — DROP PROCEDURE — HELP PROCEDURE — RENAME PROCEDURE — REPLACE PROCEDURE — SHOW PROCEDURE • New SHOW DML statement • Extended CREATE JOIN INDEX syntax supports aggregates • Extended CREATE TABLE syntax supports copying table structure definitions and table data to a new table • Performed DR fixes and enhancement requests

About This Book

Purpose

Teradata RDBMS SQL Reference, Volume 4 describes the Teradata Structured Query Language (SQL) and how to use SQL to do any of the following things:

- Define or restructure the database (Data Definition Language)
- Assign or revoke access to the data (Data Control Language)

This preface describes the organization of *Teradata RDBMS SQL Reference, Volume 4* and identifies information you should know before using it. This book should be used in conjunction with the other volumes of *Teradata RDBMS SQL Reference*.

Audience

System administrators, database administrators, and security administrators are the principal audience for this manual. NCR field engineers and other technical personnel responsible for designing and maintaining the Teradata RDBMS may also find this manual useful.

How This Book Is Organized

This book has four chapters and one appendix.

- Chapter 1: "SQL Data Definition Language Statement Syntax," lists the SQL Data Definition Language statements alphabetically and provides full syntax and usage notes for each individual statement.
- Chapter 2: "SQL Data Control Language Statement Syntax," lists the SQL Data Control Language statements alphabetically and provides full syntax and usage notes for each individual statement.
- Chapter 3: "Query and Workload Analysis Statements," lists the SQL statements that collect statistical, demographic, and query logging information for the Optimizer and for various client utilities to analyze.

  The chapter also includes a set of diagnostic statements for use with the Teradata Index Wizard and with the target level emulation facility.
- Chapter 4: "SQL Help and Database Object Definition Tools: HELP and SHOW," provides syntax and usage notes for the EXPLAIN modifier, the various HELP statements, and the various SHOW statements. These statements supply a wealth of information that can be used by database designers and software architects to optimize their Teradata databases for maximal use with minimal cost.
- Appendix A: "Notation Conventions," explains how to read syntax diagrams, character shorthand notation, and predicate calculus notation.

Prerequisites

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

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

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

List of Acronyms

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

AMP	Access Module Processor vproc
ANSI	American National Standards Institute
BTEQ	Basic TEradata Query facility
BYNET	Banyan Network - High speed interconnect
CJK	Chinese, Japanese, and Korean
CLIv2	Call Level Interface Version 2
cs0, cs1, cs2, cs3	Four code sets (codeset 0, 1, 2, and 3) used in EUC encoding.
E2I	External-to-Internal
EUC	Extended UNIX Code
FK	Foreign Key
HI	Hash Index
I2E	Internal-to-External
JI	Join Index
JIS	Japanese Industrial Standards
LT/ST	Large Table/Small Table (join)
NPPI	Nonpartitioned Primary Index
NUPI	Non-Unique Primary Index
NUSI	Non-Unique Secondary Index
OLTP	On-Line Transaction Processing
PDE	Parallel Database Extensions
PE	Parsing Engine vproc
PI	Primary Index
PK	Primary Key
PPI	Partitioned Primary Index
RDBMS	Relational Database Management System
RI	Referential Integrity
SDF	Specification for Data Formatting
SQL	Structured Query Language

SQO	Semantic Query Optimization
UCS-2	Universal Coded Character Set containing 2 bytes
UPI	Unique Primary Index
USI	Unique Secondary Index
vproc	Virtual Process

Technical Information on the Web

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

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

Contents

Preface

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

Chapter 1:
SQL Data Definition Language Statement Syntax

ALTER PROCEDURE 1–3
ALTER TABLE 1–12
ALTER TRIGGER 1–65
BEGIN LOGGING 1–67
COMMENT 1–76
CREATE DATABASE 1–80
CREATE HASH INDEX 1–86
CREATE INDEX 1–100
CREATE JOIN INDEX 1–108
CREATE MACRO
REPLACE MACRO 1–142
CREATE PROCEDURE
REPLACE PROCEDURE 1–154
CREATE PROFILE 1–173
CREATE ROLE 1–180
CREATE TABLE 1–182
CREATE TABLE (Table Kind Clause) 1–201
CREATE TABLE (Table Options Clause) 1–208
CREATE TABLE (Column Definition Clause) 1–212
CREATE TABLE (Index Definition Clause) 1–239
CREATE TABLE (Temporary/Volatile Table Preservation Clause) 1–248
CREATE TABLE (AS Clause) 1–249
CREATE TABLE (Examples) 1–257
CREATE TRIGGER
REPLACE TRIGGER 1–276
CREATE USER 1–296
CREATE VIEW
REPLACE VIEW 1–311

DATABASE 1–325
DELETE DATABASE
DELETE USER 1–327
DROP DATABASE 1–330
DROP HASH INDEX 1–332
DROP INDEX 1–333
DROP JOIN INDEX 1–337
DROP MACRO
DROP PROCEDURE
DROP TABLE
DROP TRIGGER
DROP VIEW 1–338
DROP PROFILE 1–341
DROP ROLE 1–343
DROP USER 1–345
END LOGGING 1–347
MODIFY DATABASE 1–352
MODIFY PROFILE 1–359
MODIFY USER 1–364
RENAME MACRO
RENAME PROCEDURE
RENAME TABLE
RENAME TRIGGER
RENAME VIEW 1–380
SET ROLE 1–382
SET SESSION 1–384
SET SESSION ACCOUNT 1–387
SET SESSION COLLATION 1–391
SET SESSION DATABASE 1–397
SET SESSION DATEFORM 1–398
SET TIME ZONE 1–400

Chapter 2:
SQL Data Control Language Statement Syntax

Access Privileges 2–2
Owners, Creators, and Users 2–4
Privilege Types 2–5
GIVE 2–11
GRANT 2–14
GRANT (Monitor Form) 2–15
GRANT (Role Form) 2–19
GRANT (SQL Form) 2–22
GRANT LOGON 2–43

REVOKE 2–47
REVOKE (MONITOR Form) 2–49
REVOKE (Role Form) 2–54
REVOKE (SQL Form) 2–57
REVOKE LOGON 2–70

Chapter 3:
Query and Workload Analysis Statements

Query and Workload Analysis Statements 3–3
BEGIN QUERY LOGGING 3–4
COLLECT DEMOGRAPHICS 3–11
COLLECT STATISTICS (Optimizer Form) 3–14
COLLECT STATISTICS (QCD Form) 3–27
DROP STATISTICS (Optimizer Form) 3–31
DROP STATISTICS (QCD Form) 3–37
DUMP EXPLAIN 3–40
END QUERY LOGGING 3–44
INITIATE INDEX ANALYSIS 3–46
INSERT EXPLAIN 3–55
RESTART INDEX ANALYSIS 3–62
Diagnostic Tools for Target Level Emulation and the Teradata Index Wizard 3–65
DIAGNOSTIC DUMP COSTS 3–66
DIAGNOSTIC HELP COSTS 3–68
DIAGNOSTIC SET COSTS 3–71
DIAGNOSTIC DUMP SAMPLES 3–76
DIAGNOSTIC HELP SAMPLES 3–78
DIAGNOSTIC SET SAMPLES 3–80
DIAGNOSTIC "Validate Index" 3–83

Chapter 4:
SQL Help and Database Object Definition Tools: HELP and SHOW

HELP 4–3
HELP COLUMN 4–7
HELP CONSTRAINT 4–22
HELP DATABASE
HELP USER 4–26
HELP HASH INDEX 4–29
HELP INDEX 4–31

HELP JOIN INDEX 4–38
HELP MACRO
HELP TABLE
HELP VIEW 4–40
HELP PROCEDURE 4–46
HELP SESSION 4–52
HELP STATISTICS (Optimizer Form) 4–60
HELP STATISTICS (QCD Form) 4–63
HELP TRIGGER 4–65
HELP VOLATILE TABLE 4–68
HELP (Online Form) 4–70
SHOW 4–73
SHOW HASH INDEX
SHOW JOIN INDEX
SHOW MACRO
SHOW PROCEDURE
SHOW TABLE
SHOW TRIGGER
SHOW VIEW 4–79

Appendix A:
Notation Conventions

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

Index Index-1

Chapter 1:

SQL Data Definition Language Statement Syntax

This chapter documents Teradata SQL data definition language (DDL) statements. DDL statements define, revise, and remove database objects.

Experienced SQL users can also see the simplified DDL statement descriptions in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL/Data Dictionary Quick Reference*.

Teradata data control language (DCL) statements are described in Chapter 2: "SQL Data Control Language Statement Syntax."

Query and workload analysis statements are described in Chapter 3: "Query and Workload Analysis Statements." The two forms of the COLLECT STATISTICS and DROP STATISTICS statements are now documented in Chapter 3.

Teradata data manipulation language (DML) statements are described in the chapter "SQL Data Manipulation Language Syntax" in *Teradata RDBMS SQL Reference, Volume 6*.

This chapter describes the following statements:

- "ALTER PROCEDURE" on page 1-3
- "ALTER TABLE" on page 1-12
- "ALTER TRIGGER" on page 1-65
- "BEGIN LOGGING" on page 1-67
- "COMMENT" on page 1-76
- "CREATE DATABASE" on page 1-80
- "CREATE HASH INDEX" on page 1-86
- "CREATE INDEX" on page 1-100
- "CREATE JOIN INDEX" on page 1-108
- "CREATE MACRO REPLACE MACRO" on page 1-142
- "CREATE PROCEDURE REPLACE PROCEDURE" on page 1-154
- "CREATE PROFILE" on page 1-173
- "CREATE ROLE" on page 1-180

- "CREATE TABLE" on page 1-182
  - "CREATE TABLE (Table Kind Clause)" on page 1-201
  - "CREATE TABLE (Table Options Clause)" on page 1-208
  - "CREATE TABLE (Column Definition Clause)" on page 1-212
  - "CREATE TABLE (Index Definition Clause)" on page 1-239
  - "CREATE TABLE (Temporary/Volatile Table Preservation Clause)" on page 1-248
  - "CREATE TABLE (AS Clause)" on page 1-249
  - "CREATE TABLE (Examples)" on page 1-257
- "CREATE TRIGGER REPLACE TRIGGER" on page 1-276
- "CREATE USER" on page 1-296
- "CREATE VIEW REPLACE VIEW" on page 1-311
- "DATABASE" on page 1-325
- "DELETE DATABASE DELETE USER" on page 1-327
- "DROP DATABASE" on page 1-330
- "DROP HASH INDEX" on page 1-332
- "DROP INDEX" on page 1-333
- "DROP JOIN INDEX" on page 1-337
- "DROP MACRO DROP PROCEDURE DROP TABLE DROP TRIGGER DROP VIEW" on page 1-338
- "DROP PROFILE" on page 1-341
- "DROP ROLE" on page 1-343
- "DROP USER" on page 1-345
- "END LOGGING" on page 1-347
- "MODIFY DATABASE" on page 1-352
- "MODIFY PROFILE" on page 1-359
- "MODIFY USER" on page 1-364
- "RENAME MACRO RENAME PROCEDURE RENAME TABLE RENAME TRIGGER RENAME VIEW" on page 1-380
- "SET ROLE" on page 1-382
- "SET SESSION" on page 1-384
- "SET SESSION ACCOUNT" on page 1-387
- "SET SESSION COLLATION" on page 1-391
- "SET SESSION DATABASE" on page 1-397
- "SET SESSION DATEFORM" on page 1-398
- "SET TIME ZONE" on page 1-400

ALTER PROCEDURE

Purpose

Recompiles an existing stored procedure in the database and allows changes in the following compile-time attributes of the procedure:

- SPL option
- Warnings option

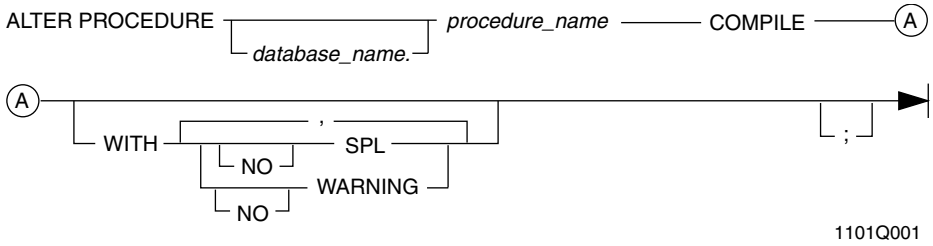
Invocation

Executable.

Stored procedures only.

Not valid inside a stored procedure.

Syntax



where:

Syntax Element …	Specifies …
database_name	the name of a qualifying database for the stored procedure to be altered. If *dbname* is not specified, the default database for the current session is assumed.
procedure_name	the name of the stored procedure to be recompiled. You can recompile only one stored procedure at a time.

Syntax Element …	Specifies …
Alter Procedure Options	
The options are stored in the data dictionary with the stored procedure definition. You can list the options in any order. Each option can be specified at most once.	
SPL NO SPL	that the source text (DDL definition) of the stored procedure should or should not be stored in Teradata. SPL is the default option.
WARNING NO WARNING	that the compilation warnings are returned or suppressed during alteration of the stored procedure. WARNING is the default option.

ANSI Compliance

ALTER PROCEDURE is a Teradata extension to ANSI SQL-99.

Authorization

To alter a stored procedure, you must have either ALTER PROCEDURE or DROP PROCEDURE privilege on that procedure or on the database containing the procedure.

Usage Notes

The ALTER PROCEDURE feature is useful in recreating in Teradata release V2R5.0 the stored procedures from Teradata release V2R4.1 or V2R4.2 without having to perform SHOW PROCEDURE and REPLACE PROCEDURE statements.

Recompilation in this way allows changing of some of the compile-time attributes of a stored procedure.

Recompilation is also useful for cross-platform archive and restoration of stored procedures.

Limitations

You cannot use ALTER PROCEDURE to change the DDL definition of a stored procedure, that is, to REPLACE the procedure.

You cannot use ALTER PROCEDURE to recompile stored procedures created in Teradata release V2R4.0 because information about the default database at the time of the stored procedure creation is not available.

Changes Made by ALTER PROCEDURE

ALTER PROCEDURE can alter the following attributes of the recompiled stored procedure:

- Platform
- TDSP version number.

  If a stored procedure created in Teradata release V2R4.1 is recompiled in release V2R5.0, the version number becomes 03.

  For information about stored procedure version numbers, see "HELP PROCEDURE" on page 4-46.

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

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

ALTER PROCEDURE also does not change the following attributes of the stored procedure being recompiled:

- Session Mode
- Creator character set
- Creator character type
- Default Database
- The access privileges defined for the stored procedure

ALTER PROCEDURE Rules

- The stored procedure specified in an ALTER PROCEDURE statement must exist.
- The stored procedure specified in an ALTER PROCEDURE statement must have been created with the SPL compile-time option. That is, the stored procedure source text must be available for the stored procedure being recompiled.
- If a stored procedure is recompiled by an ALTER PROCEDURE statement with NO SPL option, the source text of the stored procedure is deleted from Teradata and the stored procedure cannot be recompiled again.
- You can alter a stored procedure in the same session mode in which the procedure was originally created. Stored procedures created in ANSI mode cannot be recompiled in Teradata mode and vice versa.
- An ALTER PROCEDURE statement cannot be used inside a stored procedure body.

- If you do not specify any compile-time options with an ALTER PROCEDURE statement, the options with which the stored procedure was previously created, replaced, or recompiled apply. If some compile-time options are specified with an ALTER PROCEDURE statement, the unspecified options default to the existing options of the stored procedure.
- A COMMIT must be specified after an ALTER PROCEDURE statement in ANSI mode transactions, as in the case of all DDL statements.
- In Teradata mode, ALTER PROCEDURE must be the last statement in a transaction.
- All the statements other than DML, DDL, and DCL statements within the stored procedure are validated syntactically and semantically during the execution of an ALTER PROCEDURE statement.

  For DML, DDL, and DCL statements within the stored procedure body, the validation does not include the following:
  - access privileges checks
  - data type compatibility
- If the execution of an ALTER PROCEDURE statement fails, the existing version of the stored procedure is retained.
- Errors or warnings generated during the recompilation of a stored 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.

Suppression of Warnings

You can suppress all the warnings generated during recompilation using the NO WARNING option. If you do not specify this option, all the compilation warnings are returned in the response parcel.

This option is not available at present with CREATE PROCEDURE and REPLACE PROCEDURE statements.

Locks and Concurrency

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

Cross-Platform Recompilation

ALTER PROCEDURE allows recompilation of stored procedures on a new platform after a cross-platform archive and restore of the database containing the stored procedures. Normally, stored procedures cannot be used across platforms.

Example

Assume that the database Alterdb containing the stored procedure spAP1 and other objects exists on a Windows system, and you want to perform the stored procedure on a new platform, NCR UNIX MP-RAS, for example.

To do this, you need to perform the following:

Stage	Action
1	Take a dump of the database Alterdb from the Windows system.
2	Log on to a UNIX MP-RAS system and copy the data dump there.

3	Perform the SHOW statement to see the source text of the stored procedure:
	`SHOW PROCEDURE spAP1;`
	The following response is returned:
	`CREATE PROCEDURE spAP1 (in par1 INTEGER, out par2 INTEGER)`
	`BEGIN`
	`  SET par2 = par1 * par1;`
	`END;`

4	Perform the HELP PROCEDURE statement to see the platform and other attributes of the stored procedure.
	`HELP PROCEDURE spAP1 ATTR;`
	The Help information returned is as follows:
	`Transaction Semantics TERADATA  Print Mode N`
	`Platform Windows NT  Character Set UTF8`
	`Default Character DataType LATIN  Collation ASCII`
	`SPL Text Y  Version Number 03`
	`Default Database Alterdb  Warning Option Y`

5	Specify the CALL statement to perform the stored procedure spAP1 on the new platform (NCR UNIX MP-RAS). The request fails with the following message because the object was created in Windows. `*** Failure 5530 Invalid operating environment for procedure execution.`
6	Perform ALTER PROCEDURE to recompile the stored procedure: `ALTER PROCEDURE spAP1 COMPILE;`

After the recompilation is completed, you can use HELP PROCEDURE to verify that the Platform attribute has changed to UNIX MP-RAS.

Then you can successfully perform the stored procedure on the new platform by specifying the CALL statement.

```
CALL spAP1(10, par2);
```

Using ALTER PROCEDURE: Examples

The first two example make reference to stored procedure `spAP2`, shown below, to illustrate the use of ALTER PROCEDURE.

```
CREATE PROCEDURE spAP2(IN InParam INTEGER,
                       OUT OutParam INTEGER)
BEGIN
   DECLARE Var1 INTEGER DEFAULT 10;
   SET OutParam = InParam + Var1 + 1;
END;
```

The examples assume that the stored procedure has been compiled as follows:

```
BTEQ> .COMPILE FILE = testsp.spAP2
```

The SPL option is not specified, but the source text of the stored procedure `spAP2` is stored in Teradata because SPL is the default.

Example 1

This example illustrates the use of ALTER PROCEDURE to recompile a stored procedure in the Teradata release V2R5.0 with changed options. The example shows that the original character set of a stored procedure is not changed by ALTER PROCEDURE.

Step	Action

1	Verify the current attributes of the procedure using HELP PROCEDURE.
	`HELP PROCEDURE spAP2 ATTR;`
	The Help information returned is as follows:
	`Transaction Semantics TERADATA  Print Mode Y`
	`Platform UNIX MP-RAS  Character Set EBCDIC`
	`Default Character DataType UNICODE  Collation ASCII`
	`SPL Text Y  Version Number 01`
	`Default Database testsp  Warning Option Y`
	The Version Number 01 indicates that the stored procedure was created in a Teradata V2R4.1 release.

2	Change the character set of the session from EBCDIC to ASCII using BTEQ: `.SET SESSION charset 'ASCII'`
3	Perform ALTER PROCEDURE with NO SPL and NO WARNING options. `ALTER PROCEDURE spAP2 COMPILE WITH NO SPL, NO WARNING;`

Step	Action

4	Check the procedure attributes again, after completion of the alteration.
	`Transaction Semantics TERADATA  Print Mode N`
	`Platform UNIX MP-RAS  Character Set EBCDIC`
	`Default Character DataType UNICODE  Collation ASCII`
	`SPL Text N  Version Number 03`
	`Default Database testsp  Warning Option N`
	Note that the compile-time options are changed as specified, but the character set of the procedure has not changed. Version Number 03 indicates that the stored procedure has been upgraded to Teradata release V2R5.0.

Example 2: Failure Case

In this example, ALTER PROCEDURE is performed on a stored procedure originally compiled with NO SPL option. ALTER PROCEDURE fails because the source text of the stored procedure is not available for recompiling. The procedure was originally compiled with NO SPL option; as a result, the source text has not been stored in the database.

Step	Action
1	Compile the stored procedure spAP2. `.COMPILE FILE spAP2.spl WITH NOSPL`

2	Verify its attributes.
	`HELP PROCEDURE spAP2 ATTR;`
	The Help information returned is as follows:
	`Transaction Semantics TERADATA  Print Mode N`
	`Platform UNIX MP-RAS  Character Set ASCII`
	`Default Character DataType UNICODE  Collation ASCII`
	`SPL Text N  Version Number 03`
	`Default Database testsp  Warning Option Y`

3	Then perform the ALTER PROCEDURE statement.
	`ALTER PROCEDURE spAP2 COMPILE;`
	The output for the ALTER PROCEDURE statement is as follows:
	`ALTER PROCEDURE spAP2 COMPILE;`
	`*** Failure 5535 No SPL source text available for stored procedure 'spAP2'.`
	Since the execution of the ALTER PROCEDURE statement has failed, the existing version of the stored procedure spAP2 is retained.

Example 3

This example shows how compilation warnings can be suppressed using the NO WARNING option of the ALTER PROCEDURE statement.

Step	Action

1	Create a new stored procedure spAP3.
	`REPLACE PROCEDURE testsp.spAP3 ()`
	`BEGIN`
	`  DECLARE var1 INTEGER DEFAULT 10;`
	
	`  SELECT ErrorCode INTO :var1 FROM dbc.errormsgs`
	`  WHERE ErrorCode = 5526;`
	`  SET var1 = var1 + 1;`
	`END;`

2	Compile the stored procedure spAP3:
	`BTEQ> .COMPILE FILE spAP3.spl`
	When you do, you get the following output:
	`*** Procedure has been created 4 Errors/Warnings.`
	`*** Warning: 5527 Stored Procedure Created with Warnings.`
	`*** Total elapsed time was 2 seconds.`
	
	`.COMPILE FILE spAP3.spl`
	`$`
	`*** SQL Warning 5802 Statement is not ANSI.`
	`Warnings reported during compilation`
	`----------`
	`SPL5000:W(L1), W(5802):Statement is not ANSI.`
	`SPL5000:W(L3), W(5802):Statement is not ANSI.`
	`SPL5000:W(L5), W(5802):Statement is not ANSI.`
	`SPL5000:W(L6), W(5802):Statement is not ANSI.`
	`----------`

3	Recompile the procedure with NO WARNING option. `ALTER PROCEDURE spAP3 COMPILE WITH NO WARNING;` When this statement is successfully performed, the procedure spAP3 is recompiled, but the compilation warnings are not displayed.

Example 4: Session Mode Impact

This example illustrates the behavior of ALTER PROCEDURE when the session mode is changed.

Step	Action
1	Compile the stored procedure spAP3 in a Teradata mode session. `BTEQ> .COMPILE FILE spAP3.spl`

2	Change the session mode.
	`.LOGOFF`
	`.SET SESSION TRANS ANSI`
	`.LOGON testsp, password`

3	Perform ALTER PROCEDURE.
	`ALTER PROCEDURE ap3 COMPILE;`
	The statement fails with the following message:
	`*** Failure 5510 Invalid session mode for procedure execution.`
	`Statement# 1, Info =0`
	Since the execution of ALTER PROCEDURE failed, the existing version of the stored procedure spAP3 is retained.

Related Information

See the following chapter and statements for related information:

- "SQL Stored Procedures" in *Teradata RDBMS SQL Reference, Volume 6*
- "CREATE PROCEDURE REPLACE PROCEDURE" on page 1-154

ALTER TABLE

Purpose

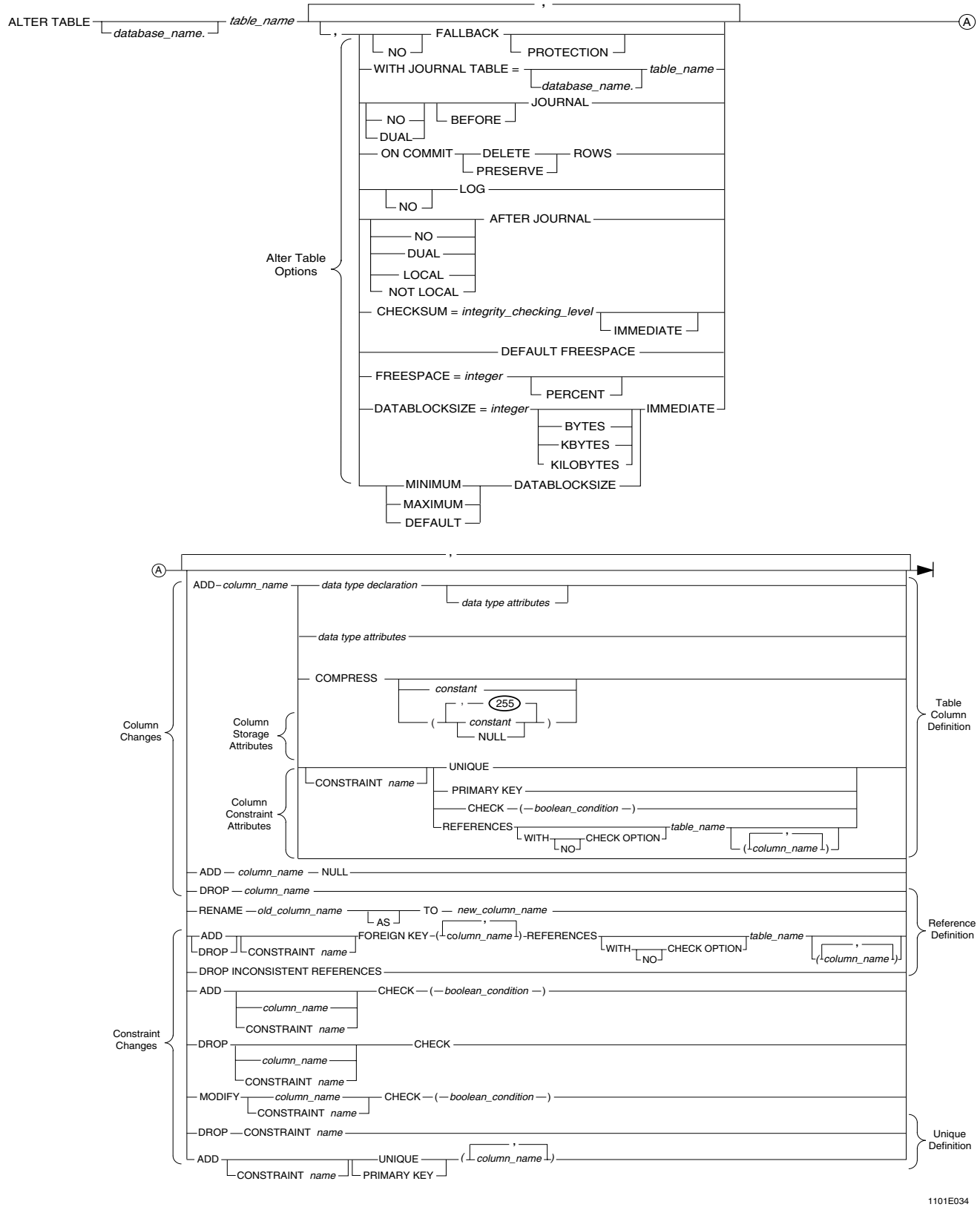
Performs all the following functions:

- Adds new attributes for one or more columns of a table or global temporary table definition.
- Adds one or more new columns to an existing table or global temporary table definition.
- Drops one or more columns from a table or global temporary table definition.
- Modifies column-level and table-level constraints.
- Modifies referential constraints.
- Requests or removes the FALLBACK option for a table.
- Requests or modifies the JOURNAL option for a table.
- Changes the DATABLOCKSIZE or percent FREESPACE for a table or global temporary table definition.
- Changes the LOG and ON COMMIT options for a global temporary table.
- Changes the name of a column.
- Modifies the properties of the primary index for a table.
- Modifies the partitioning properties of the primary index for a table, including modifications to the partitioning expression defined for use by a partitioned primary index.
- Regenerates table headers and optionally validates and corrects the partitioning of PPI table rows.

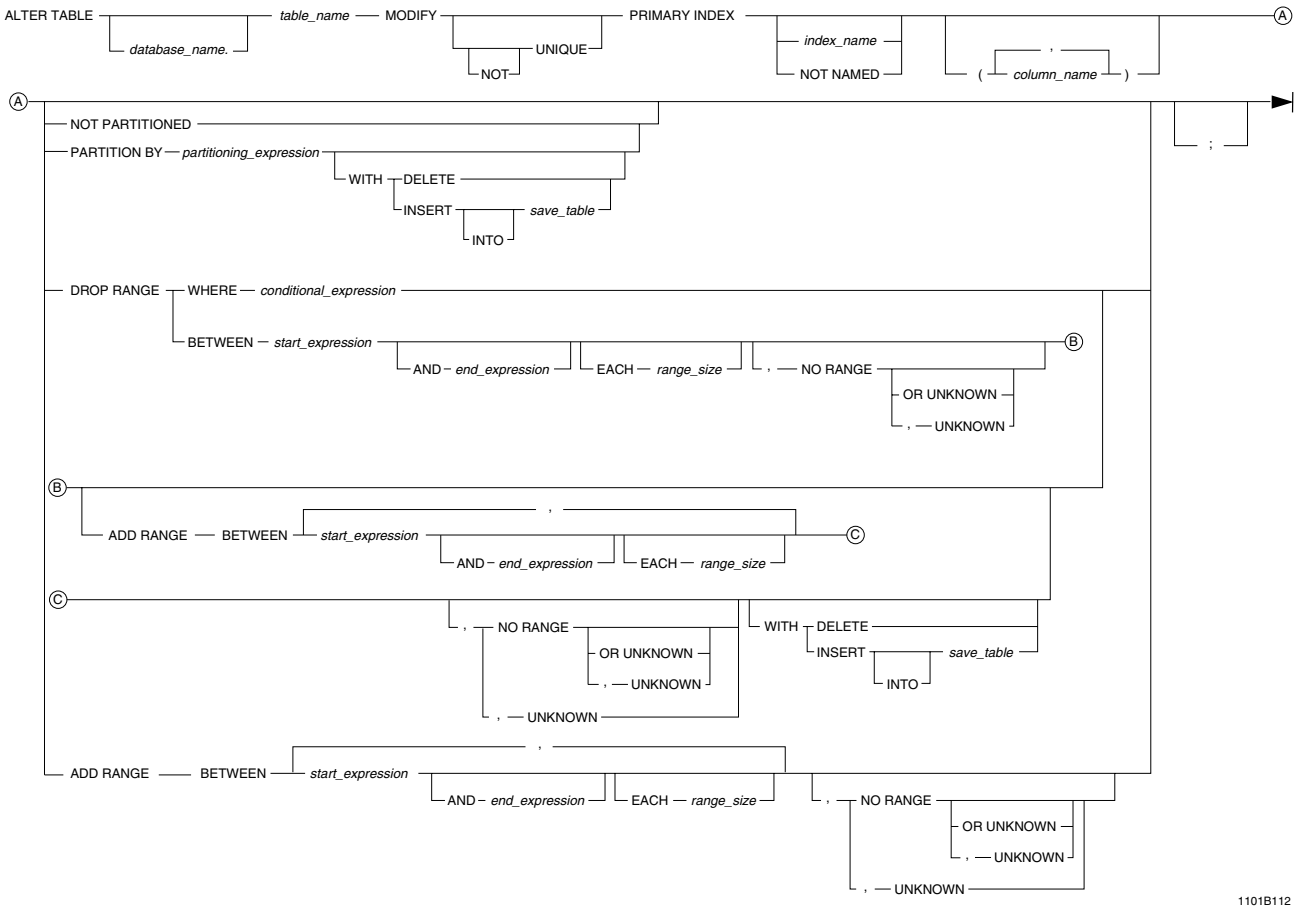
Invocation

Executable.

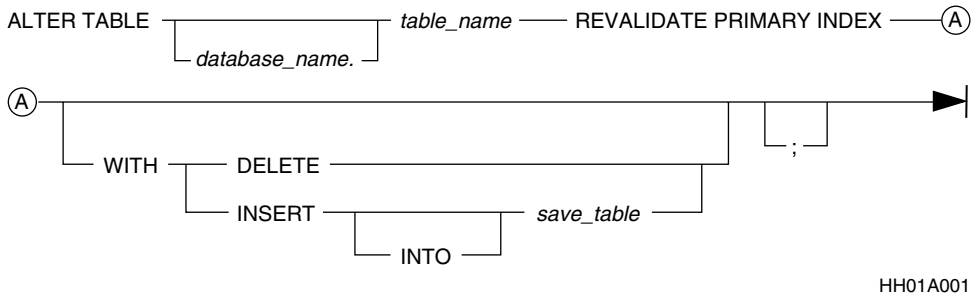
Basic Table Parameters Modification Syntax



Primary Index Modification Syntax



Partitioned Primary Index Revalidation Syntax



where:

Syntax Element …	Specifies …
database_name. *table_name*	the name of the table to be altered. If *database_name* is not specified, the default database for the current session is assumed.

Alter Table Options

The Alter Table options are a Teradata extension to ANSI SQL. The options can be listed in any order.

You cannot modify or revalidate a primary index in the same ALTER TABLE statement you perform to modify basic table parameters.

FALLBACK PROTECTION NO FALLBACK PROTECTION	to add or remove duplicate copy protection for the table. If fallback is being added, a duplicate copy of the table is created and stored. If fallback is being removed, the existing duplicate copy is deleted. The FALLBACK keyword used alone implies PROTECTION.
WITH JOURNAL TABLE = *database_name*. *table_name*	a definition for the journal table for the data table being altered. The journal table *table_name* need not reside in the same database as the data table. If a database is not specified, the default database for the current session is assumed. If a database is specified, it must exist and *table_name* must have been defined as its default journal table.
JOURNAL NO BEFORE JOURNAL DUAL BEFORE JOURNAL	the number of before change images to be maintained for the table. If the JOURNAL keyword is specified without NO or DUAL, then a single copy of the image is maintained unless FALLBACK is in effect or is also specified. If journaling is requested for a table that uses fallback protection, DUAL images are maintained automatically.
ON COMMIT DELETE ROWS ON COMMIT PRESERVE ROWS	the action to take with the contents of a global temporary table when a transaction ends. DELETE ROWS clears the temporary table of all rows. PRESERVE ROWS retains the rows in the table after the transaction is committed. You cannot change the ON COMMIT properties of a global temporary table if any materialized instance of the table exists anywhere in Teradata.

Syntax Element …	Specifies …
LOG NO LOG	the transaction journaling option for a global temporary table. LOG maintains a transaction journal for the temporary table during the time is materialized; NO LOG specifies that no transaction journal is to be kept. You cannot change the LOG/NO LOG properties of a global temporary table if any materialized instances of the table exist anywhere in Teradata.
NO AFTER JOURNAL DUAL AFTER JOURNAL LOCAL AFTER JOURNAL NOT LOCAL AFTER JOURNAL	the type of image to be maintained for the table; any existing images are not affected until the table is updated. The NO and DUAL options specify the number of after-change images to be maintained for the table. NOT LOCAL and LOCAL specify whether single after-image journal rows for non-fallback data tables are written on the same virtual AMP (LOCAL) as the changed data rows, or on another virtual AMP in the cluster (NOT LOCAL).
BEFORE JOURNAL AFTER JOURNAL	the type of image to be maintained for the table. The default for this option is established by a CREATE DATABASE, CREATE USER, or MODIFY USER statement for the database in which the table is to be created. See "CREATE TABLE" on page 1-182 for a listing of the journaling that results from the available journal and image type specifications.
DEFAULT FREESPACE	to reset the value for FREESPACE PERCENT to the system default defined for this table by DBSControl. This specification does not have an immediate effect on existing cylinders, but the new attribute value controls any new allocation of cylinder blocks using this revised free space percentage.
FREESPACE = *integer* PERCENT	a new percent of free space to remain on a cylinder after a loading operation completes. If the value specified does not fall within the valid range (0 - 75), then an error is returned. See "FREESPACE" on page 1-211 for additional information. This specification does not have an immediate effect on existing cylinders, but the new attribute value controls any new allocation of cylinder blocks using this revised free space percentage.

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td colspan="2">Data Block Size Option
You cannot modify or revalidate a primary index in the same ALTER TABLE statement you perform to modify basic table parameters.</td></tr>
<tr><td>DATABLOCKSIZE= integer BYTES/KBYTES/ KILOBYTES</td><td>the maximum data block size for blocks that contain multiple rows as the value integer. The calculation is made in the same manner as the CREATE TABLE statement.

The maximum data block size value is 128 Kbytes.

<table><tr><th>The default value for this type of data block ...</th><th>Is specified using this field of the DBSControl record ...</th></tr><tr><td>permanent</td><td>PermDBSize</td></tr><tr><td>transient and permanent journal</td><td>JournalDBSize</td></tr></table>
See Teradata RDBMS Utilities, Volume 1 A-K for more detailed information about setting default data block sizes.

For details about data block size, see “DATABLOCKSIZE” on page 1-209.</td></tr>
<tr><td>IMMEDIATE</td><td>whether to repackage data blocks into the newly specified size immediately or to wait until an affected block is accessed.

If the IMMEDIATE option is included, the attribute value will still be changed, but, in addition, the rows in all existing data blocks are repacked into data blocks of the specified size.

If the IMMEDIATE option is not included, existing data blocks are left unmodified. In both cases, the new attribute value will determine the maximum size of multiple-row data blocks subsequently modified or allocated.

See “IMMEDIATE DATABLOCKSIZE” on page 1-27.</td></tr>
<tr><td>MINIMUM DATABLOCKSIZE</td><td>the smallest possible DATABLOCKSIZE setting for the table.

MINIMUM DATABLOCKSIZE sets the maximum data block size for blocks that contain multiple rows to the minimum legal value of either 6144 or 7168 bytes (12 or 14 sectors), depending on the cylinder size set for your system.

See “DATABLOCKSIZE” on page 1-209.</td></tr>
<tr><td>MAXIMUM DATABLOCKSIZE</td><td>the largest possible DATABLOCKSIZE setting for the table.

MAXIMUM DATABLOCKSIZE sets the maximum data block size for blocks that contain multiple rows to the maximum legal value of 130 560 bytes (255 sectors).</td></tr>
</table>

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td>DEFAULT DATABLOCKSIZE</td><td>the default DATABLOCKSIZE setting for the table.

<table>
<tr><th>The default value for this type of data block ...</th><th></th><th>Is specified using this field of the DBSControl record ...</th></tr>
<tr><td>permanent</td><td></td><td>PermDBSize</td></tr>
<tr><td>transient and permanent journal</td><td></td><td>JournalDBSize</td></tr>
</table>

See Teradata RDBMS Utilities for more detailed information about setting default data block sizes.

For details about data block size, see “DATABLOCKSIZE” on page 1-209.</td></tr>
<tr><td colspan="2">Column Change Options

You cannot modify or revalidate a primary index in the same ALTER TABLE statement you perform to modify basic table parameters.</td></tr>
<tr><td>ADD column_name

data_type_declaration

data_type_attributes

column_storage_attributes

column_constraint_attributes</td><td>to add the specified column and its specified attributes.

ADD and DROP cannot both be specified on the same column in the same ALTER TABLE statement.

The ADD keyword either changes the definition of an existing column or adds a new column to the table. If the named column already exists, ADD indicates that its attributes are to be changed.

column_name specifies the name of a column whose attributes are to be changed or that is to be added or dropped.

If you do not specify explicit formatting, the new column assumes the default format for the data type, which can be specified by a custom data formatting specification (SDF) defined by the tdlocaledef utility (see Teradata RDBMS Utilities). Explicit formatting applies both to the parsing and to the retrieval of character strings.

Data types and data type attributes are the subject of Teradata RDBMS SQL Reference, Volume 3. Column storage and constraints attributes are listed in “Column and Table Constraints” in the chapter “SQL Data Definition” in that volume.

You cannot alter the compression specification for an existing column.</td></tr>
</table>

Syntax Element …	Specifies …
ADD *column_name* NULL	to change an existing column *column_name* from being NOT NULL to being nullable. You cannot add an identity column to an existing table, nor can you add the identity column attribute to an existing column. This option is a Teradata extension to the ANSI standard.
DROP *column_name*	that the named column is to be removed from the table. You can drop an identity column from an existing table, but you cannot drop just the identity column attribute and retain the column.
RENAME *old_column_name*	the name of a column that is to be renamed as *new_column_name*.
[AS] TO *new_column_name*	the new name for the column named in *old_column_name*.

Constraint Change Options

You cannot modify or revalidate a primary index in the same ALTER TABLE statement you perform to modify basic table parameters.

ADD/DROP CONSTRAINT *name* FOREIGN KEY (*column_name_list*) REFERENCES *table_name column_name*	to add or drop a named constraint.
WITH CHECK OPTION	that the constraint added or dropped is a standard referential integrity constraint for which the integrity of the relationship is checked only when the entire batch of actions of which it is a component completes. If any of the rows in the batch is found to violate the Referential Integrity rule, then the entire batch is rolled back. This clause is a Teradata extension to the ANSI SQL-99 standard.
WITH NO CHECK OPTION	that the constraint added or dropped is not to be used to validate referential integrity. This clause can only be specified with a table-level FOREIGN KEY … REFERENCES … constraint or a column-level REFERENCES … constraint. This clause is a Teradata extension to the ANSI SQL-99 standard.
DROP INCONSISTENT REFERENCES	to delete all inconsistent references defined on the table. DROP INCONSISTENT REFERENCES is commonly used after a restore, when an ALTER TABLE DROP FOREIGN KEY might not work. This clause is a Teradata extension to the ANSI SQL-99 standard.

Syntax Element …	Specifies …
ADD CHECK (*boolean_condition*) ADD *column_name* CHECK (*boolean_condition*) ADD CONSTRAINT *name* CHECK (*boolean_condition*)	to add or modify a constraint condition. When a constraint is added or modified, all existing rows of the table are accessed to validate that the current values conform to the specified search condition. If not, an error is returned and no change is made to the table definition. When a CHECK constraint is added, the table is scanned at the time the constraint is altered, and the ALTER TABLE statement is aborted if there are existing rows that violate the new constraint. This clause is a Teradata extension to the ANSI SQL-99 standard.
DROP CHECK DROP *column_name* CHECK DROP CONSTRAINT *name* CHECK	to drop all unnamed column-level CHECK constraints on *column_name*. DROP CHECK drops the unnamed table-level constraints on the table. DROP CONSTRAINT *name* CHECK or DROP CONSTRAINT *name* drop a named table-level CHECK constraint on the table.
MODIFY *column_name* CHECK (*boolean_condition*), MODIFY CONSTRAINT *name* CHECK (*boolean_condition*)	to modify a check constraint, whether named or unnamed. MODIFY CONSTRAINT *name* CHECK *(boolean_condition)* is valid only if a constraint with *name* already exists in the table. This also includes the named constraints defined as part of the column definition since those constraints are handled as named table-level constraints. For example, MODIFY *column_name* is allowed only if *column_name* already has a defined constraint. Existing rows of the table are accessed to validate that the current values conform to the specified search condition. If not, an error is returned and no change is made to the table definition. When a CHECK constraint is modified, the table is scanned at the time the constraint is altered and the ALTER TABLE statement is aborted if there are existing rows that violate the new constraint.
DROP CONSTRAINT *name*	to drop a table-level named constraint on the stated table.
ADD *unique_definition*	to add a uniqueness constraint on one or more columns of the table. The internal implementation is to add a unique secondary index. The columns listed for uniqueness must all be defined to be NOT NULL. Supported uniqueness constraints are the following: • UNIQUE • PRIMARY KEY Existing rows are accessed to validate that the proposed uniqueness constraint is not violated. If uniqueness is violated, an error is reported and no change is made. Teradata implicitly creates a unique secondary index on any column set specified as UNIQUE or PRIMARY KEY.

Syntax Element …	Specifies …
MODIFY PRIMARY INDEX Option	
You cannot modify basic table parameters or revalidate a primary index in the same ALTER TABLE statement you perform to modify a primary index. See "General Rules for the Primary Index Options Clause" on page 1-42 and "General Rules for the Modify Primary Index Option" on page 1-43.	
MODIFY [[NOT] UNIQUE] PRIMARY INDEX (*column_name*)	to change the uniqueness property of the primary index for the table. To change a nonunique primary index to a unique primary index, specify MODIFY UNIQUE PRIMARY INDEX. To change a unique primary index to a nonunique primary index, specify MODIFY NOT UNIQUE PRIMARY INDEX.
index_name NOT NAMED	to change whether the primary index for the table is named or unnamed or to change the name of a named primary index. To name an unnamed primary index or change the existing name of a primary index to something else, specify MODIFY … PRIMARY INDEX *index_name*. To drop the name of a named index, specify MODIFY … PRIMARY INDEX NOT NAMED.
NOT PARTITIONED PARTITION BY *partitioning_expression*	to change whether the primary index for the table is partitioned or not or to change the partitioning expression for an existing partitioned primary index. To change a partitioned primary index to a nonpartitioned primary index, specify NOT PARTITIONED. To change a nonpartitioned primary index to a partitioned primary index, or to change the partitioning expression for an existing partitioned primary index, specify PARTITION BY and then specify a valid partitioning expression (see "Rules for Altering the Partitioning Expression for a Primary Index" on page 1-46).
WITH DELETE	to delete any row whose partition number evaluates to a value outside the valid range of 1 - 65 535, inclusive.
WITH INSERT [INTO] *save_table*	to insert any row whose partition number evaluates to a value outside the valid range of 1 - 65 535, inclusive, into the table named by *save_table*. *save_table* and the table being altered must be distinct.

Syntax Element …	Specifies …
DROP RANGE WHERE *conditional_expression*	to use a conditional partitioning expression to drop a range set from the RANGE_N function on which the partitioning expression for the table is based. You can only drop ranges if the partitioning expression for the table is derived only from a RANGE_N function. You must base *conditional_partitioning_expression* on the system-derived PARTITION column. See "RANGE_N" in *Teradata RDBMS SQL Reference, Volume 5*, for details about the RANGE function.
DROP RANGE BETWEEN … [NO RANGE [OR UNKNOWN]]	to drop a set of ranges from the RANGE_N function on which the partitioning expression for the table is based. You can also drop NO RANGE OR UNKNOWN and UNKNOWN specifications from the definition for the RANGE_N function. You can only drop ranges if the partitioning expression for the table is derived exclusively from a RANGE_N function. Ranges must be specified in ascending order. See "RANGE_N" in *Teradata RDBMS SQL Reference, Volume 5*, for details about the RANGE function.
ADD RANGE BETWEEN … [NO RANGE [OR UNKNOWN]]	to add a set of ranges to the RANGE_N function on which the partitioning expression for the table is based. You can also add NO RANGE OR UNKNOWN and UNKNOWN specifications to the definition for the RANGE_N function. You can only add ranges if the partitioning expression for the table is derived exclusively from a RANGE_N function. Ranges must be specified in ascending order. See "RANGE_N" in *Teradata RDBMS SQL Reference, Volume 5*, for details about the RANGE function.
WITH DELETE	to delete any row whose partition number evaluates to a value outside the valid range of 1 - 65 535, inclusive.
WITH INSERT [INTO] *save_table*	to insert any row whose partition number evaluates to a value outside the valid range of 1 - 65 535, inclusive, into the table named by *save_table*. *save_table* and the table being altered must be distinct.

Syntax Element …	Specifies …
REVALIDATE PRIMARY INDEX Option	
You cannot modify either the basic table parameters or the primary index in the same ALTER TABLE statement you perform to revalidate a primary index. See "General Rules for the Primary Index Options Clause" on page 1-42 and "Rules for Altering the Partitioning Expression for a Primary Index" on page 1-46.	
REVALIDATE PRIMARY INDEX	to regenerate the table headers for a specified table with a partitioned primary index. You can optionally verify row partitioning and correct any errors that are found during the verification process.
WITH DELETE	to delete any row encountered during revalidation whose partition number evaluates to a value outside the valid range of 1 - 65 535, inclusive.
WITH INSERT [INTO] *save_table*	to insert any row encountered during revalidation whose partition number evaluates to a value outside the valid range of 1 - 65 535, inclusive, into the table named by *save_table*. *save_table* and the table being altered must be distinct. You cannot insert rows with nonvalid partitions into a regular Teradata base table.

ANSI Compliance

ALTER TABLE is ANSI SQL-99-compliant with extensions.

Authorization

To alter a table, you must have DROP TABLE privilege on that table or on the database containing the table.

Rules for 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.

FALLBACK and JOURNAL Defaults

The FALLBACK and JOURNAL defaults for a data table are originally determined by the statement that created or modified the object in which the table resides (see "CREATE TABLE" on page 1-182, "CREATE TRIGGER REPLACE TRIGGER" on page 1-276, "CREATE DATABASE" on page 1-80, and "MODIFY USER" on page 1-364).

Either or both of these defaults can be overridden by the statement that created the data table (see "CREATE TABLE" on page 1-182) or by the ALTER TABLE statement.

ALTER TABLE Not Valid for Volatile Tables

You cannot use ALTER TABLE to change the definition for a volatile table. Instead, you must DROP the old table and CREATE a new one to replace it.

Locks and Concurrency

When an ALTER TABLE statement is performed, an EXCLUSIVE lock is placed on the table being altered.

Because of the overhead required to alter a table for dropped or added columns, you should establish a final form for a table before it contains many rows. When a very large table is altered, it can be unavailable to other users for a long time.

ALTER TABLE and FALLBACK

When an ALTER TABLE statement adds FALLBACK, a full-table scan is generated. To add FALLBACK to a table, all AMPs must be online and operational.

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 Teradata.

However, when an instance of a global temporary table is materialized, Teradata 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.

Dropping Columns

When a column is dropped, Teradata must delete the field corresponding to the dropped column in every row in the table.

Indexed columns cannot be dropped without first dropping the index on those columns. If an attempt is made to drop a column on which a primary, secondary, hash, or join index is defined, an error message is returned.

Columns referenced in the UPDATE OF clause of a trigger cannot be dropped.

You can drop an identity column from an existing table, but you cannot drop just the identity column attribute and retain the column.

You cannot drop a partitioning column for a PPI without first modifying the partitioning to exclude that column from the partitioning set. If you attempt to drop a partitioning column, Teradata returns an error.

You cannot change the data type, nullability, or name of a partitioning column for a PPI.

You cannot change the data type, nullability, or name of an indexed column.

The following procedure explains what you must do to drop a column on which an index is defined.

<table>
<tr><th>To drop a column on which this type of index is defined ...</th><th>Follow this procedure ...</th></tr>
<tr><td>Primary</td><td>
<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Create a new table with the correct primary index.</td></tr>
<tr><td>2</td><td>Copy the data into the new table.
A simple way to do this is to use an INSERT … SELECT statement.</td></tr>
<tr><td>3</td><td>Drop the original table.</td></tr>
<tr><td>4</td><td>Rename the new table.</td></tr>
</table>
</td></tr>
<tr><td>Secondary
Hash
Join</td><td>
<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Drop the index.</td></tr>
<tr><td>2</td><td>Drop the index columns.</td></tr>
<tr><td>3</td><td>Define a new index on the correct columns.</td></tr>
</table>
</td></tr>
</table>

Adding Columns

When a column is added to a table that already contains data, Teradata 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 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	specified for the new column	contain the system default in the field.
DEFAULT WITH DEFAULT	*not* specified for the new column	are null for the field, and the NOT NULL phrase cannot be specified for the column.

A column that is specified for NOT NULL but has no DEFAULT or WITH DEFAULT phrase can only be added to an empty table.

You cannot add an identity column to an existing table, nor can you add the identity column attribute to an existing column.

Renaming Columns

You can rename columns within certain restrictions.

Renaming is rejected when any of the following is true:

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

Adding the UNIQUE or PRIMARY KEY Definition

A PRIMARY KEY can be added via the ALTER TABLE statement if and only if there is no PRIMARY KEY specification currently in the table, regardless of the type of index (UPI or USI) this PRIMARY KEY is currently mapped to.

Also, the PRIMARY KEY being added is always mapped to a USI. The PRIMARY KEY constraint can be dropped if and only if it is mapped to a USI.

Unnamed unique constraints must be dropped using the DROP INDEX statement.

ALTER TABLE...ADD/DROP UNIQUE/PRIMARY KEY require INDEX privileges on the table, or, for compatibility with the past, DROP TABLE privileges.

Adding or Dropping REFERENCES or FOREIGN KEY

An ALTER TABLE 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 user issuing the ALTER TABLE statement containing a REFERENCES option must have the REFERENCE privilege on the referenced table or on all specified columns of the referenced table.

DEFAULT FREESPACE

At any give time, the FREESPACE value returned in response to a SHOW TABLE statement is the value specified in the most recently entered ALTER TABLE or CREATE TABLE statement. See "CREATE TABLE" on page 1-182 and "SHOW HASH INDEX SHOW JOIN INDEX SHOW MACRO SHOW PROCEDURE SHOW TABLE SHOW TRIGGER SHOW VIEW" on page 4-79 for more information.

- Specifying the DEFAULT FREESPACE option when the percent freespace has been previously set to a value other than the default value resets the size to the system default value.
- If the FREESPACE option is not included in the ALTER TABLE statement, then the percent freespace attribute value is not changed.

IMMEDIATE DATABLOCKSIZE

Repacking the data blocks of large tables is a time-consuming process, so specifying the IMMEDIATE option for large tables will substantially increase the amount of time required for the ALTER TABLE statement to complete.

Field upgrades of systems do not change the data block size from 64 Kbyte to 128 Kbyte until a block splits. To take immediate advantage of the performance enhancements offered by the 128 Kbyte block size, you must force the data block size change directly.

To upgrade the data block size from 63.5 Kbyte to 127.5 Kbyte, perform one of the following ALTER TABLE statements 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 Kbytes, but the size increase does not occur until rows are inserted into the newly defined table.

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

When an ALTER TABLE statement with 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 new, specified size.

At any given time, the DATABLOCKSIZE value returned in response to a SHOW TABLE statement is the value specified in the most recently entered ALTER TABLE or CREATE TABLE statement.

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

Journaling

Journaling supports rollback and back up operations.

To preserve the restore capability of 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 journaling options include:

- WITH JOURNAL TABLE
- JOURNAL
- AFTER JOURNAL

NO/DUAL Journaling

IF ...	THEN ...
the JOURNAL keyword is specified without NO or DUAL	a single copy of the image is maintained unless FALLBACK is in effect or is also specified.
journaling is requested for a database that uses fallback protection	DUAL images are maintained automatically.

BEFORE/AFTER Journaling

IF ...	THEN ...
the JOURNAL keyword is specified without AFTER or BEFORE	both types of images are maintained. This option might appear twice in the same statement: once to specify a BEFORE or AFTER image, and again to specify the alternate type.
one type is specified	only the default for that type is overridden. For example, if AFTER is specified, before image journaling remains at the default setting.
both types are specified	the two specifications must not conflict.

LOCAL Journaling

See "Local Journaling" on page 1-83 and "CREATE TABLE" on page 1-182.

Maximum Number of Columns

A table can contain a maximum of 2 048 columns, and a total of 2 560 columns can be defined for a table over its lifetime. Dropping columns does not affect this rule. Therefore, if a table has 2 048 columns, the ALTER TABLE statement can be used to drop and replace up to 2 048 columns.

If new columns need to be added that would exceed this number, a new table must be created. The SHOW TABLE statement and the INSERT … SELECT statement can be used to create and load the spin-off table.

Changing Column Attributes: NOT NULL/NULL

Changing column attributes in ANSI SQL is restricted to setting or dropping a default clause.

To change the attributes of an existing column, the ALTER TABLE statement must include the ADD keyword. The following table summarizes the allowable cases for converting or adding NULL to NOT NULL, or NOT NULL to NULL columns.

WHEN you add this qualifier to a column ...	Composite changes are ...
NULL	never 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
Nonindexed	NULL	NOT NULL If and only if the column does not contain nulls.
	NOT NULL	NULL

Adding Compressed Columns

Neither the COMPRESS attribute nor the data type for a column can be changed if that would require a change to existing data. In both cases, you must first create a new table defining the proper attributes, then insert the values from the existing table into the new table.

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 statements 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), the INSERT statement can be used with a SELECT subquery to insert all the rows in a single statement (see "INSERT" in the chapter "SQL Data Manipulation Language Statement Syntax," in *Teradata RDBMS SQL Reference, Volume 6*).

You can compress nulls and as many as 255[1] distinct values per column. You can compress an unlimited number of columns per table.

IF this type of compression is used ...	THEN ...
single valued	the default is null if no value is specified explicitly.
multivalued	• 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. • There is no essential ordering for specifying values to be compressed for a given column. • The same value cannot be specified more than once in the compression value list for a given column.

1 This value is constrained by the maximum system row length because compressed values are added to the table header row for each column.

You can add a new column with multivalued compression to an existing table, but you cannot modify the compression specification for an existing column. Instead, you can either create a new table with the column designated for multivalued compression and INSERT … SELECT data into it from the existing table or you can use CREATE TABLE … AS with the target column designated for multivalued compression.

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

Compressed columns cannot be used for fast path optimization in an INSERT … SELECT operation.

Column compression is not supported for volatile tables or identity columns.

See the chapter "Database-Level Considerations" in *Teradata RDBMS Database Design* for more detailed information about value compression and its applications for performance optimization and disk capacity savings.

Rules for Adding or Modifying Compression

The following rules apply to adding or modifying rows with 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 value list 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 and any variable length columns in a table.
- The maximum number of distinct values that can be compressed per column is 255.
- If an attempt to modify or add the definition for a new column results in the table row header exceeding the maximum row length for Teradata, then the responsible ALTER TABLE request fails.
- You cannot modify an existing column to assume multivalued compression directly. Two workarounds exist for cases where you want to convert an existing column to support multivalued compression:

- The procedure for the first workaround is as follows:

Step	Action
1	Use ALTER TABLE 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 as follows:

Step	Action
1	Use CREATE TABLE 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.
3	Use ALTER TABLE to drop the old table.
4	Use ALTER TABLE to rename the new table to match the name of the dropped table.

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

  If the text exceeds the 8 000 character maximum size for DBC.TVFields.compresvaluelist, then the responsible 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 value list.

Adding and Dropping CHECK Constraints

The following section describes the rules for adding and dropping CHECK constraints.

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

- The following table explains rules for ADD and MODIFY.

THIS form ...	IS allowed only if *column_name* ...
ADD *column_name* CHECK (*boolean_condition*)	does *not* already have a constraint.
MODIFY	*does* have a constraint.

  Note that only one CHECK modification can be made per ALTER TABLE statement and that CHECK and non-CHECK alterations cannot be combined within a single ALTER TABLE statement.
- The following form drops all unnamed column level CHECK constraints on *column_name*:

  ```
  ALTER TABLE table_name DROP column_name CHECK
  ```
- The following form drops all unnamed table level CHECK constraints on the table name:

  ```
  ALTER TABLE table_name DROP CHECK
  ```
- Either of the following is legitimate syntax for dropping a named CHECK constraint:

  ```
  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 statements.
- 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 perform 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 statement, regardless of constraint type.

Adding and Dropping PRIMARY KEY and UNIQUE Constraints

ALTER TABLE … ADD UNIQUE/PRIMARY KEY 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 constraint if and only if no primary key is already defined for the table.

  There are no exceptions to this rule.

  The added primary key always maps to a unique secondary index (USI).
- You can drop a PRIMARY KEY constraint if and only if it is mapped to a unique secondary index (USI).
- For a table with a NUPI, you can only add 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.[2]
- Use the DROP INDEX statement to drop unnamed UNIQUE constraints.

Adding or Dropping Referential Integrity Constraints

Referential integrity provides for specification of columns within a referencing table that are foreign keys for columns in some other referenced table.

The columns in the referenced table must be defined either as the primary key columns for the table or have the unique attribute defined for them.

The foreign key columns in the referencing table must be identical in definition with the keys in the referenced table.

2 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.

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

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>An error table is generated when the new constraint is added to the table.
Its name is the name of the target table suffixed with the appropriate reference index number.
Its columns and primary key are the same as those of the target table.</td></tr>
<tr><td>2</td><td>The error table is created under the same user or database as the table being altered.

<table><tr><th>IF ...</th><th>THEN ...</th></tr><tr><td>a table with the same name as that generated for the error table already exists</td><td>an error is returned.</td></tr><tr><td>rows in the referencing table contain values in the foreign key columns that cannot be found in any row of the referenced table</td><td>no error is returned.

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.</td></tr></table>
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.</td></tr>
</table>

Also see the chapter “The Normalization Process” in *Teradata RDBMS Database Design* for more information about referential integrity.

The COMPRESS option is not allowed on either column. Column-level constraints are not compared for the columns.

Referential integrity is not supported for global temporary and volatile tables.

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 Referential Integrity Constraints” on page 1-34. See “Example 33: Add or Drop a Batch Referential Constraint” on page 1-59.

Adding or Dropping Soft Referential Integrity Constraints

Soft referential integrity provides 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, the rules for soft referential integrity constraints are identical to those documented by "Adding or Dropping Referential Integrity Constraints" on page 1-34. See "Referential Constraints" on page 1-230 and "Example 34: Adding a Table-Level Soft Referential Integrity Constraint" on page 1-60 for additional information.

Maintaining Foreign Key Integrity

Teradata verifies that the structure of columns defined as foreign keys or referenced by foreign keys, is not changed so as to violate the rules for definition of a foreign key constraint.

An ALTER TABLE (or DROP INDEX) statement attempting to change the structure of such a column causes an error to be returned by Teradata.

Teradata verifies that a table referenced by another is not dropped. Before dropping the table, an ALTER TABLE statement must be performed to drop the foreign key reference.

ADDING a FOREIGN KEY Constraint

If an ALTER TABLE statement adds a foreign key reference, then Teradata validates all the values in the foreign key columns against columns in the referenced table.

When the ALTER TABLE statement is performed, Teradata creates an error table with the same columns and PRIMARY KEY as the ALTER TABLE target table. If the target table has a partitioned primary index, then the primary index of the error table is also partitioned in the same way.

The name of the error table is the target table name suffixed with the reference index number. A reference index number is assigned to each FOREIGN KEY constraint for a table. You can determine the number of this reference index using either the RI_Distinct_Children or RI_Distinct_Parents system view (see *Teradata RDBMS Data Dictionary*).

The error table is created under the same user/database as the target table. If a table already exists with the same name as that generated for the error table, then an error is returned to the ALTER TABLE statement.

Teradata does not return an error if rows in the referencing table contain values in the foreign key columns which cannot be found in any row of the referenced table. Instead, a row is inserted into the error table for each such row found in the target table.

Correction of data values in the referenced or referencing tables so that full referential integrity exists between the two tables is the responsibility of the user. The rows in the error table can be used to define which corrections must be made. Maintenance of the error table is also the responsibility of the user.

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 integrity.

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. The Archive and Restore operations are the most common cause of such inconsistencies.

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 statement 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:

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

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

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 a Data Type" on page 1-38. Also see "Procedure to Change Column Data Types" on page 1-39.

For an existing column, the following items are true for data types and attributes:

- You can specify either of the following:
  - FORMAT, TITLE, or DEFAULT/WITH DEFAULT phrases.
  - Changes in data type attributes or in data types in cases where the change does not require rewriting existing rows.
- You *cannot* specify the COMPRESS phrase.

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

Changing a Data Type Attribute

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

COMPRESS cannot be changed for an existing column.

CASESPECIFIC or NOT CASESPECIFIC can be declared for character columns.

Rules for Changing a Data Type

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

You cannot change the character set attribute of an existing column. 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 for a column used in a primary or secondary index.
CHAR(n), UC CHAR(n)	CHAR(n) CHAR(n), CS	Lengths must remain identical. Can change only case specific setting.
CHAR	CHAR(n)	Not allowed if n > 1.
VARCHAR(m), UC	VARCHAR(n)	Cannot decrease maximum length. Cannot change from no UC to UC.
GRAPHIC	GRAPHIC(n)	Not allowed if n > 1.
VARGRAPHIC(m)	VARGRAPHIC(n)	Cannot decrease maxlength.
BYTE	BYTE(n)	Not allowed if n > 1.
VARBYTE(m)	VARBYTE(n)	Cannot decrease maximum length.
DATE	INTEGER	No restrictions.

Old Data Type	New Data Type	Restrictions on Changes
INTEGER	DATE	Cannot make this change because INTEGER column could include non-valid DATE values).
DECIMAL(n,0)	INTEGER	Can change only if n is in range 5 - 9.
DECIMAL(n,0)	SMALLINT	Can change only if n is 3 or 4.
DECIMAL(n,0)	BYTEINT	Can change only if n = 1 or 2.
DECIMAL(n,f)	DECIMAL(m,f)	Can change only as follows: • m >= n, no change in f • n = 1 or 2, m < 3 • n = 3 or 4, m < 5 • n = 5 to 9, m < 10 • n = 10 to 15, m <= 18

Procedure to Change Column Data Types

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

Step	Action
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 statement.

3	Catalog the access privileges of the old table before step 4. Use the following syntax:
	`SELECT username, accessright, grantauthority, columnname, allnessflag`
	`FROM dbc.allrights`
	`WHERE tablename = 'table_name' AND databasename = 'database_name';`

4	Drop the old table.

5	Rename the new table with that of the old table.
	As an example, use the following sequence of statements to expand the data type attribute for the name column from 12 to 14 characters via a temporary table:
	`CREATE TABLE temp, FALLBACK`
	`(EmpNo INTEGER NOT NULL FORMAT 'ZZZZ9',`
	`Name CHAR(14) NOT NULL,`
	`…`
	`HCap BYTEINT FORMAT 'Z9')`
	`UNIQUE PRIMARY INDEX (index_name) ;`

Step	Action

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 enter a CREATE TABLE statement for an existing table, you receive an error.
	To facilitate re-creating a table according to Step 1, you can display the CREATE TABLE statement for the table by executing the SHOW TABLE statement.
	For example, the following statement displays a reconstruction of the CREATE TABLE statement for table dept_em.
	`SHOW TABLE dept_em ;`
	If using BTEQ, when the CREATE TABLE statement is displayed, you can change the statement (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 access privileges to the employee table again, as they were lost when the table was dropped. Use the same access 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 statement.

The Teradata ALTER TABLE statement offers 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 in Teradata SQL.

Redefining the Primary Index

There are two ways to redefine the primary index for a table:

- Use the primary index modification syntax for ALTER TABLE to modify the index (see "Primary Index Modification Syntax" on page 1-14 and "General Rules for the Modify Primary Index Option" on page 1-43).
- Create a new table with the primary index defined properly and then use an INSERT … SELECT to copy the data from the old table to the new table (see "Redefining a Primary Index Without Using ALTER TABLE" on page 1-41).

You can use ALTER TABLE to perform the following changes for primary indexes:

To perform this operation ...	Specify ...
Change a NUPI to a UPI	MODIFY UNIQUE PRIMARY INDEX in the Primary Index Change Options clause (see "General Rules for the Modify Primary Index Option" on page 1-43).
Change a UPI to a NUPI	MODIFY NOT UNIQUE PRIMARY INDEX in the Primary Index Change Options clause (see "General Rules for the Modify Primary Index Option" on page 1-43).
Change a PPI to an NPPI	NOT PARTITIONED in the Primary Index Change Options clause (see "General Rules for the Modify Primary Index Option" on page 1-43 and "Rules for Altering the Partitioning Expression for a Primary Index" on page 1-46).
Change an NPPI to a PPI	a PARTITION BY clause within the Primary Index Change Options clause (see "General Rules for the Modify Primary Index Option" on page 1-43 and "Rules for Altering the Partitioning Expression for a Primary Index" on page 1-46).
Add or drop PPI partitioning expression ranges	an ADD RANGE or DROP RANGE clause within the Primary Index Change Options clause (see "General Rules for the Modify Primary Index Option" on page 1-43 and "Rules for Adding and Dropping Ranges in a Partitioning Expression" on page 1-48).
Validate a PPI by regenerating its table headers and correcting any errors in row partitioning	a REVALIDATE PRIMARY INDEX clause within the Primary Index Change Options clause (see "General Rules for the REVALIDATE PRIMARY INDEX Option" on page 1-49).

Redefining a Primary Index Without Using ALTER TABLE

To redefine the primary index for a table without using an ALTER TABLE statement, perform a procedure similar to that described in "Procedure to Change Column Data Types" on page 1-39:

Step	Action
1	Using a different name, create a new table that specifies the new index.
2	Populate the new table using an INSERT … SELECT statement.
3	Catalog the access privileges on the old table. See step 3 in "Procedure to Change Column Data Types" on page 1-39.
4	Drop the old table.
5	Rename the new table with that of the old table.
6	Grant access privileges on the new table.

Performance Implications of Altering a Primary Index Using ALTER TABLE

You should consider the following information about the immediate performance impact of altering the primary index for a table to best schedule the operation in a way that has the least impact on your production workload.

- Altering an empty table is fairly quick.
- Altering the table to have a unique or nonunique primary index is fairly quick.
- Altering the table to change the primary index name is fairly quick.
- The following cumulative performance issues are all concerned with dropping or adding new ranges or partitions to a PPI:
  - When dropping or adding new ranges or partitions for a populated table, the operation can be fairly quick because rows that remain in the retained ranges and partitions need not be processed or repartitioned.
  - There is an additional small 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 and there are populated NO RANGE [OR UNKNOWN) or UNKNOWN partitions or rows in dropped ranges that need to be moved to the added ranges because rows must be processed to determine if any of them need to be assigned to one of the new ranges and, if so, to move them to the correct range.
  - Any secondary, join, or hash indexes on the table must be updated. Updating secondary, join, and hash indexes can be lengthy operations and depend 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 degree of this 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.

General Rules for the Primary Index Options Clause

The following rules apply to both the MODIFY PRIMARY INDEX option and the REVALIDATE PRIMARY INDEX option of the Primary Index Options clause.

- You cannot alter the partitioning of the primary index for a populated table unless all the following guidelines are observed:
  - You must use ADD RANGE or DROP RANGE (or both) options to modify the partitioning.
  - If you drop ranges, you must drop them either from the beginning or from the end of the defined ranges for the index.

    - If you add ranges, you must add them either before the beginning or after the end of the defined ranges for the index and after dropping any ranges.
- You cannot alter the primary index for any of the following database objects using the ALTER TABLE statement:
    - Global temporary tables
    - Volatile tables
    - Join indexes
    - Hash indexes
- You must disable some triggers before you can perform an ALTER TABLE statement that modifies or revalidates the primary index for a table.

IF you specify this clause in an ALTER TABLE statement that modifies or revaluates the primary index ...	THEN you must disable these triggers before you perform the statement ...
WITH DELETE	all delete triggers on *table_name*. After the ALTER TABLE statement completes, you can re-enable the disabled triggers.
WITH INSERT	• all delete triggers on *table_name*. • all insert triggers on *save_table*. After the ALTER TABLE statement completes, you can re-enable the disabled triggers.

General Rules for the Modify Primary Index Option

The following general rules and observations apply to the Modify Primary Index option of the Primary Index Options Clause (see "Rules for Altering the Partitioning Expression for a Primary Index" on page 1-46 for the subset of rules that apply to partitioning expressions):

- You cannot modify both primary index and non-primary index table definitions within a single ALTER TABLE statement because the operations are mutually exclusive.
- You cannot modify both primary index definitions and revalidate a primary index within a single ALTER TABLE statement because the operations are mutually exclusive.
- Any change to how a primary index is partitioned affects the values for the system-derived columns PARTITION and ROWID for most, if not all, rows in a table.
- Any change to the structure of a table increments its version number.

  This means that you cannot perform any of the following actions over a DDL statement that alters table structure:
    - Cluster restore
    - Single AMP restore
    - Rollforward or rollback of a permanent journal

If the ARC utility encounters such a structural table change during a rollforward or rollback operation, then that operation is stopped and an error message is returned in the output listing.

- At least one specified option must change the definition of the primary index or an error is returned.

  Specifying a different partitioning expression for an existing PPI table is considered to be a valid change even if the results of the new expression are identical to those produced by the former partitioning expression.
- The following rules apply to changing the name of a primary index:
  - The *index_name* string cannot be the same as the name of any secondary index or constraint defined for the table.
  - If you specify a string for *index_name*, then that is the name of the primary index.
  - If you specify NOT NAMED, then the primary index has no name.
  - If you specify neither an *index_name* string nor a NOT NAMED phrase, then no change is made to the current name.
- If you do not specify a column set for the primary index, then the existing column set for the primary index is retained.
- 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, an error is returned.
- If you specify a column set for the primary index of a non-empty table, then that column set must be identical to the existing column set for the primary index.

  Conversely, you can specify a column set for the primary index that differs from the current primary index column set only if the table is empty.

  Otherwise, an error is returned.
- If you specify a column set for the primary index of a non-empty table that differs from a current primary index that is defined implicitly by a PRIMARY KEY or UNIQUE constraint, then that constraint is dropped.
- If the current primary index for a table is defined implicitly by a PRIMARY KEY or UNIQUE constraint and a PARTITION BY *partitioning_expression* phrase is specified, then that constraint is dropped.
- If you specify a PARTITION BY *partitioning_expression* phrase, then the table is altered to have a partitioned primary index partitioned on the new partitioning expression with the following qualifications:
  - The table is empty

    or
  - The new partitioning_expression clause is defined using ADD and DROP specifications that do not:
    - Drop ranges except from either the beginning or end of the partition range list and that retain at least one remaining range.
    - Add ranges between the resulting beginning and ending partition ranges.

  Otherwise, an error is reported.

- If you specify a NOT PARTITIONED phrase and the existing table has a PPI, then its primary index is modified to be an NPPI if the table is empty.

  If the table is not empty, then an error is reported.
- If you specify a NOT PARTITIONED phrase and the existing table has an NPPI, then the primary index is not changed with respect to its partitioning.
- If you specify neither a PARTITIONED BY nor a NOT PARTITIONED phrase, then the primary index is not changed with respect to its partitioning.
- If you specify a UNIQUE phrase, then the new primary index for the table is a UPI.

  Note that a PPI can be unique only if all its partitioning expression columns are also included in the column set defining the primary index itself.

  You cannot specify UNIQUE unless any of the following conditions is true for the current primary index for the table:
  - The primary index is already 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 empty.

  Otherwise, an error is reported.
- If a resulting primary index is unique and if there is also an existing USI on the same column set, whether explicitly or implicitly defined, then that USI and any associated PRIMARY KEY or UNIQUE constraint is dropped because it is redundant.
- If a resulting primary index is a NUPI and the current primary index was implicitly defined by a PRIMARY KEY or UNIQUE constraint, then that constraint is dropped.
- If you specify a NOT UNIQUE phrase, then the new primary index for the table is a NUPI.
- If you specify neither UNIQUE nor NOT UNIQUE, then the uniqueness of the primary index for the table is not changed.
- You cannot specify a nonunique NPPI if there is an existing USI defined on the same column set.

Rules for Altering the Partitioning Expression for a Primary Index

The following rules and observations apply to the Modify Primary Index option of the Primary Index Options Clause when you alter the partitioning expression for a PPI (also see "Rules for Adding and Dropping Ranges in a Partitioning Expression" on page 1-48):

- You can write any valid SQL expression as a partitioning expression with the following exclusions:
  - Expressions that require comparisons of CHARACTER or GRAPHIC data to be evaluated. Such comparisons use the following operators in their evaluation:
    - =
    - >
    - <
    - >=
    - <=
    - BETWEEN
    - LIKE
  - CHARACTER and GRAPHIC comparison functions such as the following:
    - CHAR2HEXINT
    - INDEX
    - LOWER
    - MINDEX
    - POSITION
    - TRANSLATE
    - TRANSLATE_CHK
    - TRIM
    - UPPER
    - VARGRAPHIC
  - Aggregate functions
  - Grouped row ordered analytic functions
  - Built-in functions
  - The RANDOM function
  - The HASHAMP and HASHBAKAMP functions[3]
  - The ROWID and PARTITION system-derived columns
  - The UPPERCASE attribute
  - Set operators
  - Subqueries

3 HASHROW and HASHBUCKET *are* permitted.

- If the data type for the result of *partitioning_expression* is not INTEGER, then the value is implicitly cast to the INTEGER type.

  If the result type cannot be cast to INTEGER, then an error is returned.
- If *partitioning_expression* is only a CASE_N function, then the number of conditions defined must be less than or equal to 65 533[4] (see the description of CASE_N in *Teradata RDBMS SQL Reference, Volume 5*).
- If partitioning_expression is only a RANGE_N function, then the number of ranges defined for the function must be less than or equal to 65 533[3] (see the description of RANGE_N in *Teradata RDBMS SQL Reference, Volume 5*).
- Teradata derives a table-level CHECK constraint from the partitioning expression. The text for this derived constraint cannot exceed 8 192 characters.
- If you do not specify a WITH phrase, then the new partitioning expression for all rows currently in the table must, after casting to INTEGER if it is not already typed as INTEGER, generates a non-null value between 1 and 65 535.

  Otherwise, an error is returned.
- If you specify a WITH DELETE phrase, then any row for which the new partitioning expression does not generate, after casting to INTEGER if it is not already typed as INTEGER, a non-null value between 1 and 65 535 is deleted.
- If you specify a WITH INSERT [INTO] *save_table* phrase, any row for which the new partitioning expression does not generate, after casting to INTEGER if it is not already typed as INTEGER, a non-null value between 1 and 65 535, is deleted from the table and inserted into *save_table*.

  save_table cannot be the same table as *table_name* or an error is returned.

  save_table must have the same number of columns with compatible data types as *table_name*, otherwise an error is returned.

  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 transaction mode ...	THEN the following block of work is rolled back ...
ANSI	request.
Teradata	transaction.

- If evaluation of the new partitioning expression for a row causes evaluation errors (such as divide by zero), then any changes made to the table or to *save_table* are rolled back and the primary index is not changed.
- The new partitioning expression becomes the effective partitioning expression for the primary index as soon as the table is altered successfully.

4 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 65 533 conditions.

- If there is a subsequent attempt to insert or update a row of a PPI table such that the partitioning expression for that row does not generate, after casting to INTEGER if it is not already typed as INTEGER, a non-null value between 1 and 65 535, then an error is returned for the insert or update operation.
- Increasing the number of active partitions might degrade the performance of primary index accesses and joins, but allows for finer partition elimination (see "Comparison of Partitioned and Nonpartitioned Primary Index Access for Some Typical Operations" in the chapter "Primary Indexes" in *Teradata RDBMS Database Design* for details).

Rules for Adding and Dropping Ranges in a Partitioning Expression

The following rules apply to the DROP RANGE and ADD RANGE options:

- If you specify either DROP RANGE or ADD RANGE, the existing primary index must be partitioned by a partitioning expression that consists solely of a RANGE_N function.

  Otherwise, Teradata returns an error.
- The conditional expression in a DROP RANGE WHERE clause must reference the system-derived column PARTITION and no other columns.

  If the table you are attempting to modify has a user-defined column named PARTITION, then this syntax is not valid and returns an error.
- Each range specified for a *start_expression* or *end_expression* must conform to the rules of a RANGE_N function (see *Teradata RDBMS SQL Reference, Volume 5*, for details). For example, the following rules must be observed:
  - A range value must be compatible in data type with the test expression that defines the existing partitioning expression for the primary 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.
- You can specify either an OR UNKNOWN clause or an UNKNOWN clause for a range expression sequence, but not both.

  If you specify both, an error is returned.
- An ALTER TABLE statement that specifies DROP RANGE or ADD RANGE options is equivalent to the following generalized ALTER TABLE statement syntax expressed in pseudo-BNF format:

  ```
  ALTER TABLE <table_name>
   MODIFY [[NOT] UNIQUE] PRIMARY INDEX [index_name | NOT  NAMED]
     [(primary_index_column_list)]
     PARTITION BY new_partitioning_expression
     [null_partition_handler];
  ```

  where *null_partition_handler* refers to a WITH INSERT | DELETE clause and *new_partitioning_expression* is the same as the existing partitioning expression except for the following properties:

- If you specify the DROP RANGE WHERE *conditional_expression* option, the new partitioning expression does not include the specified existing ranges and partitions where *conditional_expression* is true.

  An error is returned if *conditional_expression* evaluates to unknown for an existing range.
- If you specify a DROP RANGE BETWEEN range sequences clause, the new partitioning expression does *not* include the specified ranges and partitions in the range sequence.

  The specified ranges and partitions must match existing ranges (after expansion with EACH clauses in both the old partitioning expression and the DROP RANGE clause) and partitions; otherwise, an error is returned.

  For ranges to match, their start and end must match. A dropped range that spans multiple existing ranges is not considered to be a match.
- If you specify an ADD RANGE BETWEEN range sequences clause, the new partitioning expression *does* include the addition of the new ranges and partitions merged in sequential order with the existing ranges after first applying the DROP RANGE clause.

  The ranges cannot overlap with any of the existing ranges after first applying the DROP RANGE clause or an error is returned.

  If you add partitions, they must not conflict with or be the same as existing partitions after first applying the DROP RANGE clause or an error is returned.

- The resulting new partitioning expression must specify at least one range or an error is returned.
- 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 partitions.

General Rules for the REVALIDATE PRIMARY INDEX Option

REVALIDATE PRIMARY INDEX regenerates table headers for a PPI and optionally verifies the partitioning of rows in the table. If partitioning errors are detected, they are then corrected.

You might need to revalidate a PPI when any of the following conditions exist:

- The table headers for a table with a partitioned primary index are marked as non-valid.

  This is most often found after an upgrade or after restoring or copying a table to a system having a different operating system or different hardware.
- The RoundHalfwayMagUp setting in the DBSControl record is changed and the partitioning expression includes decimal operations.

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

- The table is restored or copied to another system that has a different RoundHalfwayMagUp setting in its DBSControl record and the partitioning expression includes decimal operations.

  A change in decimal rounding rules can cause the partitioning expression to evaluate differently.
- The table is restored or copied to another system and the partitioning expression includes floating point operations.

  Floating point calculations might cause the partitioning expression to evaluate differently on different systems.
- The table is restored or copied to another system that has a different hashing algorithm and the partitioning expression includes a HASHROW or HASHBUCKET function.

  A different hashing function can cause the partitioning expression to evaluate differently.
- Teradata is upgraded in such a way that the calculation of the partitioning expression changes.
- The CheckTable utility or valid SQL queries indicate rows with incorrect internal partition numbers or rows in the wrong partition.

  If this occurs unexpectedly, please report the problem to the GSC immediately.

The following rules apply to the REVALIDATE PRIMARY INDEX option:

- The primary index must be a partitioned primary index.

  Otherwise, an error is returned.
- If you specify neither a WITH DELETE nor a WITH INSERT phrase for the option, then only the table headers for the table are regenerated and no error checking or repair is performed.
- If you specify either a WITH DELETE or a WITH INSERT phrase, then in addition to regenerating the table headers for the table, all the rows in the table are validated for their partition number and row hash.
- If you specify a WITH DELETE phrase, then any row for which the new partitioning expression does not generate a non-null value between 1 and 65 535 (after casting to INTEGER if the value is not already typed as INTEGER) is deleted.
- If you specify a WITH INSERT [INTO] *save_table* phrase, any row for which the new partitioning expression does not generate a non-null value between 1 and 65 535 (after casting to INTEGER if the value is not already typed as INTEGER) is deleted from the table and inserted into *save_table*.

  save_table cannot be the same table as *table_name* or an error is returned.

  save_table must have the same number of columns with compatible data types as *table_name*, otherwise an error is returned.

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 the new partitioning expression for a row causes evaluation errors (such as divide by zero), then any changes made to the table or to *save_table* are rolled back and the primary index 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 with one of the following solutions:
  - Make the primary index for the table an NPPI.
  - 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 internal partition number and row hash are updated 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 an error is returned.
- Any secondary indexes, join indexes, and hash indexes are updated as needed.
- Values of the system-derived columns PARTITION and ROWID might change for the rows in the table.

  When you revalidate a table, its version number is incremented because its structure is altered. When the version number changes, you cannot perform any of the following actions over a DDL statement that alters table structure:
  - Cluster restore
  - Single AMP restore
  - Rollforward or rollback of a permanent journal

  If the ARC utility encounters such a structural table change during a rollforward or rollback operation, then that operation is stopped and an error message is returned in the output listing.

Examples

The following examples illustrate the use of ALTER TABLE.

Example 1: Add Columns to a Table

Use the following to add the columns Street and City to the Employee table:

```
ALTER TABLE Employee
  ADD Street VARCHAR(30),
  ADD City VARCHAR(20);
```

Example 2: Add Fallback to a Table

Use the following to implement fallback protection for the Employee table:

```
ALTER TABLE Employee,
  FALLBACK;
```

Example 3: Add a Single Before-Image Journal and Dual After-Image Journal

If a table named NewEmp is set for no journaling, but the database has a defined journal, the following statement could be used to add a single before-image journal and a dual after-image journal:

```
ALTER TABLE NewEmp,
  BEFORE JOURNAL,
  DUAL AFTER JOURNAL;
```

Example 4: Change a Single Before-Image Journal to a Dual Before-Image Journal and Drop Two Columns

The following request changes the single before image journal to a dual image journal and also drops two columns:

```
ALTER TABLE NewEmp,
  DUAL BEFORE JOURNAL
  DROP Phone,
  DROP PRef;
```

Example 5: Add a TITLE Phrase

The following example changes the attributes of the DeptNo column in the Department table to include a TITLE phrase:

```
ALTER TABLE Personnel.Department
  ADD DeptNo TITLE 'Depart';
```

Example 6: Add the Null Attribute to Columns

The following statements add the NULL qualifier to columns training_skill and column_2:

```
ALTER TABLE personnel
  ADD training_skill NULL;

ALTER TABLE abc
  ADD column_2 NULL;
```

Example 7: Composite Changes: NULL to NOT NULL

A composite change to a column implies changing more than one attribute of that column in one statement. NULL columns can be changed to NOT NULL, along with other attribute changes, in one statement. The following example changes both the case and NULL attribute of column_2. An error is returned if abc contains any rows with nulls in column_2.

```
ALTER TABLE abc
  ADD column_2 CASESPECIFIC NOT NULL;
```

Example 8: Composite Changes: NOT NULL to NULL

Composite changes to columns that involve changing the NOT NULL attribute to NULL are not valid. For example, the following *cannot* be done

```
ALTER TABLE abc
  ADD column_2 TITLE 'newname' NULL;
```

To make these changes, you must perform two separate ALTER TABLE statements:

```
ALTER TABLE abc ADD column_2 TITLE 'newname';
ALTER TABLE abc ADD column_2 NULL;
```

Example 9: DATABLOCKSIZE

The following ALTER TABLE statement changes the maximum data block size of 130 560 bytes (255 sectors) and repacks existing data into blocks of that size:

```
ALTER TABLE Employee,
  DATABLOCKSIZE = 130560 BYTES IMMEDIATE;
```

Example 10: FREESPACE

The following ALTER TABLE statement changes the freespace to 5 percent:

```
ALTER TABLE Employee,
  FREESPACE = 5 PERCENT;
```

Example 11: Add a New Column With Multivalued Compression

The following ALTER TABLE statement adds a new compressible column and specifies the maximum number of distinct values for compression.

```
ALTER TABLE mkw.x
  ADD b CHARACTER (2)
  COMPRESS (
   'a1','a2','a3','a4','a5','a6','a7','a8','a9','a10',
   'b1','b2','b3','b4','b5','b6','b7','b8','b9','b10',
   'c1','c2','c3','c4','c5','c6','c7','c8','c9','c10',
   'd1','d2','d3','d4','d5','d6','d7','d8','d9','d10',
   'e1','e2','e3','e4','e5','e6','e7','e8','e9','e10',
   'f1','f2','f3','f4','f5','f6','f7','f8','f9','f10',
   'g1','g2','g3','g4','g5','g6','g7','g8','g9','g10',
   'h1','h2','h3','h4','h5','h6','h7','h8','h9','h10',
   'i1','i2','i3','i4','i5','i6','i7','i8','i9','i10',
   'j1','j2','j3','j4','j5','j6','j7','j8','j9','j10',
   'k1','k2','k3','k4','k5','k6','k7','k8','k9','k10',
   'l1','l2','l3','l4','l5','l6','l7','l8','l9','l10',
   'm1','m2','m3','m4','m5','m6','m7','m8','m9','m10',
   'n1','n2','n3','n4','n5','n6','n7','n8','n9','n10',
   'o1','o2','o3','o4','o5','o6','o7','o8','o9','o10',
   'p1','p2','p3','p4','p5','p6','p7','p8','p9','p10',
   'q1','q2','q3','q4','q5','q6','q7','q8','q9','q10',
   'r1','r2','r3','r4','r5','r6','r7','r8','r9','r10',
   's1','s2','s3','s4','s5','s6','s7','s8','s9','s10',
   't1','t2','t3','t4','t5','t6','t7','t8','t9','t10',
   'u1','u2','u3','u4','u5','u6','u7','u8','u9','u10',
   'v1','v2','v3','v4','v5','v6','v7','v8','v9','v10',
   'w1','w2','w3','w4','w5','w6','w7','w8','w9','w10',
   'x1','x2','x3','x4','x5','x6','x7','x8','x9','x10',
   'y1','y2','y3','y4','y5','y6','y7','y8','y9','y10',
   'z1','z2','z3','z4','z5');
```

Adding or Dropping Named Constraints

The following set of examples deals with adding or dropping named constraints.

Example 12: Add a PRIMARY KEY Constraint

table_1 must not already have a defined PRIMARY KEY constraint. Constraint name primary_1 must not duplicate an existing constraint name.

```
ALTER TABLE table_1
  ADD CONSTRAINT primary_1 PRIMARY KEY (field_1, field_2);
```

field_1 and field_2 must be NOT NULL or an error is returned.

Example 13: Add a Named CHECK Constraint

Constraint name check_1 must not be an existing name in table_1.

```
ALTER TABLE table_1 ADD CONSTRAINT check_1
  CHECK (field_1 > field_2);
```

Example 14: Add a Named UNIQUE Constraint

Constraint name unique_1 must not be an existing name in table_1.

```
ALTER TABLE table_1
  ADD CONSTRAINT unique_1
  UNIQUE (field_1, field_2);
```

Example 15: Add a named FOREIGN KEY Constraint

Constraint name reference_1 must not be an existing name in table_1.

```
ALTER TABLE table_1
  ADD CONSTRAINT reference_1
  FOREIGN KEY (field_1) REFERENCES table_2;
```

Example 16: Drop a Named CHECK Constraint

Constraint name check_1 must exist in `table_1`.

```
ALTER TABLE table_1
  DROP CONSTRAINT check_1;
```

Example 17: Drop a Named CHECK Constraint

Constraint name `check_1` must exist in `table_1` as a CHECK constraint.

```
ALTER TABLE table_1
  DROP CONSTRAINT check_1
  CHECK;
```

Example 18: Drop a UNIQUE Constraint

Constraint name unique_1 must exist in table_1.

```
ALTER TABLE table_1
  DROP CONSTRAINT unique_1;
```

Example 19: Drop a Named Constraint

Constraint name reference_1 must exist in table_1.

```
ALTER TABLE table_1
  DROP CONSTRAINT reference_1;
```

Example 20: Modify a Named CHECK Constraint

Replace the current definition for CHECK constraint check_1 with the new definition.

Constraint name check_1 must be an existing CHECK constraint.

```
ALTER TABLE table_1
  MODIFY CONSTRAINT check_1
  CHECK (field_2 > 0);
```

Adding or Dropping Unnamed Constraints

The following set of examples deals with adding or dropping unnamed constraints.

Example 21: Add an Unnamed CHECK Constraint

The following statement adds an unnamed CHECK constraint to ensure that values in column field_2 are always greater than 100.

```
ALTER TABLE table_1
  ADD CHECK (field_2 > 100);
```

Example 22: Add a Column With an Unnamed CHECK Constraint

The following statement adds an unnamed CHECK constraint to ensure that values in column field_1 are always greater than 0.

This example is valid only if there is not an existing unnamed column level CHECK for field_1 in table_1.

```
ALTER TABLE table_1
  ADD field_1
  CHECK (field_1 > 0);
```

Example 23: Add a Multicolumn Unnamed UNIQUE Constraint

The following statement adds an unnamed uniqueness constraint to the columns named field_3 and field_4.

```
ALTER TABLE table_1
  ADD UNIQUE (field_3, field_4);
```

Example 24: Add a FOREIGN KEY Constraint

The following statement adds a foreign key constraint on the column named field_2 in table_1.

```
ALTER TABLE table_1
  ADD FOREIGN KEY (field_2) REFERENCES table_3;
```

Example 25: Drop All Unnamed Table-Level CHECK Constraints

This example drops all unnamed table-level CHECK constraints.

```
ALTER TABLE table_1
  DROP CHECK;
```

Example 26: Drop an Unnamed Column-Level CHECK Constraint

This example drops the unnamed column level CHECK constraint from column field_1 in table_1.

```
ALTER TABLE table_1 DROP field_1
  CHECK;
```

Example 27: Drop a FOREIGN KEY Constraint

This example drops the foreign key constraint on the column named column field_2 in table table_3.

```
ALTER TABLE table_1
  DROP FOREIGN KEY (field_2) REFERENCES table_3;
```

Example 28: Add an Unnamed CHECK Constraint

This example modifies column by adding an unnamed CHECK constraint. Column field_1 is an existing, already constrained field in table_1.

```
ALTER TABLE table_1
  MODIFY field_1
  CHECK (field_1 IS NOT NULL);
```

Example 29: Non-Valid Use of ALTER TABLE With Constraints

The following statement is not valid because you can add or drop only one constraint per ALTER TABLE statement.

```
ALTER TABLE table_1
  DROP CONSTRAINT check_1,
  ADD CONSTRAINT check_2
  CHECK (field_2 > 0);
```

Teradata returns the message:

```
Only a check specification is allowed for the modification.
```

Example 30: Non-Valid Use of ALTER TABLE With Named Constraints

The following statement is not valid because the constraint named dup_constr_name already exists in table_1. For this reason, you cannot add it to the table.

```
ALTER TABLE table_1
  ADD CONSTRAINT dup_constr_name
  FOREIGN KEY (field_3) REFERENCES table_2;
```

Teradata returns the message:

```
Constraint with the same name 'dup_constru_name' already
exists in table.
```

Example 31: Non-Valid Use of ALTER TABLE With Named Constraints

The following statement is not valid because the constraint named no_such_constr does not exist in table_1. For this reason, you cannot modify or drop it.

```
ALTER TABLE table_1
  MODIFY CONSTRAINT no_such_constr
  CHECK (field_1 > 0);
```

Example 32: Non-Valid Use of ALTER TABLE With Named Constraints

The following statement is not valid because the specified constraint does not exist; therefore, it cannot be dropped.

```
ALTER TABLE table_1
  DROP CONSTRAINT no_such_constr;
```

Teradata returns the message:

```
The specified constraint name does not exist in table.
```

Example 33: Add or Drop a Batch Referential Constraint

This example first adds a table-level batch referential constraint to column d1 in table drs.t2 referencing column c1 in table drs.t1 and then drops that same constraint.

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


CREATE SET TABLE drs.t2, NO FALLBACK,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL
     (
      d1 INTEGER,
      d2 INTEGER,
      d3 INTEGER);


ALTER TABLE drs.t2
ADD CONSTRAINT fpk1
FOREIGN KEY (d1) REFERENCES WITH CHECK OPTION drs.t1 (c1);


ALTER TABLE drs.t2
DROP FOREIGN KEY (d1) REFERENCES WITH CHECK OPTION drs.t1 (c1);
```

Example 34: Adding a Table-Level Soft Referential Integrity Constraint

The first statement adds a table-level soft referential integrity relationship on foreign key column A3 with column E1 in table E. Referential integrity is not enforced for this relationship.

The second statement drops the foreign key SRI constraint that was created by the first.

```
ALTER TABLE A
ADD FOREIGN KEY (A3) REFERENCES WITH NO CHECK OPTION E(E1);

ALTER TABLE A
DROP FOREIGN KEY (A3) REFERENCES E(E1);
```

Example 35: CREATE TABLE Statement for Examples 35 - 42

Examples 35 through 42 assume that the following table has been created:

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

Example 36: Non-Valid MODIFY PRIMARY INDEX Statements

The following statements are illegal because they do not specify options that alter the table:

```
ALTER TABLE orders
MODIFY PRIMARY INDEX;

ALTER TABLE orders
MODIFY NOT UNIQUE PRIMARY INDEX;
```

Example 37: Modify the Primary Index Partition Ranges

The following ALTER TABLE statement modifies the table created in "Example 35: CREATE TABLE Statement for Examples 35 - 42" on page 1-60. The statement is valid if the table is empty.

```
ALTER TABLE orders
MODIFY PRIMARY INDEX
PARTITION BY RANGE_N(
  o_orderdate BETWEEN DATE '1993-01-01' AND DATE '2000-12-31'
                      EACH INTERVAL '1' MONTH);
```

Example 38: Drop and Add Primary Index Partition Ranges

The following ALTER TABLE statement modifies the table created in "Example 35: CREATE TABLE Statement for Examples 35 - 42" on page 1-60. The outcome of this statement is identical to the table alterations performed in "Example 39: Drop and Add Primary Index Partition Ranges" and "Example 40: Drop and Add Primary Index Partition Ranges" using different syntax.

The statement is valid if there are no rows with values for o_orderdate between January 1 1992 (DATE '1992-01-01') and December 31 1992 (DATE '1992-12-31').

```
ALTER TABLE orders
MODIFY PRIMARY INDEX
  DROP RANGE BETWEEN DATE '1992-01-01' AND DATE '1992-12-31'
                     EACH INTERVAL '1' MONTH
  ADD RANGE BETWEEN DATE '1999-01-01' AND DATE '2000-12-31'
                    EACH INTERVAL '1' MONTH;
```

Example 39: Drop and Add Primary Index Partition Ranges

The following ALTER TABLE statement modifies the table created in "Example 35: CREATE TABLE Statement for Examples 35 - 42" on page 1-60. The result is equivalent to the outcome of the statements used in "Example 38: Drop and Add Primary Index Partition Ranges" and "Example 40: Drop and Add Primary Index Partition Ranges" using different syntax and is valid if there are no rows with values for o_orderdate between January 1 1992 (DATE '1992-01-01') and December 31 1992 (DATE '1992-12-31').

```
ALTER TABLE orders
MODIFY PRIMARY INDEX
  DROP RANGE WHERE PARTITION BETWEEN 1 AND 12
  ADD RANGE BETWEEN DATE '1999-01-01' AND DATE '2000-12-31'
                    EACH INTERVAL '1' MONTH;
```

Example 40: Drop and Add Primary Index Partition Ranges

The following ALTER TABLE statement modifies the table created in "Example 35: CREATE TABLE Statement for Examples 35 - 42" on page 1-60. The result is equivalent to the outcome of the statements used in "Example 38: Drop and Add Primary Index Partition Ranges" and "Example 39: Drop and Add Primary Index Partition Ranges" using different syntax and is valid if there are no rows with values for o_orderdate between January 1 1992 (DATE '1992-01-01') and December 31 1992 (DATE '1992-12-31').

```
ALTER TABLE orders
MODIFY PRIMARY INDEX
  DROP RANGE WHERE Orders.PARTITION IN (1,2,3,4,5,6,7,8,9,10,11,12)
  ADD RANGE BETWEEN DATE '1999-01-01'
                    EACH INTERVAL '1' MONTH,
                    DATE '2000-01-01' AND DATE '2000-12-31'
                    EACH INTERVAL '1' MONTH;
```

Example 41: Drop and Add Primary Index Partition Ranges and Delete Rows Outside the Defined Ranges

The following ALTER TABLE statement modifies the table created in "Example 35: CREATE TABLE Statement for Examples 35 - 42" on page 1-60. It is valid if there are 0 or more rows with o_orderdate values between January 1 1992 (DATE '1992-01-01') and December 31 1992 (DATE '1992-12-31'). As a result of the WITH DELETE specification, those rows, if any, are deleted from orders:

```
ALTER TABLE orders
MODIFY PRIMARY INDEX (o_orderkey)
  DROP RANGE BETWEEN DATE '1992-01-01' AND DATE '1992-12-31'
                     EACH INTERVAL '1' MONTH
  ADD RANGE BETWEEN DATE '1999-01-01' AND DATE '2000-12-31'
                    EACH INTERVAL '1' MONTH
WITH DELETE;
```

Example 42: Using MODIFY PRIMARY INDEX to Repartition a Table and Save Non-Valid Rows That Result From the Redefinition in a Save Table

Assume that the orders table defined in "Example 35: CREATE TABLE Statement for Examples 35 - 42" on page 1-60 exists and the following new save table has been created to handle rows that are no longer valid when you change the partitioning expression for the orders table:

```
CREATE TABLE old_orders
     (o_orderkey INTEGER NOT NULL,
      o_custkey INTEGER,
      o_orderstatus CHAR(1) CASESPECIFIC,
      o_totalprice DECIMAL(13,2) NOT NULL,
      o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      o_orderpriority CHAR(21),
      o_clerk CHAR(16),
      o_shippriority INTEGER,
      o_comment VARCHAR(79))
   UNIQUE PRIMARY INDEX (o_orderkey);
```

The following ALTER TABLE statement is valid if there are 0 or more rows with o_orderdate between January 1, 1992 and December 31, 1992 (DATE '1992-01-01 AND DATE '1992-12-31').

The rows having those values, if any, are moved into the old_orders save table and then deleted from orders:

```
ALTER TABLE Orders
MODIFY PRIMARY INDEX
  DROP RANGE BETWEEN DATE '1992-01-01' AND DATE '1992-12-31'
                     EACH INTERVAL '1' MONTH
  ADD RANGE BETWEEN DATE '1999-01-01' AND DATE '2000-12-31'
                     EACH INTERVAL '1' MONTH
WITH INSERT INTO old_orders;
```

Example 43: Adding a Name to a Previously Unnamed Primary Index

To name the primary index of the Orders table defined in "Example 35: CREATE TABLE Statement for Examples 35 - 42" on page 1-60, you could perform the following statement:

```
ALTER TABLE orders
MODIFY PRIMARY INDEX OrdPI;
```

Example 44: Revalidating the Partitioning for a PPI Table

Suppose any of the following events make you think that incorrect partitioning of the Orders table has occurred:

- You suspect that row partitioning might not be correct after a restore.
- You copy the table to a system with different hardware or operating system.
- Some sort of system malfunction has occurred.

You can validate the partitioning of Orders table rows using either of the following ALTER TABLE statements, with the first statement deleting any problematic rows and the second moving them into a new table:

```
ALTER TABLE orders
REVALIDATE PRIMARY INDEX WITH DELETE;

ALTER TABLE orders
REVALIDATE PRIMARY INDEX WITH INSERT INTO old_orders;
```

Example 45: Unmatched Partitioning Expression Ranges

Assume that a table exists with the following definition.

```
CREATE TABLE cust_orderdate
   (o_custkey INTEGER,
    o_orderdate DATE)
 PRIMARY INDEX (o_custkey)
  PARTITION BY RANGE_N(
   o_orderdate BETWEEN DATE '2000-01-01' AND DATE '2000-12-31'
                       EACH INTERVAL '1' MONTH;
```

The following ALTER TABLE statement returns an error because the expanded partitioning expression ranges defined in the PARTITION BY clause of the CREATE TABLE statement do not match the expanded partitioning ranges defined by the ALTER TABLE specification.

```
ALTER TABLE cust_orderdate
MODIFY PRIMARY INDEX
  DROP RANGE BETWEEN DATE '2000-01-01' AND DATE '2000-04-30';
```

The current table, after expansion of the EACH clause, has 12 ranges, each with a width of one month. The DROP RANGE clause of this ALTER TABLE statement, as written, attempts to drop a single range with a width of four months, and that range does not match any of the ranges currently defined for cust_orderdate.

The two following ALTER TABLE statements produce the result intended by the previous non-valid ALTER TABLE statement, each using a different syntax.

```
ALTER TABLE cust_orderdate
MODIFY PRIMARY INDEX
  DROP RANGE BETWEEN DATE '2000-01-01' AND DATE '2000-04-30'
                     EACH INTERVAL '1' MONTH;


ALTER TABLE cust_orderdate
MODIFY PRIMARY INDEX
  DROP RANGE WHERE PARTITION <=4;
```

ALTER TRIGGER

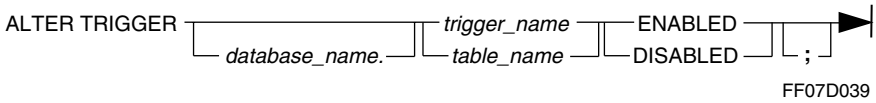
Purpose

Enables or disables a trigger.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
database_name	the name of a qualifying database for the trigger to be altered.
trigger_name	the name of the trigger to be enabled or disabled. Using the *trigger_name* means that you can enable or disable only one trigger on a table at a time.
table_name	the name of the table for which all triggers are to be enabled or disabled.
ENABLED	a keyword meaning that one or all triggers on the object defined are to be enabled. Enabled triggers follow normal trigger protocol.
DISABLED	a keyword meaning that one or all triggers on the object defined are to be disabled. Disabled triggers never fire; in order for a trigger to fire, it must first be enabled.

ANSI Compliance

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

Privileges

To perform ALTER TRIGGER, you must have the DROP TRIGGER privilege on the target object.

Default Behavior

If you enter a table name instead of a trigger name, then ALTER TRIGGER enables or disables all of the triggers defined on that table.

BEGIN LOGGING

Purpose

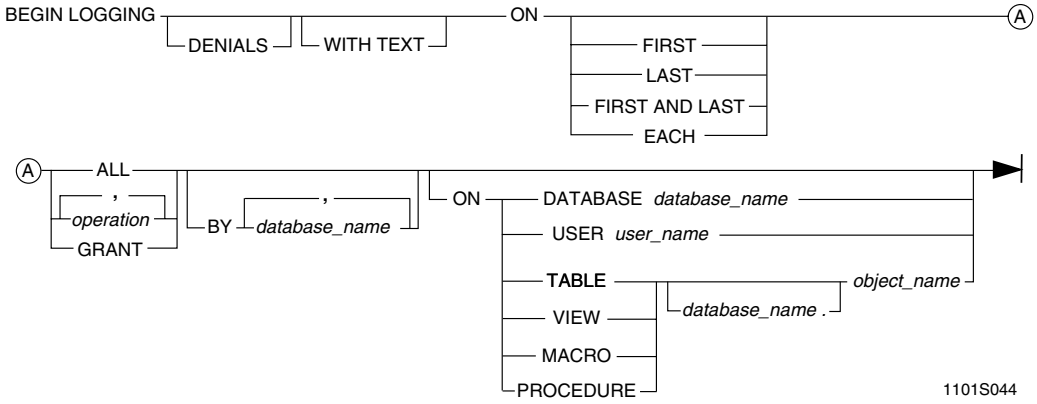
Starts the auditing of SQL requests that attempt to access data.

Also see "END LOGGING" on page 1-347.

Invocation

Executable.

Syntax



where:

Syntax Element ...	Specifies ...
DENIALS	that entries should be made only when statement execution fails because the user did not have the privilege(s) necessary to perform the statement. DENIALS is applied to only those actions listed in the BEGIN LOGGING statement that contains it. For example, two BEGIN LOGGING statements can specify the same object, user, and action, but different frequency and DENIALS options. This allows the user to log all denials, but only the first successful use of a privilege. If this option is not specified, a log entry is made if the privilege check either fails or succeeds.

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>WITH TEXT</td><td>that the text of the statement which caused the log entry is to be saved in the log.

If two BEGIN LOGGING statements specify the same user, object, and action, and one of the statements specifies the DENIALS option, then WITH TEXT can be specified on either one of the statements, on both statements, or on neither statement.

<table>
<tr><th>IF WITH TEXT is specified …</th><th>THEN the text is …</th></tr>
<tr><td>only on the statement that also specifies DENIALS</td><td>saved only when the entry is created as a result of a denial.</td></tr>
<tr><td>either without DENIALS or on both statements</td><td>always saved.</td></tr>
<tr><td>on END LOGGING</td><td>not saved for the specified actions or the DENIALS specification, but the logging frequency is not affected.</td></tr>
</table>
</td></tr>
<tr><td>FIRST
LAST
FIRST AND LAST
EACH</td><td>the frequency with which to log events.

The default is FIRST.

The only valid combination is FIRST AND LAST.

A log entry is made for either the first, the last, the first and last, or each time during any session that the specified action is attempted against the specified object.

If logging is already begun for an action on an object, a subsequent statement to begin logging for the same object, action, user, and DENIALS specification causes only the current frequency of logging to be changed to whatever the new statement specifies.</td></tr>
<tr><td>ALL
operation
GRANT</td><td>the events to record in log entries.

ALL specifies that a log entry will potentially be made when any of the actions in the following list is attempted against the specified object. ALL does not include the CREATE ROLE, DROP ROLE, CREATE PROFILE, or DROP PROFILE actions, because they do not apply to a specific object.

Note that the column name and ID are displayed for each logged column when you log privileges to UPDATE and REFERENCES.

If logging has begun for ALL actions on an object, a statement to begin or end logging for an action will change logging activity only for the specified action.</td></tr>
</table>

Syntax Element …	Specifies …
ALL *operation* GRANT (continued)	One or more action names from the following list can be used to specify the type of access to be logged. These are the same as the functions for which an access privilege can be granted. • CHECKPOINT • CREATE DATABASE • CREATE MACRO • CREATE PROCEDURE • CREATE PROFILE • CREATE ROLE • CREATE TABLE • CREATE TRIGGER • CREATE USER • CREATE VIEW • DELETE • DROP DATABASE • DROP MACRO • DROP PROCEDURE • DROP PROFILE • DROP ROLE • DROP TABLE • DROP TRIGGER • DROP USER • DROP VIEW • DUMP • EXECUTE • EXECUTE PROCEDURE • GRANT • INDEX • INSERT • REFERENCES • RESTORE • ROLLBACK DATABASE • ROLLFORWARD DATABASE • SELECT • UPDATE
BY *database_name*	the community for which log entries should be made. If you specify *user_name*, then it must be a user currently defined to Teradata. Absence of the BY *user_name* option specifies *all* users (those already defined to Teradata as well as any defined in the future while this logging directive is in effect). If neither BY *user_name* nor ON keyword *object_name* are specified, then the specified action is logged for all users and objects throughout Teradata.

Syntax Element …	Specifies …
ON	one or more entities for which log entries are to be made. Absence of the ON keyword *object_name* option specifies logging of all entities the user attempts to access. If neither ON keyword *object_name* nor BY *user_name* is specified, then the specified action is logged for *all* users and objects throughout Teradata. The ON keyword *object_name* option does not apply to the CREATE PROFILE, DROP PROFILE, CREATE ROLE, or DROP ROLE action.
DATABASE *database_name* USER *user_name* TABLE *database_name.table_name* VIEW *database_name.view_name* MACRO *database_name.macro_name* PROCEDURE *database_name.stored_procedure_name*	the name of the object to be logged. The keyword DATABASE, MACRO, PROCEDURE, TABLE, USER, or VIEW must precede each object name. The object name is the name of a database, macro, procedure, table, user, or view on which logging is to be established. You cannot log hash or join indexes. Instead, establish logging on the base tables they belong to. The name of a macro, procedure, table, or view must be qualified by the name of the database or user under which that object is defined.

ANSI Compliance

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

Authorization

You must have the EXECUTE privilege on the DBC.AccLogRule macro.

Permissions

BEGIN LOGGING can only be performed if the DIPACC DBCSQL script, which is provided on the software release tape, has been run to create the DBC.AccLogRule security macro, and if Teradata has been reset to initialize the logging software.

Otherwise, access logging is not allowed.

Only those sites that require access logging should create the DBC.AccLogRule macro. The feature extracts a performance penalty even if little or no logging is performed.

How Logging Works

Each time a user named in a BEGIN LOGGING statement attempts to perform a specified action against a specified object, an entry is logged in the system table DBC.AccLogTbl. Possible entry data includes the type of access, the text of the request, the frequency of access, the action requested, the name of the requesting user, and the referenced objects.

- Logging can be ended on any action, user, or object for which logging is currently active.
- A logging entry does not indicate that a statement was performed; rather, it indicates that Teradata checked the privileges necessary to perform the statement.

Access Logging on Hash and Join Indexes

Because neither hash indexes nor join indexes cannot be accessed directly, you cannot specify access logging for either type of object. If you need to perform access logging, specify it for the relevant base table.

When a hash or join index is used in a query plan, the access privileges are checked for the base table and logged appropriately.

Object Limits

Up to 20 objects can be in a BEGIN LOGGING statement. If more than 20 objects are required, use two or more BEGIN LOGGING statements.

If more or less logging is desired, the full function of the BEGIN LOGGING statement can be used. Also, logging can be ended on any action, user, or object for which logging is currently active.

Changing Options: No Access Privileges If MODIFY USER Is Used

You can enter a self-referent MODIFY USER statement.

In such a case, no access privileges are required in order to change the following options:

- BEFORE JOURNAL
- AFTER JOURNAL
- DEFAULT JOURNAL TABLE
- PASSWORD
- STARTUP
- COLLATION
- DEFAULT DATABASE

Logging is triggered by the use of access privileges. Therefore, no logging occurs if you enter a self-referent MODIFY USER statement.

BEGIN LOGGING Generates Entries in DBC.AccLogTbl

The rules resulting from the execution of BEGIN LOGGING statements are entered in the system table DBC.AccLogRuleTbl. When logging is begun, the specified privilege checks performed by Teradata generates entries in system table DBC.AccLogTbl. The contents of these tables can be monitored via the system views DBC.AccLogRules and DBC.AccessLog.

Aged log entries can be purged using a DELETE statement on the view DBC.DeleteAccessLog.

Privilege checks generate log entries. MODIFY is not a privilege that can be granted, so MODIFY is not a privilege that is checked. MODIFY operations are not logged.

To log MODIFY operations, select logging for the type of access that the MODIFY will or might require.

Specifying such actions as DROP, INSERT, DELETE, and UPDATE can cause logging on actions used by a MODIFY.

For example, to log MODIFY DATABASE operations, specify logging on the DROP DATABASE privilege.

DROP DATABASE is the documented access privilege requirement for MODIFY DATABASE.

ALL, Operation, or GRANT Actions with Access Privileges

Rules related to access privileges are the following:

- CREATE DATABASE/MACRO/PROCEDURE/TABLE/VIEW/USER and DROP DATABASE/USER are allowed only on databases or users.
- DROP TABLE includes ALTER TABLE.

  DROP MACRO, PROCEDURE, or VIEW includes REPLACE MACRO, PROCEDURE, or VIEW.
- Only DROP, EXECUTE, and GRANT are allowed on macros and stored procedures.
- DUMP, RESTORE, and CHECKPOINT are not allowed on views.
- DATABASE, TABLE, VIEW, MACRO, PROCEDURE, and USER confer both CREATE and DROP privileges.
- Role and profile access privileges do not apply to specific objects. Do not use the ON keyword *object_name* option if you specify a CREATE PROFILE, CREATE ROLE, DROP PROFILE, or DROP ROLE action.

Log Entry Generated If Logging Rule Present

A log entry is generated only if a logging rule is present for the object or for the user whose privilege is being checked. Log entries can contain two names.

Object Level Used in the Logging Statement

When a logging statement specifies logging at the database level, actions against all the tables in that database are candidates for log entries. Also, the object level used in granting an access privilege should be considered when specifying the object level of a logging statement.

Scenario A

Assume that DatabaseA contains several tables, and that INSERT privilege has been granted to PUBLIC on Table1 only. Assume also that a current statement specifies "BEGIN LOGGING ON FIRST INSERT ON DatabaseA". Subsequently, if users other than the owner submit insert statements against tables in DatabaseA, the log entries are:

- A granted entry for the first insert into Table1
- A denied entry for the first attempt at an insert into any other table in the database

When logging is specified at the database level, the same type of action against separate tables in that database might not be logged as separate actions.

Scenario B

Assume that DatabaseA contains Table1 and Table2, and that a current BEGIN LOGGING statement specifies logging on FIRST INSERT for DatabaseA. Next, assume that rows were inserted into both Table1 and Table2 in the same session. The logging result would be a single entry identifying the table that was the object of the first insert.

If the access privilege is at the database level and the logging specification is at the table level, only actions against the table are considered for logging entries (the absence of an access privilege at the table level does not necessarily result in a log entry).

A single log entry is generated for the table according to the results of privilege checks at both the table level and the database level.

IF ...	THEN ...
either check succeeds	a granted entry is generated.
neither check succeeds	a denied entry is generated.
access logging is specified for a data object (for example, a table)	log entries are generated whether that object is accessed by name or through a view or macro.

Scenario C

A logging statement specifying FIRST SELECT ON DatabaseA.Table1 causes a log entry to be generated only if an access statement is directly on the table.

```
SELECT ... FROM Table_1
```

The following actions do not log access on DatabaseA.Table1 because the user does not require direct access to Table 1 to perform these statements.

- `SELECT … FROM View1_of_Table_1`
- `EXECUTE MACRO_1`

  where Macro1 contains the statement `SELECT … FROM Table1`.

Users can type a MODIFY USER statement that references themselves. In such a case, no access privileges are required to change the following options:

- BEFORE JOURNAL
- AFTER JOURNAL
- DEFAULT JOURNAL TABLE
- PASSWORD
- STARTUP
- COLLATION
- DEFAULT DATABASE.

Logging is triggered by the use of access privileges. Therefore, no logging will take place if the user enters a MODIFY USER statement that only changes one or more of these options for the user himself or herself.

Rules for Using DDL Statements

BEGIN LOGGING statements must be submitted following the rules for DDL statement usage.

- A DDL statement cannot be combined with other statements as part of a multistatement request.
- A DDL statement can only be included in an explicit transaction (one or more requests bracketed by BEGIN/END TRANSACTION statements) if it is one of the following:
  - the only statement in the transaction, or
  - structured as a single-statement request that appears as the last request in the transaction.

When a BEGIN LOGGING statement is submitted, Teradata checks that the executing user has EXECUTE privilege on the special system macro associated with that statement (see "Permissions" on page 1-70). No checks are made for privileges on the user or objects identified in the LOGGING statement.

If the statement cannot be performed, an error message is returned to the user. Possible errors could be an invalid action name, user name, or object name.

Access Logging Must Be Disabled To Run INITIATE INDEX ANALYSIS

You must disable access logging to perform an INITIATE INDEX ANALYSIS or RESTART INDEX ANALYSIS statement.

Once the index analysis completes, you can enable access logging once again.

Storage of Log Entries

Logging entries are stored in the system table DBC.AccLogTbl. The view DBC.AccessLog offers access to the contents of this table.

If BEGIN LOGGING statements are not submitted, the system default is not to generate any entries on any user action.

See *Teradata RDBMS Security Administration* for more detail on using the BEGIN LOGGING and END LOGGING statements.

Example

The following example illustrates the use of BEGIN LOGGING:

```
BEGIN LOGGING WITH TEXT ON EACH USER, DATABASE, GRANT;
```

For More Information

See "END LOGGING" on page 1-347.

COMMENT

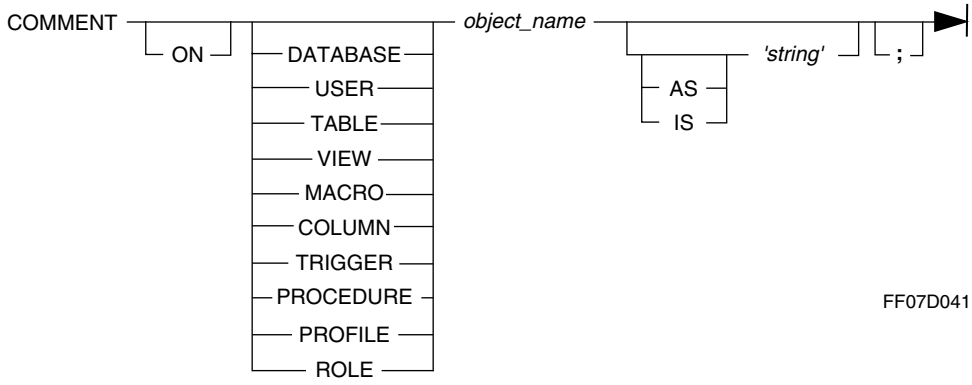
Purpose

Creates or returns a user-defined description of a user-defined object (view, table, hash index, join index, trigger, stored procedure, role, profile, or column) in the data dictionary.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
DATABASE USER TABLE VIEW MACRO COLUMN TRIGGER PROCEDURE PROFILE ROLE	an optional keyword that describes the type of object for which the comment is stored.

Syntax Element …	Specifies …
object_name	the name of the object. A column can be a table column, a view column, a join index column, a hash index column, or a parameter specified in a macro or stored procedure. If an optional keyword does not precede the object name, the object is deduced from the level of qualification in the name. That is, the unqualified name x implies COMMENT ON DATABASE x; the name x.y implies COMMENT ON TABLE, VIEW, PROCEDURE, or MACRO y; and the name x.y.z implies COMMENT ON COLUMN (or parameter specified in a macro or stored procedure) z. The object type is not checked. Use the fully qualified name if there is any ambiguity.
AS IS	an optional keyword that introduces a clause defining a comment string.
'string'	the commentary on or description of the object name. Maximum string length is 255 characters from any supported client character sets. An existing string can be changed by specifying a new string. If no string is specified, any string previously stored is returned.

ANSI Compliance

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

Authorization

You must have the DROP privilege on the object. If the object is a column, the DROP privilege must be on the table that contains the column.

No access privilege is needed to return a comment.

Rules for Using COMMENT

COMMENT is treated as a DDL statement, so it cannot be used in 2PC mode.

The COMMENT statement is typically used in conjunction with a CREATE statement (except for CREATE INDEX statement), to document a newly created object.

Depending on whether a string is defined (with or without an AS or IS keyword), entering a COMMENT statement produces the following results:

- If a string is specified, it replaces a previous string in the data dictionary (provided the user who specifies the string has the proper DROP privilege).
- If a string is not specified, the previous string (or, if no previous string is stored, a null) is returned.

Japanese Characters

On Japanese language sites, comments can contain single byte characters and/or multibyte characters from KanjiEBCDIC, KanjiEUC, or KanjiShift-JIS character sets.

For example, the following comment contains single byte and multibyte characters.

```
THIS COMMENT IS USED FOR TABLE TAB日本語のコメント
```

Examples

The following examples illustrate the use of COMMENT.

Example 1

The following statement can be used to specify a description of the Name column in the Employee table:

```
COMMENT ON COLUMN Employee.Name
  IS 'Employee name, last name followed by first initial';
```

Example 2

Given the following statement:

```
COMMENT ON COLUMN Employee.Name;
```

Teradata returns:

```
Employee name, last name followed by first initial
```

Example 3

Suppose you have a hash index named OrdHIdx defined on the Orders table in the Accounting database.

You could type the following statement to define a comment on the hash index OrdHIdx.

```
COMMENT Accounting.OrdHIdx AS 'hash index on Orders';
```

Then if you were to perform the following statement, the text that follows would be returned to the requestor.

```
COMMENT Accounting.OrdHIdx;

hash index on Orders
```

CREATE DATABASE

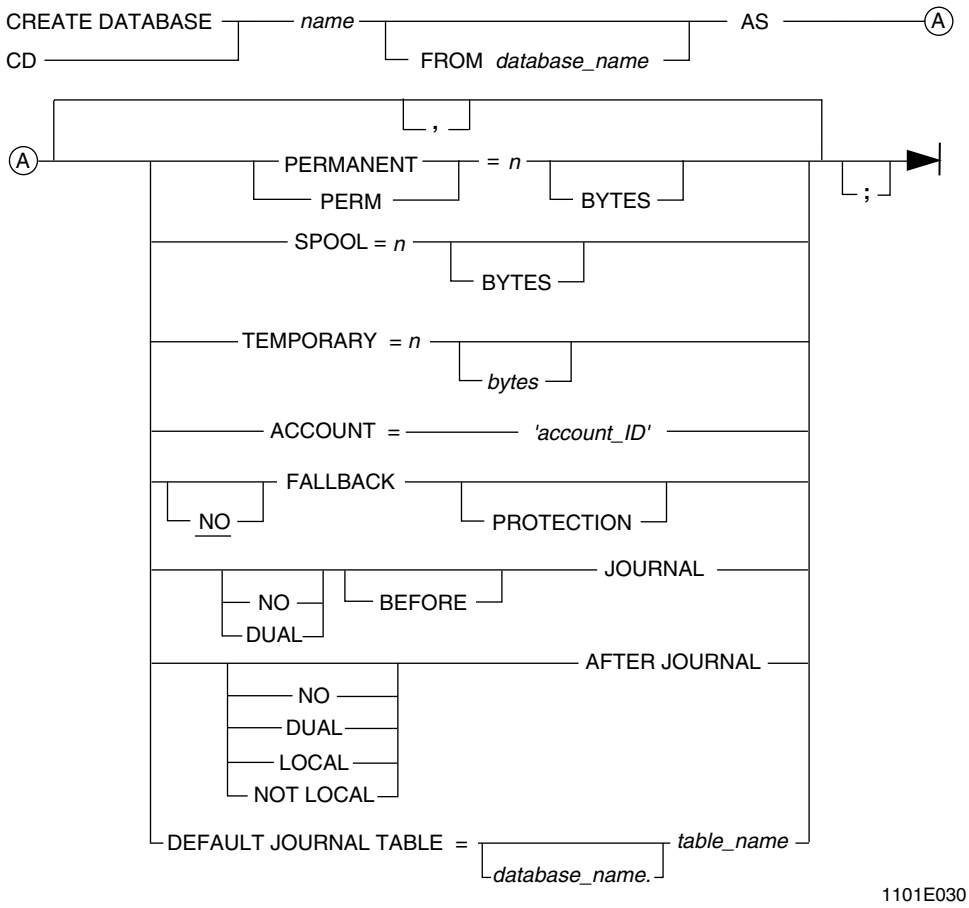
Purpose

Creates a database in which other objects can be created.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
name	the name of the new database.
database_name	the name of the immediate owning user or database. The default is the user name associated with the current session.

Syntax Element …	Specifies …
PERMANENT = *n* BYTES	the total number of bytes to be reserved for permanent storage of the new database. The space is taken from unallocated space in the database of the immediate owner. The number of bytes can be entered as an integer, decimal, or floating point value. This option must be specified. There is no default.
SPOOL = *n* BYTES	the number of bytes to be allocated for spool and volatile temporary files. The default is the largest value that is not greater then the owner spool space and that is a multiple of the number of AMPs on the system. *n* must not exceed the owner spool space. The number of bytes can be entered as an integer, decimal, or floating point value.
TEMPORARY = *n*	a definition for how much space (in bytes) is to be allocated by default for creating temporary tables by users within this database. Note that temporary space is reserved prior to spool space for any user defined with this characteristic. Disk usage for a materialized temporary table is charged to the temporary space allocation of the user who referenced the table. The number of bytes can be entered as an integer, decimal, or floating point value. If no default temporary space is defined for a database, then the space allocated for any temporary tables created in that database is set to the maximum temporary space allocated for its immediate owner.
ACCOUNT = *'account_ID'*	the account to be charged for the space used by this database. If not specified, it defaults to the account identifier of the immediate owner database. *account_ID* can be up to 30 characters and must be enclosed by APOSTROPHE characters.
FALLBACK PROTECTION NO FALLBACK PROTECTION	whether to create and store a duplicate copy of each table created in the new database. The default value is NO FALLBACK. This setting can be overridden for a particular data table when the table is created (see "CREATE TABLE" on page 1-182). The FALLBACK keyword used alone implies PROTECTION.

Syntax Element …	Specifies …
NO DUAL BEFORE JOURNAL	the number of before change images to be maintained by default for each data table created in the new database. The JOURNAL keyword without NO or DUAL implies single copy journaling. If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection. The JOURNAL keyword without BEFORE implies both types (BEFORE and AFTER) of images.
NO DUAL LOCAL NOT LOCAL AFTER JOURNAL	the type of image to be maintained by default for data tables created in the new database. If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection. NOT LOCAL and LOCAL specify whether single after-image journal rows for non-fallback data tables are written on the same virtual AMP (LOCAL) as the changed data rows, or on another virtual AMP in the cluster (NOT LOCAL). See also "Local Journaling" on page 1-83. The JOURNAL keyword without AFTER implies both types (BEFORE and AFTER) of images. The default is no journaling. If only AFTER JOURNAL is specified, then a before change image is not maintained. If both types are specified, the two specifications must not conflict. This setting can be overridden for a particular data table when the table is created (see "CREATE TABLE" on page 1-182).
DEFAULT JOURNAL TABLE = *database_name.table_name*	the default table that is to receive the journal images of data tables created in the new database. *table_name* must be defined if journaling is requested, but it need not reside in the new database. *table_name* is automatically created in the new database if *database_name* is not specified, or the new database is specified. If a different database is specified, then that database must exist and *table_name* must have been defined as its default journal table.

ANSI Compliance

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

CREATE DATABASE is functionally equivalent to the ANSI CREATE SCHEMA statement.

Authorization

The privileges granted to the creator and owner on the new database are WITH GRANT OPTION.

You must have the CREATE DATABASE privilege on the immediate owner database. The creator and owner receive all privileges on the newly created database.

The newly created database also receives all relevant privileges on itself.

No privilege is needed to specify a DEFAULT JOURNAL TABLE in a database other than the database being created. However, when a table is created that specifies a journal, then the user creating the table must have INSERT privilege on the journal table.

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.
- A database can contain only one journal table. However, any table in a particular database can use a journal table in a different database.
- When a database contains a 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 performed, Teradata places an EXCLUSIVE lock on the database being created.
- You can only specify options once.

Local Journaling

The LOCAL single AFTER image journal is supported analogously to single BEFORE image journal:

The access privileges required to create or drop LOCAL single AFTER image journal are the same as for the analogous operation on single BEFORE image journal.

LOCAL single AFTER image journaling is restricted to non-fallback data tables.

Archive/Recovery rules for LOCAL single AFTER image journal and 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.

MultiLoad, Teradata Warehouse Builder, FastLoad and Archive/Recovery are impacted by the use of LOCAL journaling.

Also see *Teradata RDBMS Database Design*.

Examples

The following examples illustrate the use of CREATE DATABASE:

Example 1

The following statement creates a database named Personnel from database Administration:

```
CREATE DATABASE Personnel FROM Administration
AS PERMANENT = 5000000 BYTES,
  FALLBACK,
  BEFORE JOURNAL, DUAL AFTER JOURNAL,
  DEFAULT JOURNAL TABLE = Personnel.FinCopy;
```

The FALLBACK keyword specifies that for each table created in the Personnel database, the default is to store a secondary (duplicate) copy in addition to the primary copy.

The JOURNAL option specifies that the default journaling for each data table is to maintain a single copy of the before change image and dual copies of the after change image. A duplicate before change image is maintained automatically for any table in this database that uses both the fallback and the journal defaults.

The DEFAULT JOURNAL TABLE clause is required because journaling is requested. This clause specifies that a new journal table named “FinCopy” is to be created in the new database.

Example 2

To create a new database for the Finance department, use the CREATE DATABASE statement as follow.

```
CREATE DATABASE finance FROM sysadmin
AS PERMANENT = 60000000
  ,SPOOL = 120000000
  ,FALLBACK PROTECTION
  ,AFTER JOURNAL
  ,BEFORE JOURNAL
  ,DEFAULT JOURNAL TABLE = finance.journals
  ,ACCOUNT = 'ACCTG';
```

The Finance database is created from the space available in Sysadmin. The 60 000 000 value represents that many bytes of storage. To create a database, the initiator must have CREATE DATABASE privileges on the FROM entry. The new database receives all privileges that have been granted to the initiator.

In the example, the FROM clause allocates space for finance from the Sysadmin space rather than from user space; therefore, Sysadmin is the immediate owner of Finance, and Marks is not in the hierarchy of ownership. However, Marks is granted automatic creator privileges on Finance, which include the privilege to create other objects in the space owned by Finance.

The CREATE DATABASE statement does not include the PASSWORD and STARTUP clauses or the DEFAULT DATABASE and COLLATION options. Because these clauses and options affect the session environment, and only a user can establish a session, they do not apply to a database.

CREATE HASH INDEX

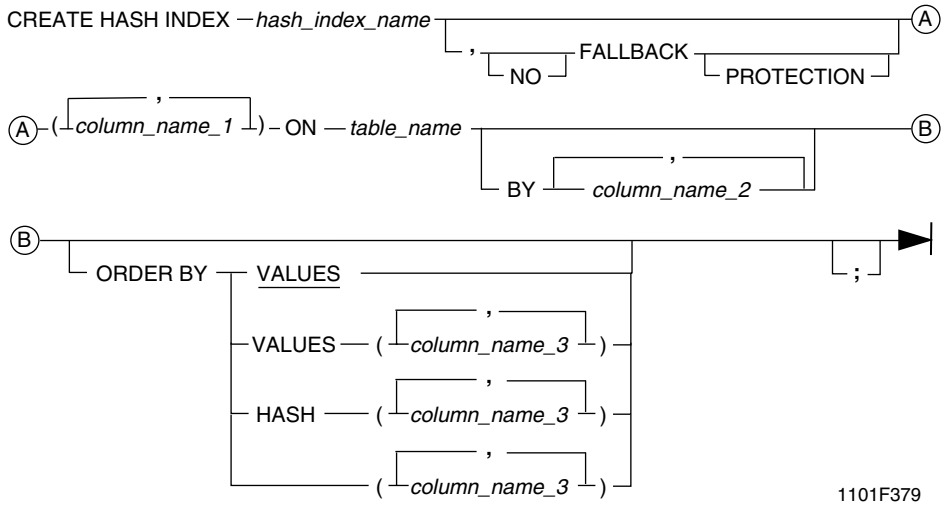
Purpose

Creates a new hash index on a base table.

Invocation

Executable.

Syntax



where:

Syntax element …	Specifies …
hash_index_name	the name given to the hash index created by this statement. Hash index names conform to the rules for naming tables, including explicit qualification with a database or user name. A hash index need not be defined in the same database as the base table represented in its definition.
FALLBACK [PROTECTION]	that the hash index uses fallback protection. The default if you do not make a specification is to use the fallback specification for the database in which the hash index is defined. PROTECTION is an optional noise keyword that has no effect on the hash index definition.

Syntax element …	Specifies …
NO FALLBACK [PROTECTION]	that the hash index does not use fallback protection. The default if you do not make a specification is to use the fallback specification for the database in which the hash index is defined. PROTECTION is an optional noise keyword that has no effect on the hash index definition. Note that if a hash index is not fallback protected, an AMP failure prevents the following events from occurring. • Processing queries on the hash index • Updating the table on which the hash index is defined
column_name_1	the name of a column to be contained in the hash index. All columns defined in the *column_name_1* list must be drawn from the base table on which the hash index is defined. You cannot qualify the columns in the *column_name_1* list. If two specified columns have the same name, both names must be aliased. See “Standard Form for Data in the Teradata RDBMS” in the chapter “SQL Data Handling Fundamentals” in *Teradata RDBMS SQL Reference, Volume 1*. The maximum number of columns you can specify, including the columns implicitly added to the hash index when it is created, is 64. Just as it does for secondary index definitions, Teradata extends the specified *column_name_1* list transparently with additional elements that make it possible for the hash index to provide an access path to corresponding rows in the base table on which it is defined.
table_name	the name of the base table on which the hash index is defined. You *cannot* define a hash index on temporary or volatile tables, views, macros, trigger, stored procedure, journal table, or join or hash index.

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>column_name_2</td><td>the column set on which the hash index rows are partitioned.

The column set you specify must be drawn from the columns specified by column_name_1.

You cannot specify the same column twice in the column_name_2 list.

You cannot qualify any of the column names in the column_name_2 list.

This column set acts as the primary index for the hash index: its rows are hashed on the column set that you specify.

If you do not specify this column set, then the hash index rows are hashed on the primary index column set of the base table on which the index is defined, which makes it hash-local with respect to its base table.

You cannot specify a partitioned column set. This means that you cannot create a PPI for hash indexes.</td></tr>
<tr><td>ORDER BY</td><td>row ordering on each AMP by a single NUSI column: either value-ordered or hash-ordered.

If you do not specify an ORDER BY clause, then the hash index rows have the same ordering as that specified for the base table on which the index is defined.

You must specify a full ORDER BY clause, including an explicit column list, for a hash index defined on a partitioned primary index table.

Rules for using an ORDER BY clause are shown in the following two tables.

<table>
<tr><th>IF you …</th><th>THEN …</th></tr>
<tr><td>specify a BY clause</td><td>the columns specified for the column_name_2 list must be a subset of the column_name_1 list.</td></tr>
<tr><td>specify a BY clause and the base table is hash-ordered</td><td>you must specify an ORDER BY clause for the hash index definition.</td></tr>
<tr><td>do not specify a BY clause</td><td>the hash index rows are distributed by the hash of the primary index column set of the base table on which they are defined. As a result, the hash index is AMP-local.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>ORDER BY
(continued)</td><td>
<table>
<tr><th>IF you specify ORDER BY …</th><th>THEN …</th></tr>
<tr><td>VALUES</td><td>column_name_3 is limited to a single numeric column with a size of four or fewer bytes.

The hash index rows are stored in ascending order by the values of this column.

The following data types are supported for the order key values.
• BYTEINT
• DATE
• DECIMAL
• INTEGER
• SMALLINT</td></tr>
<tr><td>HASH</td><td>• you must specify a BY clause
• the columns specified for column_name_2 and column_name_3 must be identical.

The hash index rows are ordered on the row hash value of these columns.</td></tr>
<tr><td>without specifying the HASH or VALUES keywords</td><td>VALUES is used to order the rows by default.</td></tr>
</table>
</td></tr>
<tr><td>VALUES</td><td>value-ordering for the ORDER BY column.

Select VALUES to optimize queries that return a contiguous range of values, especially for a covering index or a nested join.</td></tr>
<tr><td>HASH</td><td>hash-ordering for the ORDER BY column.

Select HASH to limit hash-ordering to one column, rather than all columns of the primary index (the default).</td></tr>
<tr><td>column_name_3</td><td>the name of the column set on which the index is to be ordered.

If you do not specify a column set, then the default ordering is the same as specified for the base table.</td></tr>
</table>

ANSI Compliance

CREATE HASH INDEX is a Teradata extension to the ANSI SQL-99 standard. ANSI does not define DDL for creating indexes.

Authorization

You must have the CREATE TABLE privilege on the database in which the hash index is created.

You must have the INDEX or DROP TABLE privilege on the indexed base table or its containing database.

The creator of a hash index is granted the following table privileges on the index.

- Drop table
- Index
- Dump
- Restore

Comparison of Hash and Single-Table Join Indexes

The reasons for using hash indexes are similar to those for using single-table join indexes. Not only can hash indexes optionally be specified to be distributed in such a way that their rows are AMP-local with their associated base table rows, they can also provide an alternate direct access path to those base table rows. This facility makes hash indexes somewhat similar to secondary indexes in function. Hash indexes are also useful for covering queries so that the base table need not be accessed at all.

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 Teradata when the relevant columns of their base table are updated by a DELETE, INSERT, or UPDATE statement.
- Both can be the object of any of the following SQL statements.
  - COLLECT STATISTICS (Optimizer Form)
  - DROP STATISTICS
  - HELP INDEX
  - SHOW HASH INDEX
- Both receive their space allocation from permanent space and are stored in distinct tables.
- Both can be compressed.
- Both can be FALLBACK protected.
- Neither can be queried or directly updated.
- Neither can have a partitioned primary index.
- Neither can store an arbitrary query result.
- Both share the same restrictions for use with the MultiLoad, Teradata Warehouse Builder, FastLoad, and Archive/Recovery 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 multiple rows in the referenced base tables.
Column list cannot contain aggregate or ordered analytical functions.	Column list can contain aggregate functions.
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[a] or by including the index columns of the unique[b] primary index or USI on the base table.

a. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement.
b. NUPIs cannot be used for this purpose.

Compression of Hash Index Rows

When the columns of the order key are not unique, the rows of the hash index are compressed. 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.

Hash index compression, unlike join index compression, is not specified in the CREATE statement. Instead, Teradata determines which columns are to be compressed and which are not.

The fixed part of the row is made up of the columns that define the order key. The repeating part is composed of the remaining columns. See "Hash and Join Index Storage" in the chapter "Join and Hash Indexes" in *Teradata RDBMS Database Design* for a more detailed description of this sort of row compression.

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 "Collecting Statistics on Base Table Columns Instead of Hash Index Columns" on page 1-92 for further information.

The Teradata Statistics Wizard client utility provides an objective means for determining which columns you should collect statistics on to support particular workloads. See *Teradata Statistics Wizard User Guide* for further information.

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 OrdHIdx8 (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 OrdHIdx8 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 statements.

```
COLLECT STATISTICS ON OrdHIdx8 COLUMN o_custkey;
COLLECT STATISTICS ON OrdHIdx8 COLUMN o_orderdate;
```

It is always better to collect the statistics for a multicolumn index 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.

For example, a query with a predicate 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.

The Teradata Statistics Wizard utility does not support hash indexes.

Collecting 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.

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 *worse* access plans, then you should always report the incident to NCR support personnel.

When you are considering collecting statistics for a hash 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 hash index must access it with a direct probe, a full table scan, or a range scan. With this in mind, consider the following things when deciding which columns to collect statistics for.

-

IF an execution plan might involve ...	THEN collect statistics on the ...
search condition keys	column set that constitutes the search condition predicate.
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.

-

IF a 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 *column_name_1* 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 clause predicates, you should consider collecting statistics on it as well, particularly if that column is the sort key for a value-ordered hash index.

Hash Indexes Are Maintained by Teradata

Hash indexes are automatically maintained by Teradata when updates (UPDATE, DELETE, 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 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 partitioned primary index. To define a hash index on a PPI table, you must also specify a full ORDER BY clause, including an explicit column list.

If you experience 3710 errors when running queries against a PPI table with a hash index, you can increase the amount of memory that is available to the Optimizer by updating several DBSControl flags to values that ensure that it can process any request on the PPI table.

The recommended initial settings are as follows:

MaxParseTreeSegs = 2000

OptMaxGlobalJI = 8

You might find that your query workloads are such that these values need to be increased still further.

OptMaxGlobalJI is an internal DBSControl field. You must contact a Teradata support representative to change its value.

Restrictions and Limitations for Load Utilities

You cannot use FastLoad, MultiLoad, or Teradata Warehouse Builder 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 either of 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 Teradata Warehouse Builder.

Load utilities like TPump, which perform standard SQL row inserts and updates, *are* supported for hash-indexed tables.

Restrictions and Limitations for Archive and Restore

Though you can archive a database that contains hash indexes, Teradata does not automatically rebuild those indexes when you restore the archived database; instead, they are marked as not valid.

To work around this, you must drop and recreate the hash index and collect its statistics before you can use it in a query.

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.

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.

Triggers Are Not Supported With Hash Indexes

You cannot define triggers and hash indexes on the same table.

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.

Examples for Base Table with Unique Primary Index

The subsequent examples demonstrate hash indexes based on the following table definition.

```
CREATE TABLE Orders
     (o_orderkey      INTEGER       NOT NULL,
      o_custkey       INTEGER,
      o_orderstatus   CHARACTER(1)  CASESPECIFIC,
      o_totalprice    DECIMAL(13,2) NOT NULL,
      o_orderdate     DATE          FORMAT 'yyyy-mm-dd' NOT NULL,
      o_orderpriority CHARACTER(21),
      o_clerk         CHARACTER(16),
      o_shippriority  INTEGER,
      o_comment       VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey);
```

UPI Base Table Example 1

The following hash index is distributed on its primary index o_orderdate and stored in value order on o_orderdate.

```
CREATE HASH INDEX OrdHIdx_1 (o_orderdate) ON Orders
BY (o_orderdate)
ORDER BY (o_orderdate);
```

Equivalently, you could specify ORDER BY VALUES instead of ORDER BY.

Note that the ORDER BY column list complies with the rules that it be limited to a single column, that the column must be in the *column_name_1* list, must be a numeric type, and must be four or fewer bytes.

UPI Base Table Example 2

The following hash index is distributed on its primary index o_custkey and stored in row-hash order on o_custkey.

```
CREATE HASH INDEX OrdHIdx_2 (o_custkey) ON Orders
BY (o_custkey)
ORDER BY HASH (o_custkey);
```

UPI Base Table Example 3

The following hash index is distributed on o_orderkey (the primary index for the base table Orders) and stored in value order on o_orderkey (the order key for the base table Orders).

```
CREATE HASH INDEX OrdHIdx_3 (o_orderdate) ON Orders
ORDER BY VALUES;
```

Note that the ordering column of the base table complies with the rules that it be limited to a single numeric column four or fewer bytes in length.

UPI Base Table Example 4

The following hash index is distributed on o_orderkey (the primary index for the base table Orders) and stored in value order on o_orderkey.

```
CREATE HASH INDEX OrdHIdx_4 (o_orderdate) ON Orders;
```

UPI Base Table Example 5

The following hash index is distributed on o_orderkey (the primary index for the base table Orders) and stored in value order on o_orderdate.

```
CREATE HASH INDEX OrdHIdx_5 (o_orderdate) ON Orders
ORDER BY (o_orderdate);
```

Equivalently, you could specify ORDER BY VALUES instead of ORDER BY.

UPI Base Table Example 6

The following hash index is distributed on its primary index o_orderdate and stored in value order on o_orderkey (the primary index for the base table Orders).

```
CREATE HASH INDEX OrdHIdx_6 (o_orderdate) ON Orders
BY (o_orderdate)
ORDER BY VALUES;
```

Note that the ordering column of the base table complies with the rules that it be limited to a single numeric column four or fewer bytes in length.

Examples for Base Table with Nonunique Primary Index

The subsequent examples demonstrate hash indexes based on the following table definition.

```
CREATE TABLE Lineitem
     (l_orderkey      INTEGER       NOT NULL,
      l_partkey       INTEGER       NOT NULL,
      l_suppkey       INTEGER,
      l_linenumber    INTEGER,
      l_quantity      INTEGER       NOT NULL,
      l_extendedprice DECIMAL(13,2) NOT NULL,
      l_discount      DECIMAL(13,2),
      l_tax           DECIMAL(13,2),
      l_returnflag    CHARACTER(1),
      l_linestatus    CHARACTER(1),
      l_shipdate      DATE          FORMAT 'yyyy-mm-dd',
      l_commitdate    DATE          FORMAT 'yyyy-mm-dd',
      l_receiptdate   DATE          FORMAT 'yyyy-mm-dd',
      l_shipinstruct  VARCHAR(25),
      l_shipmode      VARCHAR(10),
      l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey);
```

NUPI Base Table Example 1

The following hash index is distributed explicitly on l_shipdate and stored explicitly in value order on l_shipdate.

```
CREATE HASH INDEX LineHIdx_1 (l_shipdate) ON Lineitem
BY (l_shipdate)
ORDER BY (l_shipdate);
```

Equivalently, you could specify ORDER BY VALUES instead of ORDER BY.

NUPI Base Table Example 2

The following hash index is distributed explicitly on l_partkey and stored explicitly in row-hash order on l_partkey.

```
CREATE HASH INDEX LineHIdx_2 (l_partkey) ON Lineitem
BY (l_partkey)
ORDER BY HASH (l_partkey);
```

NUPI Base Table Example 3

The following hash index is distributed implicitly on l_orderkey, the primary index for the base table Lineitem, and stored explicitly in value order on l_orderkey.

```
CREATE HASH INDEX LineHIdx_3 (l_shipdate) ON Lineitem
ORDER BY VALUES;
```

The ordering column of the base table complies with the rules that it be limited to a single numeric column four or fewer bytes in length.

NUPI Base Table Example 4

The following hash index is distributed implicitly on l_orderkey, the primary index for the base table Lineitem, and also stored implicitly in value order on l_orderkey.

```
CREATE HASH INDEX LineHIdx_4 (l_shipdate) ON Lineitem;
```

NUPI Base Table Example 5

The following hash index is distributed implicitly on l_orderkey, the primary index for the base table Lineitem, and stored explicitly in value order on l_shipdate.

```
CREATE HASH INDEX LineHIdx_5 (l_shipdate) ON Lineitem
ORDER BY (l_shipdate);
```

Equivalently, you could specify ORDER BY VALUES instead of ORDER BY.

Note that the ORDER BY column list complies with the rules that it be limited to a single column, that the column must be in the *column_name_1* list, must be a numeric type, and must be four or fewer bytes.

NUPI Base Table Example 6

The following hash index is distributed explicitly on l_shipdate and stored implicitly in value order on l_orderkey, the primary index for the base table Lineitem.

```
CREATE HASH INDEX LineHIdx_6 (l_shipdate) ON Lineitem
BY (l_shipdate)
ORDER BY VALUES;
```

Note that the ordering column of the base table complies with the rules that it be limited to a single numeric column four or fewer bytes in length.

Further Information

For an overview of Teradata hash indexes, see “Hash Indexes” in the chapter “SQL Data Definition, Control, and Manipulation” of *Teradata RDBMS SQL Reference, Volume 1* and the chapters “Teradata Indexes: The Basics” and “Join and Hash Indexes” in *Teradata RDBMS Database Design* for more examples and discussion of the functions and features of hash indexes.

CREATE INDEX

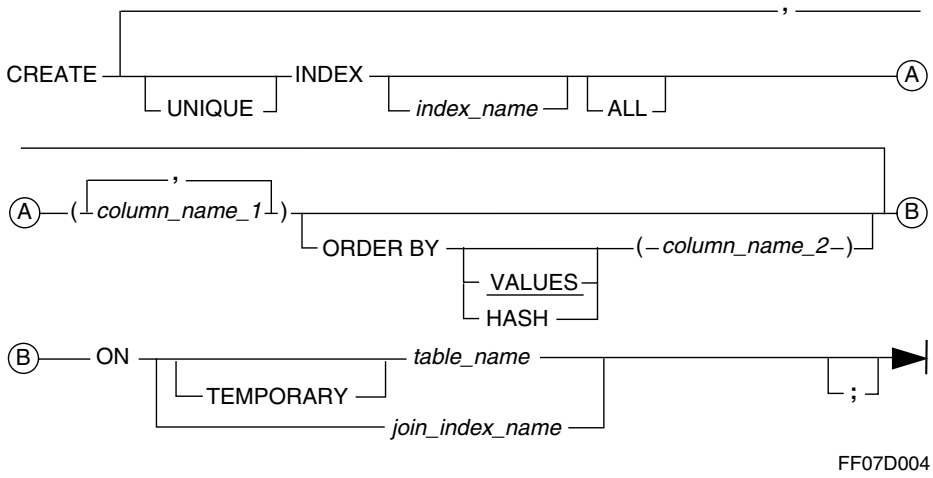
Purpose

Creates a new secondary index on an existing table or join index.

Invocation

Executable.

Syntax



where:

Syntax Element ...	Specifies ...
UNIQUE	that no two rows in the table can have the same value or combination of values in the indexed fields. You cannot specify UNIQUE for an index on a join index. You cannot specify UNIQUE if you use an ORDER BY clause.
index_name	an optional name for the secondary INDEX (30 characters maximum).

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>ALL</td><td>that a NUSI should retain row ID pointers for each logical row of a join index (as opposed to only the compressed physical rows).

ALL also ignores the NOT CASESPECIFIC attribute of data types so a NUSI can include case-specific values.

ALL enables a NUSI to cover a join index, enhancing performance by eliminating the need to access a join index when all values needed by a query are in the secondary index. However, ALL might also require the use of additional index storage space.

Use this keyword when a NUSI is being defined for a join index and you want to make it eligible for the Optimizer to select when covering reduces access plan cost. ALL can also be used for an index on a table, however.

You cannot specify multiple indexes that differ only by the presence or absence of the ALL option.</td></tr>
<tr><td>column_name_1</td><td>the names of one or more columns whose field values are to be indexed.

You can specify up to 64 columns for the new index. The index is based on the combined values of each column. Unless you use the ORDER BY clause, all columns are hash-ordered.

Multiple indexes can be defined on the same columns as long as each index differs in its ordering option (VALUES versus HASH).</td></tr>
<tr><td>ORDER BY</td><td>row ordering on each AMP by a single NUSI column: either value-ordered or hash-ordered.

Rules for using an ORDER BY clause are shown in the following table.

<table>
<tr><th>IF you specify ORDER BY …</th><th>THEN …</th></tr>
<tr><td>with VALUES</td><td>column_name_2 must be numeric with four bytes or less</td></tr>
<tr><td>without HASH or VALUES</td><td>VALUES is assumed.</td></tr>
<tr><td>with a column_name_2</td><td>column_name_2 must be one of the columns in the column_name_1 list.</td></tr>
<tr><td>without a column_name_2</td><td>column_name_2 is assumed to be the first column in the column_name_1 list.</td></tr>
</table>
</td></tr>
<tr><td>VALUES</td><td>value-ordering for the ORDER BY column.

Select VALUES to optimize queries that return a contiguous range of values, especially for a covered index or a nested join.</td></tr>
</table>

Syntax Element …	Specifies …
HASH	hash-ordering for the ORDER BY column. Select HASH to limit hash-ordering to one column, rather than all columns (the default). Hash-ordering a multicolumn NUSI on one of its columns allows the NUSI to participate in a nested join where join conditions involve only that ordering column.
column_name_2	a column in the *column_name_1* list that specifies the sort order to be used. Supported data types for a value-ordered, four-bytes-or-less *column_name_2* are: • DATE • BYTEINT • INTEGER • DECIMAL • SMALLINT
TEMPORARY	that the index is being created on a materialized global temporary table. If you do not specify TEMPORARY then the index is created on the base global temporary table definition. In this case, if there are materialized instances of the table you cannot alter the index on the base global temporary table. If you specify TEMPORARY against a permanent table, Teradata returns an error. If you specify TEMPORARY correctly, but no materialized instance of the definition exists in the system, then Teradata materializes an instance before creating an index for it. You cannot create indexes on a volatile table.
table_name	the table for which an index is to be created.
join_index_name	the name of the join index on which this index is to be defined. You cannot define a USI on a join index. Other secondary indexes are allowed.

ANSI Compliance

CREATE INDEX is a Teradata extension to the ANSI SQL-99 standard. ANSI does not define DDL for creating indexes.

The CREATE UNIQUE INDEX syntax is functionally equivalent to adding a UNIQUE or PRIMARY KEY constraint to a column in ANSI SQL-99.

Authorization

You must have the INDEX or DROP TABLE privilege on the named table or join index.

Locks and Concurrency

If no locking modifier is specified for the table that contains the index columns, or if WRITE is specified, a WRITE lock is placed on the table. If READ, ACCESS, or SHARE is specified, a READ lock is placed on the table.

The READ or WRITE lock is upgraded to EXCLUSIVE after the index subtables are created, but prior to locking the dictionary and modifying the headers. This improves concurrency by reducing the time SELECT statements are blocked.

A CREATE INDEX statement for a table can run concurrently with a SELECT statement that has an ACCESS lock (or READ lock, if READ lock was requested for the CREATE INDEX) on the same table, until the lock is upgraded to EXCLUSIVE; when that occurs, the statement that requests the lock first blocks the other.

A CREATE INDEX statement for a table can run concurrently with a COLLECT STATISTICS statement 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 statement completes. Any subsequent COLLECT STATISTICS are blocked until the CREATE INDEX completes.

A CREATE INDEX statement 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.

Maximum Number of Indexes

Up to 32 secondary indexes can be defined for one CREATE INDEX statement. Also, up to 32 secondary and join indexes can be defined for one table. This permits all index operations for one table to be treated as a single transaction; that is, all of the indexes are created, or none of them are created. Up to 64 columns can be defined for each index.

Alternatively, secondary indexes can be created as part of the CREATE TABLE statement.

Secondary Indexes Require Subtable Creation, Which Takes Space

Secondary indexes require the creation of subtables in which to store index values and associated row IDs. Each secondary index subtable requires additional disk space, and insert, delete, and update operations require more processing time when secondary index subtables also need to be changed.

Secondary Indexes Can Shorten Search Time

Secondary index values can shorten table search time for set (multirow) 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 or rows containing that value. Therefore, when secondary index values are used as conditional expressions in SQL statements, only the row or rows that contain the specified value are accessed.

UNIQUE Versus Nonunique

The UNIQUE option prevents the occurrence of duplicate index values. Only one row can have a value 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.

EXPLAIN and Visual EXPLAIN Help To Analyze Each Secondary Index

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, a full-table scan is used. It is advisable, therefore, to test the performance of a secondary index before implementing it.

You can use either the SQL EXPLAIN modifier (see “EXPLAIN Modifier” in the chapter “SQL Data Manipulation Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 6*) or the Visual EXPLAIN client utility (see *Teradata Visual Explain Getting Started Guide)* to generate a description of the processing sequence to help determine whether a secondary index should be retained. You can also use the Teradata Index Wizard client utility to determine optimum secondary indexes for particular SQL workloads. See *Teradata Index Wizard User Guide* and the chapter “Database Foundations for the Teradata Index Wizard” in *Teradata RDBMS SQL Reference, Volume 2*.

Named Indexes

The index name must conform to the usual rules for a Teradata SQL identifier and be unique in the table.

Also see “DROP INDEX” on page 1-333.

Relationships Between Secondary Indexes and Primary Indexes

You can create a USI on the same column set that defines a nonunique primary index for a table. 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 PPIs 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 any primary index. This is valid for a PPI even if the primary index column set does not include all the columns used in the partitioning expression.

Examples

The following examples illustrate the use of CREATE INDEX.

Example 1

The following statement can be used to create a secondary index on the DeptNo column of the Employee table:

```
CREATE INDEX (DeptNo) ON Employee;
```

The UNIQUE option is not used in this statement because multiple rows will contain the same department number. The index should be useful, however, because department numbers are often used for conditional retrievals and for ordering employee listings.

Example 2: Named Indexes

This example demonstrates a named index.

```
CREATE UNIQUE INDEX Ident (Name, SocSecNo) ON Employee;
```

This definition allows the named index to be dropped by using its name:

```
DROP INDEX Ident ON Employee;
```

Example 3: Creating a Named Unique Secondary Index

This example creates a named unique secondary index *index_1*. Index *index_1* must not be an existing name in table *table_1*.

```
CREATE UNIQUE INDEX index_1 (field_1, field_2) ON table_1;
```

Example 4: Creating a Named Nonunique Secondary Index

This example creates a named nonunique secondary index *index_2*. Index *index_2* must not be an existing name in table *table_2*.

```
CREATE INDEX index_2 (field_1) ON table_2;
```

Example 5: Creating an Unnamed Unique Secondary Index

This example creates an unnamed unique secondary index.

```
CREATE UNIQUE INDEX (field_1, field_2) ON table_3;
```

Example 6: Creating an Unnamed Nonunique Secondary Index

This example creates an unnamed nonunique secondary index.

```
CREATE INDEX (field_1) ON table_4;
```

Example 7: Creating a Secondary Index On Top of a Join Index

This example creates a secondary index on top of the join index defined in the example set for CREATE JOIN INDEX (see "CREATE JOIN INDEX" on page 1-108) and then shows the explanation for a query made against the base tables that uses both indexes.

Note that you cannot create a *unique* secondary index on top of a join index; any such index must be *nonunique*.

```
CREATE INDEX shipidx (l_shipdate) ON OrderJoinLine;

 *** Index has been created.
 *** Total elapsed time was 5 seconds.

EXPLAIN
SELECT o_orderdate, o_custkey, l_partkey, l_quantity,
l_extendedprice
FROM Lineitem, Order
WHERE l_orderkey = o_orderkey
AND l_shipdate = '1997-09-18';

Explanation
------------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     RowHash to prevent global deadlock for LOUISB.OrderJoinLine.
  2) Next, we lock LOUISB.OrderJoinLine for read.
  3) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.OrderJoinLine by way of index # 12
     "LOUISB.OrderJoinLine.l_shipdate = 970918" with a residual
     condition of ("(NOT (LOUISB.OrderJoinLine.o_orderdate IS NULL ))
     AND (LOUISB.OrderJoinLine.l_shipdate = 970918)") into Spool 1,
     which is built locally on the AMPs.  The input table will not be
     cached in memory, but it is eligible for synchronized scanning.
     The result spool file will not be cached in memory.  The size of
     Spool 1 is estimated to be 500 rows.  The estimated time for this
     step is 0.37 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

Example 8: Non-Valid Use of CREATE INDEX

This example demonstrates how *not* to use CREATE INDEX.

```
CREATE UNIQUE INDEX ThisIsAnExtremelyLongNamedIndex
(field_1,field_2) ON table_7;
```

The name for the index exceeds the maximum of 30 characters.

Teradata returns the message:

```
Name is longer than 30 characters.
```

Further Information

For an overview of Teradata secondary indexes, see “Secondary Indexes” in the chapter “SQL Data Definition, Control, and Manipulation” in *Teradata RDBMS SQL Reference, Volume 1* and the chapter "Secondary Indexes," in *Teradata RDBMS Database Design*.

CREATE JOIN INDEX

Purpose

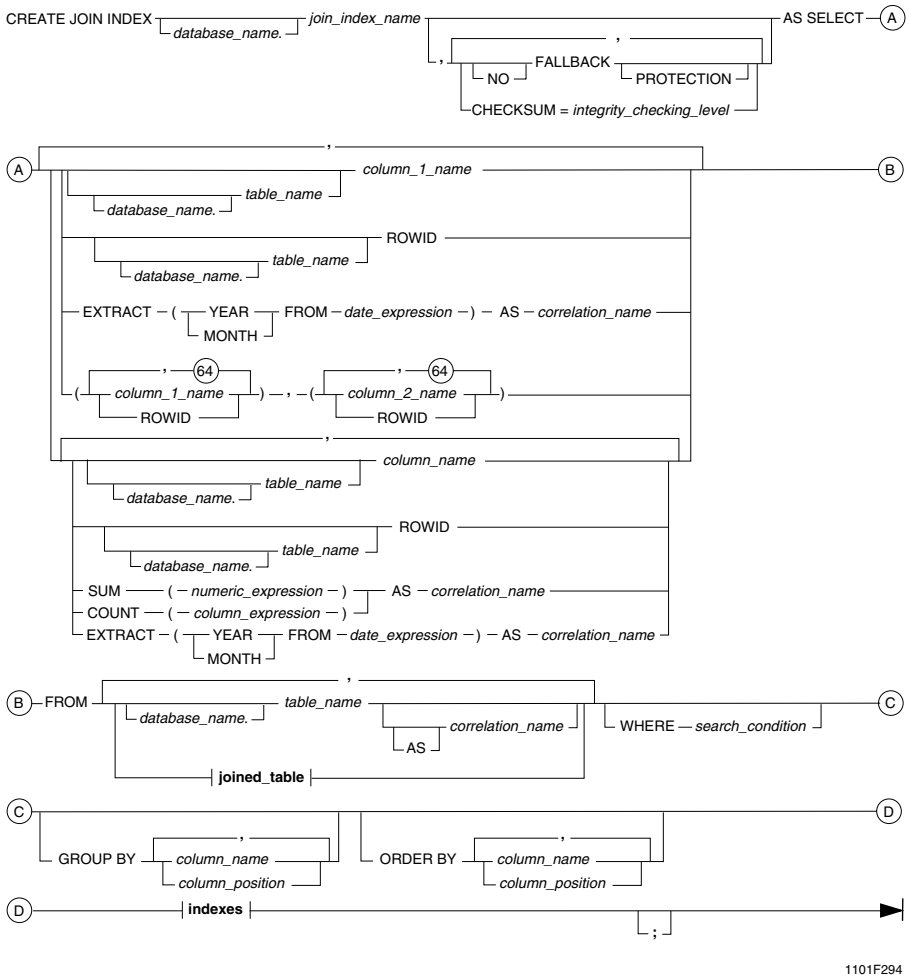
Creates a join index that has one of the following functions.

- Joins multiple tables (optionally with aggregation) in a prejoin table
- Replicates all or a vertical subset of a single base table and partitions its rows with a primary index on a foreign key column to facilitate joins of very large tables by hashing them to the same AMP
- Aggregates one or more columns of a single table or the join results of multiple tables in a summary table

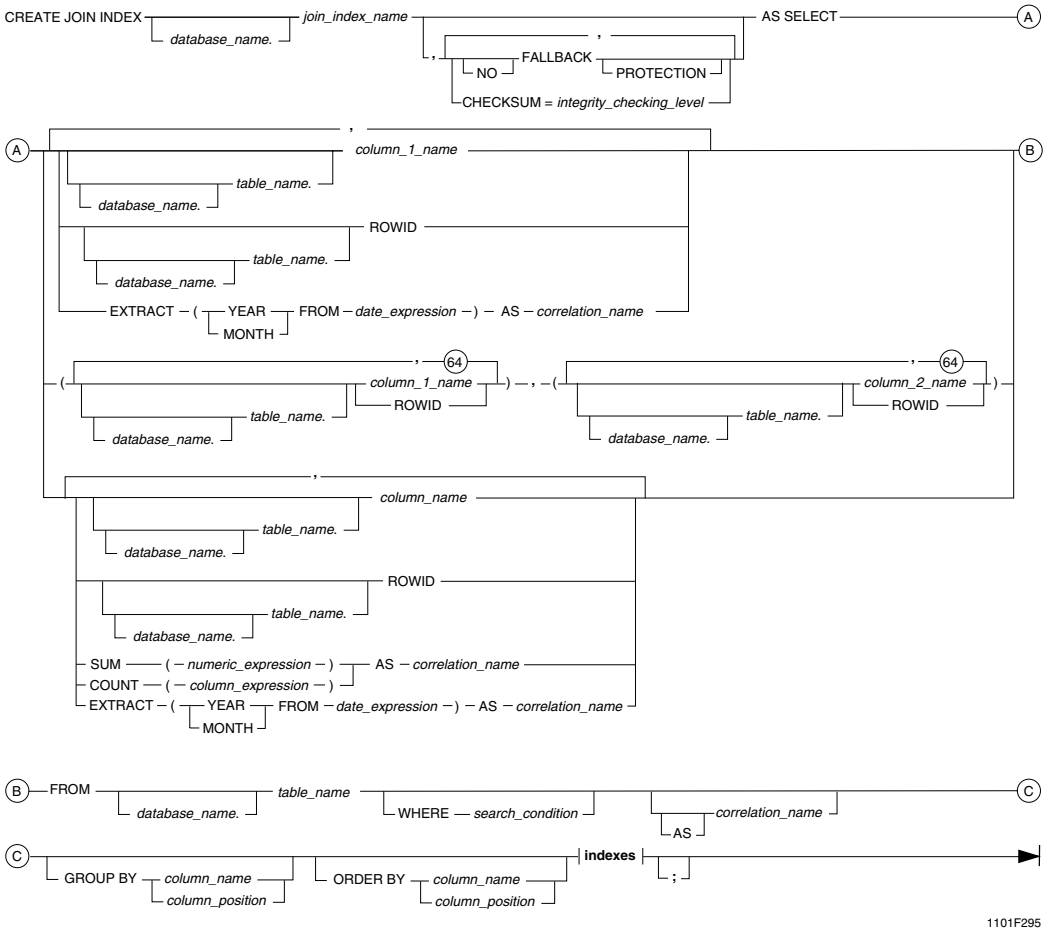
Invocation

Executable.

Multitable Syntax

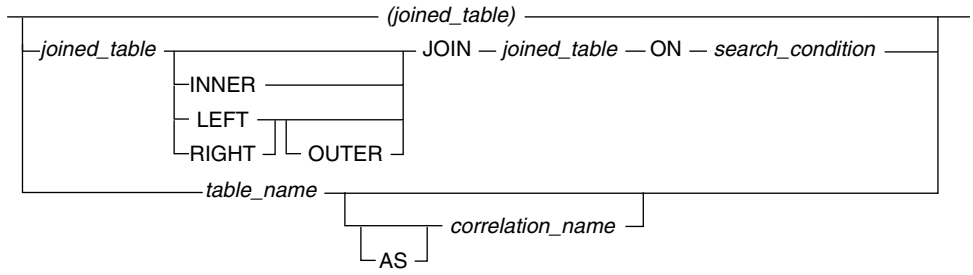


Single-Table Syntax

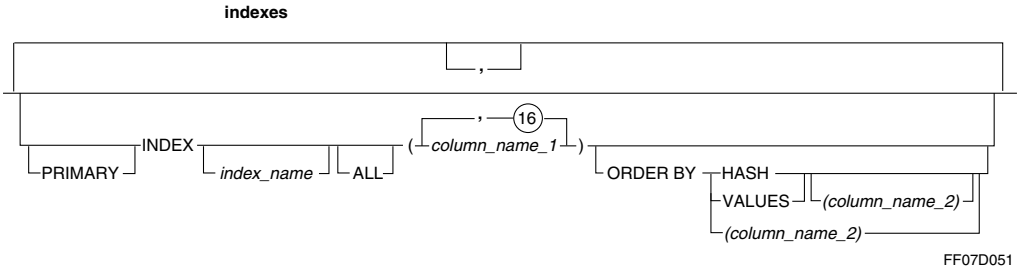


Joined_Table Excerpt

joined_table



Index Excerpt



where:

Syntax Element ...	Specifies ...
join_index_name	the name given to the join index created by this statement. Join index names conform to the rules for naming tables, including explicit qualification with a database or user name. A join index need not be defined in the same database as the base tables represented in its definition.
FALLBACK [PROTECTION]	that the join index uses fallback protection. PROTECTION is an optional noise keyword that has no effect on the join index definition.
NO FALLBACK [PROTECTION]	that the join index does not use fallback protection. This is the default. PROTECTION is an optional noise keyword that has no effect on the join index definition. Note that if a join index is not fallback protected, an AMP failure prevents the following events from occurring. • Processing queries on the join index • Updating the table on which the join index is defined
column_1_name	the name of a column to be included in the join index. If two specified columns have the same name, both names must be aliased. See "Standard Form for Data in the Teradata RDBMS" in Chapter 1: "SQL Data Handling Fundamentals" in *Teradata RDBMS SQL Reference, Volume 1*. Specify the keyword ROWID as one of the values for either the *column_1_name* list or the *column_2_name* list to enable the Optimizer to join a partial covering index to its base table to access any noncovered columns. You can only specify ROWID in the outermost SELECT list 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. If a *column_2_name* is also defined, then each distinct value of *column_1_name* is stored only once. You can specify no more than 64 columns per referenced base table per join index.

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>column_1_name
(continued)</td><td>
<table>
<tr><th>IF column_2_name is …</th><th>THEN the limit on the total number of columns that can be specified for column_1 _name is …</th></tr>
<tr><td>defined</td><td>64.</td></tr>
<tr><td>not defined</td><td>not defined.

The maximum is restricted only by the number of columns that can be defined within the row size limit.

This limit is not related in any way to the restriction of 64 columns that can be defined per referenced base table per join index.</td></tr>
</table>
</td></tr>
<tr><td>EXTRACT</td><td>to use the EXTRACT date function to extract a year or month from a date column.

See “EXTRACT Function” in the chapter ”DateTime and Interval Functions and Expressions” in Teradata RDBMS SQL Reference, Volume 5.

Note that join index definitions support only the extraction of years and months from a date column.</td></tr>
<tr><td>date_expression</td><td>a date column or date expression from which either the year or the month is to be extracted.</td></tr>
<tr><td>correlation_name</td><td>a mandatory unique name for a sum, count, or extracted year or month. This name permits further use of the result as the grouping key for a GROUP BY clause or the ordering key for an ORDER BY clause.</td></tr>
<tr><td>SUM</td><td>to use the SUM aggregate function to compute the sum of the column or column expression specified by numeric_expression.

To avoid overflow, type the SUM field in a join index definition as FLOAT. If you do not assign a data type, Teradata types it as FLOAT automatically. If you assign a type other than FLOAT, an error message occurs.

Note that SUM DISTINCT is not permitted.

If you specify SUM, but not COUNT, then Teradata adds a COUNT(*) expression to the column list and names it CountStar.

Also see “SUM Function” in the chapter “Aggregate Functions” in Teradata RDBMS SQL Reference, Volume 5.</td></tr>
</table>

Syntax Element …	Specifies …
COUNT	to use the COUNT aggregate function to count the number of values in the column or column expression specified by *column_expression*. To avoid overflow, type the COUNT field in a join index definition as FLOAT. If you do not assign a data type, Teradata types it as FLOAT automatically. If you assign a type other than FLOAT, an error message occurs. Note that COUNT DISTINCT is not permitted. Also see "COUNT Function" in the chapter "Aggregate Functions" in *Teradata RDBMS SQL Reference, Volume 5*. Note that all aggregate join indexes must have either a COUNT(*) or a COUNT(*non_null_expression*) in their definition to support index maintenance. If none is specified, Teradata creates a COUNT(*) implicitly, adds it to the definition, and gives it the default name of CountStar.
numeric_expression	an expression having a numeric data type that is to be summed or counted over a specified column.
column_2_name	a list of multiple columns to be included in the join index. Represents the repeated portion of the join index row. If two specified columns have the same name, both names must be aliased. See "Standard Form for Data in the Teradata RDBMS" in Chapter 1: "SQL Data Handling Fundamentals" in *Teradata RDBMS SQL Reference, Volume 1*. You can specify the keyword ROWID as one of the values for *column_2_name* to enable the Optimizer to join a partial covering index to its base table to access any noncovered columns. Do not duplicate the ROWID specification if you have already defined ROWID as one of the *column_1_name* list values. You can only specify ROWID in the outermost SELECT list 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. Do *not* specify *column_2_name* values for aggregate join indexes. You can specify no more than 64 columns for *column_2_name*. If you specify *column_2_name*, then you can specify no more than 64 columns for *column_1_name* because the total column limit for a compressed join index is 128 columns. There is no limit for the number of these repeated portions that can be stored for a given fixed portion of a join index; however, there is a *physical* limit in that the total size of any one fixed portion/repeated portion pair cannot exceed Teradata row size limit.

Syntax Element …	Specifies …
table_name	the name of a table from which a column or columns to be included in the join index are taken as well as the name of a table to be joined with at least one other table to form a join index. You cannot specify the name of a derived table, temporary or volatile table, view, macro, trigger, stored procedure, journal table, or join or hash index. Each table in the FROM clause should have at least one of its fields contained in either *column_1_name* or *column_2_name*. In the case of a single-table join index, all columns in a FROM clause table must be contained in the *column_1_name* list. Each table name can be qualified by a database name if necessary and can have a maximum of 64 different columns referenced in the entire join index definition. Every table must be connected to at least one other table by means of a join condition. The cross join and full outer join conditions are not supported. The maximum number of *table_name* references allowed is the same as the system limit for SELECT statements. A FROM clause can have any number of simple tables, but is limited to a single joined *non*-simple table expression.

table_name (continued)	For example, the two following examples are legal. The first is a simple table list and the second is a single non-simple table expression.
	`FROM table_1, table_2, table_3,  table_4`
	`FROM (table_1 INNER JOIN table_2 ON x1 = x2)`
	`INNER JOIN (table_3 ON x1 = x3)`
	The following example is *not* legal because it has more than one non-simple table expression.
	`FROM (table_1 INNER JOIN table_2`
	`ON table_1 x1 = x2),(table_3`
	`INNER JOIN table_4 ON x3 = x4)`

correlation_name	an aliased name for *table_name*. This name must have been defined previously. Correlation names are also referred to as range variables.

Joined Table

The *joined_table* clause is not used for the single-table syntax.

joined_table	the name of a table (or already joined tables) from which the columns of this join index are to be derived.
INNER JOIN	keywords defining the join as an ordinary join.

Syntax Element …	Specifies …
LEFT OUTER JOIN RIGHT OUTER JOIN	keywords defining the join as either a left or a right outer join. For most non-aggregate applications, this is the better choice for defining a join index. You cannot specify outer joins for a join index that also contains aggregate operators in its definition. When you specify a LEFT or RIGHT outer join, the following rules apply: • No subsequent WHERE clause is permitted. • The outer table joining column for each condition must be contained in either *column_1_name* or *column_2_name*. • The inner table of each join condition must have at least one non-nullable column in either *column_1_name* or *column_2_name*.
ON	that a search condition for a join follows.
search_condition	a conditional expression used to eliminate all rows from a query that do not evaluate to TRUE. Multiple join conditions must be connected using the AND logical operator. Data types for any columns used in a join condition must be drawn from the same domain because neither explicit nor implicit data type conversions are permitted.
table_name AS *correlation_name*	the same definitions as for "table_name" on page 1-113 and "correlation_name" on page 1-113.
End Joined_Table	
WHERE *search_condition*	a conditional expression used to eliminate all rows from a query that do not evaluate to TRUE. Multiple join conditions must be connected using the AND logical operator. Data types for any columns used in a join condition must be drawn from the same domain because neither explicit nor implicit data type conversions are permitted. You can only specify a join condition using an inequality operator if you specify multiple join conditions and use the AND logical operator to connect the inequality join condition or conditions with an equality join condition Do not specify a WHERE clause for a single-table join index.
GROUP BY *column_name* *column_position*	the name of the column or columns or their position within the join index definition used to group aggregate index rows on the AMPs. If you specify a GROUP BY clause but do not specify either a COUNT or a SUM operator, then Teradata adds a COUNT(*) expression to the column list and names it CountStar.

Syntax Element …	Specifies …
ORDER BY *column_name* *column_position*	the name or position of the column within the join index definition used to order index rows on the AMPs. Sort order is restricted to ASC. Aggregate columns and expressions are *not* permitted as a *column_name* or *column_position*. *column_name* must be in the *column_name_1* list. Data in *column_name* is limited to up to four bytes of the following data types: • DATE • BYTEINT • INTEGER • DECIMAL • SMALLINT If not specified in the ORDER BY clause, the join index rows on an AMP are sorted by the row hash of their primary index.

Indexes

The INDEXES clause is used with both the multitable and single-table syntaxes.

PRIMARY	that the defined index is a nonunique primary index. The primary index for an aggregate join index must be drawn from the columns specified in the GROUP BY clause of the join index definition. You cannot specify any aggregated columns as the primary index. The primary index must be defined as a NUPI even when it is defined on a unique column. You cannot define a UPI for a join index. If you do not define an explicit NUPI, then the first column defined for the join index is assigned to be the NUPI by default. The primary index cannot be partitioned.
index_name	the name of the index being defined.
ALL	that the defined NUSI is to maintain Row ID pointers for each logical row of the join index (as opposed to only the compressed physical rows). ALL also ignores the NOT CASESPECIFIC attribute of data types so a NUSI can include case-specific values. ALL enables a NUSI to cover a join index, which enhances performance by eliminating the need to access the join index when all values needed by a query are in the secondary index. However, ALL might also require the use of additional index storage space. Use this keyword only when a secondary index is being defined on top of a join index. You cannot specify ALL with a PRIMARY index. You cannot specify multiple indexes that differ only by the presence or absence of the ALL option.

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>column_name_1</td><td>one or more columns whose field values are to be indexed. If more than one column is named, the new index is based on the combined values of each column.

If two specified columns have the same name, both names must be aliased. See “Standard Form for Data in the Teradata RDBMS” in the chapter “SQL Data Handling Fundamentals” in Teradata RDBMS SQL Reference, Volume 1.

A maximum of 64 columns can be defined for one index.

If you specify ORDER BY, then lists of both the column_name_1 and column_name_2 are limited to a single column, and column_name_2, if specified, must be identical to column_name_1.

Multiple indexes can be defined on the same columns as long as each index differs in its ordering option (VALUES versus HASH).</td></tr>
<tr><td>ORDER BY</td><td>row ordering on each AMP by a single NUSI column: either value-ordered or hash-ordered.

Rules for using an ORDER BY clause are shown in the following table.
<table>
<tr><th>IF you specify ORDER BY …</th><th>THEN …</th></tr>
<tr><td>VALUES</td><td>column_name_2 must be numeric with four bytes or less</td></tr>
<tr><td>without specifying the HASH or VALUES keywords</td><td>VALUES is assumed by default.</td></tr>
<tr><td>with a column_name_2</td><td>column_name_2 must be one of the columns specified in the column_name_1 list.</td></tr>
<tr><td>without a column_name_2</td><td>column_name_2 is assumed to be the first column specified in the column_name_1 list.</td></tr>
</table></td></tr>
<tr><td>VALUES</td><td>value-ordering for the ORDER BY column.

Select VALUES to optimize queries that return a contiguous range of values, especially for a covered index or a nested join.</td></tr>
<tr><td>HASH</td><td>hash-ordering for the ORDER BY column.

Select HASH to limit hash-ordering to one column, rather than all columns of the primary index, which is the default.

Hash-ordering a multicolumn NUSI on one of its columns allows the NUSI to participate in a nested join where join conditions involve only that ordering column.</td></tr>
</table>

Syntax Element …	Specifies …
column_name_2	a column in the *column_name_1* list that specifies the sort order to be used. Supported data types for a value-ordered, four-bytes-or-less *column_name_2* are: • DATE • BYTEINT • INTEGER • DECIMAL • SMALLINT
End Indexes	

ANSI Compliance

CREATE JOIN INDEX is a Teradata extension to the ANSI SQL-99 standard. ANSI does not define DDL for creating indexes.

Authorization

You must have the following sets of privileges to create a join index:

- CREATE TABLE privileges on the database in which the join index is to be created
- DROP TABLE or INDEX privileges on each of the covered tables or their containing databases

Owners have implicit privileges on the join index. The following default privileges are granted to the creator of a join index:

- SELECT
- DROP TABLE
- DUMP
- RESTORE
- INDEX

Maximum Number of Indexes Definable Per Data, Hash, or Join Index Table

Up to 32 secondary[5] , hash, and join indexes, in any combination, can be defined for one user data base table.

5 A NUSI defined with an ORDER BY clause counts as 2 indexes in this calculation.

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: in other words, that the time spent generating the query plan does not exceed the accrued performance enhancement.

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 often best collected directly on the base table columns rather than separately as is always required for multitable join indexes.

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	• "Collecting Statistics on a Single-Table Join Index" on page 1-119. • "Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns" on page 1-120. • "Guidelines for Collecting Statistics On Single-Table Join Index Columns" on page 1-120.
multitable	"Collecting Statistics on Multitable Join Indexes" on page 1-121.

You can use the Teradata Statistics Wizard client utility to determine which indexes best support a join index for a specified SQL query workload. See *Teradata Statistics Wizard User Guide* for further information about the Teradata Statistics Wizard.

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. See "Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns" on page 1-120 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 PRIMARY INDEX 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 statements.

```
COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_custkey);
COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_orderdate);
```

It is always better to collect the statistics for a multicolumn index[6] 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.

For example, a query with a predicate 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.

6 Note that you can collect statistics on multiple non-indexed columns as well. See "Collecting Statistics on Multiple Non-Indexed Columns" on page 3-22.

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.

There are two reasons why you might need to collect statistics on single-table join index columns instead of their underlying base table columns.

- 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 Non-Indexed Columns" on page 3-22).
  - 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 itself.

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 NCR support 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. This is particularly critical for accurate cardinality estimates.
-

IF an execution plan might involve ...	THEN collect statistics on the ...
search condition keys	column set that constitutes the search condition predicate.
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.

-

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 *column_name_1* 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 clause predicates, 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.

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 in order to improve the performance of its creation as well as its update maintenance. Note that the same procedure should be followed if you decide to collect statistics on the columns of a single-table join index rather than the corresponding columns of the base table on which it is defined.

Step	Action
1	Collect statistics on the base tables underlying the planned multitable join index.
2	Immediately create the new multitable join index.

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 performant, 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 third normal form.

You can also define a simple join index on a single table. This permits you to hash some or all of the columns of a large replicated base table on a foreign key that hashes rows to the same AMP as another large table. In some situations, this is more performant than building a multitable join index on the same columns. The advantage comes from 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 table describes a general procedure for defining a single-table join index:

Step	Action
1	Define a join index on the frequently joined columns of the table to be partitioned on a new primary index.
2	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[a] to enable the Optimizer to join a partial covering index to its base table to access any noncovered columns. Even though you do not explicitly specify this join when you write your query, it counts against the 64 table restriction on joins. Do *not* specify WHERE clauses. They are not allowed in single-table join indexes.
3	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.
4	If performance suggests it, use CREATE INDEX to create one or more NUSIs on the join index.
5	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" on page 1-119 and "Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns" on page 1-120 for clarification.

a. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement.

Support for single-table join indexes as partial covering indexes (an index having the capability to access base table columns via a prime access key stored in the join index) is provided by the following join index substitution logic.

IF the definition for a single-table join index ...	THEN that join index ...
specifies any one or more of the following items: • ROWID keyword as a column name in the *column_name* list[a] • 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	qualifies as a partial covering index if it also contains a subset of the columns specified by 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 single-table join index definition.
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.

a. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement.

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.

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...
the base table ROWID only[a]	Row ID
the base table primary index	Merge In this case, both the primary index column set and the ROWID, if present, are used to make the join.

a. 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 tables per query block restriction on joins.

IF this item is specified in the join index definition …	THEN it is …
UPI	not necessary to include ROWID in the index definition[a] to enable it as a partial cover because a UPI is sufficient to identify any base table row.
NUPI	necessary to include either of the following specifications to make it eligible for partial coverage of a query. • The base table NUPI column set *and* ROWID in the index definition. • The base table NUPI column set *and* a base table USI column set in the index definition. The NUPI plus ROWID option is the preferable choice.

a. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement.

The Optimizer specifies a partially covering single-table 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 a fully covering index.

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, ROWID
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;

 *** Help information returned. 29 rows.
 *** Total elapsed time was 1 second.
```

```
Explanation
------------------------------------------------------------------------
  1) First, we lock a distinct STJI."pseudo table" for read on a
     RowHash to prevent global deadlock for STJI.j1.
  2) Next, we lock a distinct STJI."pseudo table" for read on a RowHash
     to prevent global deadlock for STJI.t2.
  3) We lock a distinct STJI."pseudo table" for read on a RowHash to
     prevent global deadlock for STJI.t1.
  4) We lock STJI.j1 for read, we lock STJI.t2 for read, and we lock
     STJI.t1 for read.
  5) We do an all-AMPs JOIN step from STJI.t2 by way of a RowHash match
     scan. with a condition of ("STJI.t2.y2 = 1"), which is joined to
     STJI.j1.  STJI.t2 and STJI.j1 are joined using a merge join, with
     a join condition of ("STJI.j1.y1 = STJI.t2.x2").  The input table
     STJI.j1 will not be cached in memory.  The result goes into Spool
     2, which is redistributed by hash code to all AMPs.  Then we do a
     SORT to order Spool 2 by field Id 1.  The size of Spool 2 is
     estimated with no confidence to be 1 row.  The estimated time for
     this step is 0.06 seconds.
  6) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
     all-rows scan, which is joined to STJI.t1.  Spool 2 and STJI.t1
     are joined using a row id join, with a join condition of (
     "Field_1 = STJI.t1.RowID").  The input table STJI.t1 will not be
     cached in memory.  The result goes into Spool 1, which is built
     locally on the AMPs.  The size of Spool 1 is estimated with no
     confidence to be 1 row.  The estimated time for this step is 0.20
     seconds.
  7) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 0.26 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.

```
INSERT 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 a distinct STJI."pseudo table" for read on a
     RowHash to prevent global deadlock for STJI.t2.
  2) Next, we lock a distinct STJI."pseudo table" for read on a RowHash
     to prevent global deadlock for STJI.t1.
  3) We lock STJI.t2 for read, and we lock STJI.t1 for read.
  4) We do an all-AMPs RETRIEVE step from STJI.t1 by way of an all-rows
     scan with a condition of ("NOT (STJI.t1.y1 IS NULL)") into Spool 2,
     which is redistributed by hash code to all AMPs.  Then we do a
     SORT to order Spool 2 by row hash.  The size of Spool 2 is
     estimated with low confidence to be 101 rows.  The estimated time
     for this step is 3 minutes and 9 seconds.
  5) We do an all-AMPs JOIN step from STJI.t2 by way of a RowHash match
     scan., which is joined to Spool 2 (Last Use).  STJI.t2 and Spool 2
     are joined using a merge join, with a join condition of ("y1 =
     STJI.t2.x2").  The result goes into Spool 1, which is built
     locally on the AMPs.  The size of Spool 1 is estimated with no
     confidence to be 3 rows.  The estimated time for this step is 0.18
     seconds.
  6) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 3 minutes and 9 seconds.
```

Aggregate Join Indexes

Define an aggregate join index as a join index that specifies SUM or COUNT aggregate operations. No other aggregate functions are permitted in the definition of a join index.

To avoid numeric overflow, the COUNT and SUM fields in a join index definition must be typed as FLOAT. If you do not assign a data type to COUNT and SUM, Teradata types them as FLOAT automatically. If you assign a type other than FLOAT, an error message occurs.

The primary function of an aggregate join index is to provide the Optimizer with a performant, 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 third normal form.

Restrictions on Inequality Conditions in WHERE Clauses

The following rules apply to the use of inequality conditions in a WHERE clause.

- Inequality conditions between columns from the same table are always valid.
- Inequality conditions between columns from different tables are supported only if they are ANDed to at least one equality condition.

- Inequality conditions can be specified only for columns having the same data type in order to enforce domain integrity. The only valid comparison operators for an inequality condition are the following:
  - <
  - <=
  - >
  - >=

The following join index definition is valid because the WHERE clause inequality is on columns o_totalprice and o_unitprice, which are from the same table:

```
CREATE JOIN INDEX ord_price_idx AS
SELECT c_name, o_orderkey, o_orderdate
FROM orders, customer
WHERE o_totalprice <> o_unitprice;
```

The following join index definition is valid because the WHERE clause inequality on o_totalprice and c_acctbal, columns from different tables, is ANDed with the previous equality condition on o_custkey and c_custkey:

```
CREATE JOIN INDEX ord_cust_idx AS
SELECT c_name, o_orderkey, o_orderdate
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_totalprice > c_acctbal;
```

The following join index definition is *not* valid because the WHERE clause inequality condition is on columns from different tables and has no logical AND relationship with an equality condition:

```
CREATE JOIN INDEX ord_cust_idx AS
SELECT c_name, o_orderkey, o_orderdate
FROM orders, customer
WHERE o_totalprice > c_acctbal;
```

Restrictions on Inequality Conditions in ON Clauses

Inequality conditions are not supported under any circumstances for ON clauses in join index definitions.

Outer Joins and Aggregate Join Indexes

Outer joins are *not* permitted in aggregate join index definitions.

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.

Distribution of Join Index Rows

The hash distribution of join index rows across AMPs is controlled by the specified PRIMARY INDEX clause. Note that the primary index for a join index must be nonunique and not partitioned: UPIs and PPIs are not permitted. If you do not define a primary index explicitly, then the first column defined for the join index is assigned to be the primary index.

By default, rows are sorted locally on each AMP by the hash code of the primary index column or columns. 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 value or by hash.

Sorting a join index by data values, as opposed to hash codes, is especially useful for range queries that involve the sort key. Value ordering is limited to a single four-byte numeric column.

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);
```

Join Indexes Are Maintained by Teradata

Join indexes are automatically maintained by Teradata when updates (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.

Creating Join Indexes for Tables With Partitioned Primary Indexes

If you experience 3710 errors when running queries against a PPI table with a join index, you can increase the amount of memory that is available to the Optimizer by updating several DBSControl flags to values that ensure that it can process any request on the PPI table.

The recommended initial settings are as follows:

MaxParseTreeSegs = 2000

OptMaxGlobalJI = 8

You might find that your query workloads are such that these values need to be increased still further.

OptMaxGlobalJI is an internal DBSControl field. You must contact a Teradata support representative to change its value.

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 perform COLLECT STATISTICS on appropriate columns and indexes to ensure that the Optimizer has accurate statistical summaries of their demographics.

You can also perform DROP, HELP and SHOW statements on join indexes; however, it is *not* legal to directly query or update a join index.

For example, the following query is not legal 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 Teradata implicitly maintains all updates to join indexes to ensure that no information is ever lost and that semantic integrity is always enforced.

Restrictions and Limitations for Load Utilities

You cannot use FastLoad, MultiLoad, or Teradata Warehouse Builder to load data into base tables that have join indexes because those indexes are not maintained during the execution of these utilities. If you attempt to load data into base tables with join indexes using either of 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 Teradata Warehouse Builder.

Load utilities like TPump, which performs standard SQL row inserts and updates, *are* supported for join-indexed tables.

Restrictions and Limitations for Archive and Restore

Though you can archive a database that contains join indexes, the Archive utility does not save the join indexes themselves. Because no record of any join index that was defined in the database remains after a Restore operation is performed after a Restore operation is performed, the SHOW JOIN INDEX and HELP DATABASE statements do not indicate that the index ever existed.

If you perform a partial database restoration, then any join indexes that reference tables restored from that database are marked non-valid.

The following table describes the behavior of the SHOW JOIN INDEX and HELP DATABASE statements in this situation.

This SQL statement ...	Reports the following information about join indexes after a partial database restoration ...
SHOW JOIN INDEX	a warning that the join index is not valid.
HELP DATABASE	the names of the pre-existing join indexes and *no* indication that they are no longer valid.

After either a partial of full database restoration, you must drop any affected join index definitions, recreate them, and collect new statistics for them.

Restrictions and Limitations for Ordered Analytical Functions

When an ordered analytical function is specified on columns that are also defined for a compressed join index, the Optimizer does not select the join index to process the query.

Triggers Are Not Supported With Join Indexes

You cannot define triggers and join indexes on the same table.

Partitioned Primary Index Tables Are Supported With Join Indexes

You can create join indexes on a table that has a partitioned primary index; however you *cannot* specify a PPI for a join index.

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.

Example: Creating and Using a Simple Join Index

This example set uses the following table definitions.

```
CREATE TABLE customer
     (c_custkey INTEGER,
      c_name CHAR(26),
      c_address VARCHAR(41),
      c_nationkey INTEGER,
      c_phone CHAR(16),
      c_acctbal DECIMAL(13,2),
      c_mktsegment CHAR(21),
      c_comment VARCHAR(127))
UNIQUE PRIMARY INDEX (c_custkey);

CREATE TABLE orders
     (o_orderkey INTEGER NOT NULL,
      o_custkey INTEGER,
      o_orderstatus CHAR(1),
      o_totalprice DECIMAL(13,2) NOT NULL,
      o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      o_orderpriority CHAR(21),
      o_clerk CHAR(16),
      o_shippriority INTEGER,
      o_commment VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey);

CREATE TABLE lineitem
     (l_orderkey INTEGER NOT NULL,
      l_partkey INTEGER NOT NULL,
      l_suppkey INTEGER,
      l_linenumber INTEGER,
      l_quantity INTEGER NOT NULL,
      l_extendedprice DECIMAL(13,2) NOT NULL,
      l_discount DECIMAL(13,2),
      l_tax DECIMAL(13,2),
      l_returnflag CHAR(1),
      l_linestatus CHAR(1),
      l_shipdate DATE FORMAT 'yyyy-mm-dd',
      l_commitdate DATE FORMAT 'yyyy-mm-dd',
      l_receiptdate DATE FORMAT 'yyyy-mm-dd',
      l_shipinstruct VARCHAR(25),
      l_shipmode VARCHAR(10),
      l_comment VARCHAR(44))
PRIMARY INDEX (l_orderkey);
```

The following statement defines a join index on these tables. Subsequent examples demonstrate the effect of this join index on how the Optimizer processes various queries.

```
CREATE JOIN INDEX order_join_line AS
SELECT (l_orderkey, o_orderdate, o_custkey, o_totalprice),
       (l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM lineitem
LEFT JOIN orders ON l_orderkey = o_orderkey
ORDER BY o_orderdate
PRIMARY INDEX (l_orderkey);

 *** Index has been created.
 *** Total elapsed time was 15 seconds.
```

The following EXPLAIN shows how the newly created join index, order_join_line, might be used by the Optimizer.

```
EXPLAIN
SELECT o_orderdate, o_custkey, l_partkey, l_quantity,
       l_extendedprice
FROM lineitem, orders
WHERE l_orderkey = o_orderkey;

Explanation
------------------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock LOUISB.order_join_line for read.
  3) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.order_join_line by way of an all-rows scan with a condition
     of ("NOT (LOUISB.order_join_line.o_orderdate IS NULL)") into Spool 1,
     which is built locally on the AMPs.  The input table will not be
     cached in memory, but it is eligible for synchronized scanning.
     The result spool file will not be cached in memory.  The size of
     Spool 1 is estimated to be 1,000,000 rows.  The estimated time for
     this step is 4 minutes and 27 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

The following EXPLAIN report shows how the join index might be used in a query when an additional search condition is added on the join indexed rows.

```
EXPLAIN
SELECT o_orderdate, o_custkey, l_partkey, l_quantity,
       l_extendedprice
FROM lineitem, orders
WHERE l_orderkey = o_orderkey
AND o_orderdate > '1997-11-01';
```

```
Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock LOUISB.order_join_line for read.
  3) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.order_join_line with a range constraint of (
     "LOUISB.order_join_line.Field_1026 > 971101") with a residual
     condition of ("(NOT (LOUISB.order_join_line.o_orderdate IS NULL ))
     AND (LOUISB.order_join_line.Field_1026 > 971101)") into Spool 1,
     which is built locally on the AMPs.  The input table will not be
     cached in memory, but it is eligible for synchronized scanning.
     The size of Spool 1 is estimated to be 1000 rows.  The estimated time
     for this step is 0.32 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

The following EXPLAIN report shows how the join index might be used in a query when aggregation is performed on the join indexed rows.

```
EXPLAIN
SELECT l_partkey, avg(l_quantity), AVG(l_extendedprice)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey
AND o_orderdate > '1997-11-01'
GROUP BY l_partkey;
```

```
Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     RowHash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock LOUISB.order_join_line for read.
  3) We do a SUM step to aggregate from join index table
     LOUISB.order_join_line with a range constraint of (
     "LOUISB.order_join_line.Field_1026 > 971101") with a residual
     condition of ("(LOUISB.order_join_line.Field_1026 > 971101) AND (NOT
     (LOUISB.order_join_line.o_orderdate IS NULL ))"), and the grouping
     identifier in field 1.  Aggregate Intermediate Results are
     computed globally, then placed in Spool 3.  The input table will
     not be cached in memory, but it is eligible for synchronized
     scanning.
  4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of
     an all-rows scan into Spool 1, which is built locally on the AMPs.
     The size of Spool 1 is estimated to be 10 rows.  The estimated time
     for this step is 0.32 seconds.
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

The following EXPLAIN report shows how the join index might be used in a query when join indexed rows are used to join with another base table.

```
EXPLAIN
SELECT o_orderdate, c_name, c_phone, l_partkey, l_quantity,
       l_extendedprice
FROM lineitem, orders, customer
WHERE l_orderkey = o_orderkey
AND o_custkey = c_custkey;
```

```
Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.customer.
  3) We lock LOUISB.order_join_line for read, and we lock LOUISB.customer
     for read.
  4) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.order_join_line by way of an all-rows scan with a condition
     of ("NOT (LOUISB.order_join_line.o_orderdate IS NULL)") into Spool 2,
     which is redistributed by hash code to all AMPs.  Then we do a
     SORT to order Spool 2 by row hash.  The size of Spool 2 is
     estimated to be 1,000,000 rows.  The estimated time for this step
     is 1 minute and 53 seconds.
  5) We do an all-AMPs JOIN step from LOUISB.customer by way of a
     Row Hash match scan with no residual conditions, which is joined to
     Spool 2 (Last Use).  LOUISB.customer and Spool 2 are joined using
     a merge join, with a join condition of ("Spool_2.o_custkey =
     LOUISB.customer.c_custkey").  The result goes into Spool 1, which
     is built locally on the AMPs.  The size of Spool 1 is estimated to
     be 1,000,000 rows.  The estimated time for this step is 32.14
     seconds.
  6) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

The following EXPLAIN shows how the join index might be used in a query of a single table.

```
EXPLAIN
SELECT l_orderkey, l_partkey, l_quantity, l_extendedprice
FROM lineitem
WHERE l_partkey = 1001;

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock LOUISB.order_join_line for read.
  3) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.order_join_line by way of an all-rows scan with a condition
     of ("LOUISB.order_join_line.l_partkey = 1001") into Spool 1, which
     is built locally on the AMPs.  The input table will not be cached
     in memory, but it is eligible for synchronized scanning.  The
     result spool file will not be cached in memory.  The size of Spool
     1 is estimated to be 100 rows.  The estimated time for this step
     is 59.60 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

Example: Creating a Sparse Join Index

You can create join indexes that limit the number of rows in the index to only those that are accessed when, for example, a frequently run query references only a small, well known subset of the rows of a large base table. By using a constant expression to filter the rows included in the join index, you can create what is known as a sparse index.

For example, the following DDL creates an aggregate join index containing only the sales records from 2000:

```
CREATE JOIN INDEX J1 AS
SELECT storeid, deptid, SUM(sales_dollars)
FROM sales
WHERE EXTRACT(year, sales_date) = 2000
GROUP BY storeid, deptid;
```

When a user query is entered, the Optimizer determines if accessing J1 gives the correct answer and is more efficient than accessing the base tables. This sparse join index would be selected by the Optimizer only for queries that restricted themselves to data from the year 2000. For example. a query might require data from departments 42, 43, and 44, but only for the year 2000. Because the join index J1 contains all of the data for year 2000, it might be used to satisfy the following query:

```
SELECT storeid, deptid, SUM(sales_dollars)
FROM sales
WHERE EXTRACT(year FROM sales_date) = 2000
AND deptid IN(42, 43, 44)
GROUP BY storeid, deptid;
```

Example: Defining and Using a Simple Join Index With an *n*-way Join Result

The following example shows the creation of a join index defined with a multiway join result and then shows how the Optimizer uses the join index to process a query on the base tables for which it is defined.

```
CREATE JOIN INDEX cust_order_join_line AS
SELECT (l_orderkey, o_orderdate, c_nationkey, o_totalprice),
       (l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM (lineitem
LEFT JOIN orders ON l_orderkey = o_orderkey)
INNER JOIN customer ON o_custkey = c_custkey
PRIMARY INDEX (l_orderkey);

 *** Index has been created.
 *** Total elapsed time was 20 seconds.

EXPLAIN
SELECT l_orderkey, o_orderdate, o_totalprice, l_partkey,
       l_quantity, l_extendedprice, l_shipdate
FROM lineitem, orders, customer
WHERE l_orderkey = o_orderkey
AND o_custkey = c_custkey
AND c_nationkey = 10;

 *** Help information returned. 16 rows.
 *** Total elapsed time was 1 second.
```

```
Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.cust_order_join_line.
  2) Next, we lock LOUISB.cust_order_join_line for read.
  3) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.cust_order_join_line by way of an all-rows scan with a
     condition of ("LOUISB.cust_order_join_line.c_nationkey = 10") into
     Spool 1, which is built locally on the AMPs.  The input table will
     not be cached in memory, but it is eligible for synchronized
     scanning.  The result spool file will not be cached in memory.
     The size of Spool 1 is estimated to be 200 rows.  The estimated time
     for this step is 3 minutes and 57 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

Example: Creating and Using an Aggregate Join Index

This example set uses the following table definitions.

```
CREATE TABLE customer
  (c_custkey INTEGER NOT NULL,
   c_name CHAR(26) CASESPECIFIC NOT NULL,
   c_address VARCHAR(41),
   c_nationkey INTEGER,
   c_phone CHAR(16),
   c_acctbal DECIMAL(13,2),
   c_mktsegment CHAR(21),
   c_comment VARCHAR(127)
  )
UNIQUE PRIMARY INDEX (c_custkey);

CREATE TABLE orders
  (o_orderkey INTEGER NOT NULL,
   o_custkey INTEGER,
   o_orderstatus CHAR(1) CASESPECIFIC,
   o_totalprice DECIMAL(13,2) NOT NULL,
   o_orderdate DATE FORMAT 'YYYY-MM-DD' NOT NULL,
   o_orderpriority CHAR(21),
   o_clerk CHAR(16),
   o_shippriority INTEGER,
   o_comment VARCHAR(79)
  )
UNIQUE PRIMARY INDEX (o_orderkey);
```

Consider the following aggregate join query.

```
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_orderdate > DATE '1998-09-20'
AND o_orderdate < DATE '1998-10-15'
GROUP BY c_nationkey;
```

Without an aggregate join index, a typical execution plan for this query might involve the following stages.

Stage	Process
1	Redistribute orders into a spool file.
2	Sort the spool file on o_custkey.
3	Merge join the sorted spool file and the customer file.
4	Aggregate the result of the merge join.

Suppose you define the following aggregate join index, which aggregates o_totalprice over a join of orders and customer.

```
CREATE JOIN INDEX ord_cust_idx AS
SELECT c_nationkey, SUM(o_totalprice(FLOAT)) AS price,
       o_orderdate
FROM orders, customer
WHERE o_custkey = c_custkey
GROUP BY c_nationkey, o_orderdate
ORDER BY o_orderdate;
```

The execution plan produced by the Optimizer for this query includes an aggregate step on the aggregate join index, which is much smaller than either one of the join tables. You can confirm this by performing an EXPLAIN on the query.

```
EXPLAIN
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_orderdate > DATE '1998-09-20'
AND o_orderdate < DATE '1998-10-15'
GROUP BY c_nationkey;

*** Help information returned. 18 rows.
 *** Total elapsed time was 3 seconds.
Explanation
  1) First, we lock a distinct TPCD."pseudo table" for read on a
     RowHash to prevent global deadlock for TPCD.ord_cust_idx.
  2) Next, we lock TPCD.ord_cust_idx for read.
  3) We do a SUM step to aggregate from join index table
     TPCD.ordcustidx by way of an all-rows scan with a condition of (
     "(TPCD.ord_cust_idx.O_ORDERDATE > DATE '1998-09-20') AND
     (TPCD.ord_cust_idx.O_ORDERDATE < DATE '1998-10-15')"), and the
     grouping identifier in field 1.  Aggregate Intermediate Results
     are computed globally, then placed in Spool 2.  The size of Spool
     2 is estimated to be 1 row.
  4) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of
     an all-rows scan into Spool 1, which is built locally on the AMPs.
     The size of Spool 1 is estimated with no confidence to be 1 row.
     The estimated time for this step is 0.17 seconds.
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1.
```

Using the EXTRACT Function With Join Indexes

Join index definitions support the EXTRACT function.

The following example illustrates the use of the EXTRACT function in the definition of an aggregate join index.

```
CREATE JOIN INDEX ord_cust_idx_2 AS
SELECT c_nationkey, SUM(o_totalprice(FLOAT)) AS price,
       EXTRACT(YEAR from o_orderdate) AS o_year
FROM orders, customer
WHERE o_custkey = c_custkey
GROUP BY c_nationkey, o_year
ORDER BY o_year;
```

The aggregation is based only on the year of o_orderdate, which has fewer groups than the entire o_orderdate, so ord_cust_idx_2 is much smaller than ord_cust_idx.

On the other hand, the use for ord_cust_idx_2 is more limited than ord_custidx. In particular, ord_cust_idx_2 can only be used to satisfy queries that select full years of orders.

For example, ord_cust_idx_2 cannot be used for the query analyzed in "Example: Creating and Using an Aggregate Join Index" on page 1-137, but the following query *does* profit from its use.

```
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_orderdate > DATE '1998-01-01'
AND o_orderdate < DATE '1998-12-31'
GROUP BY c_nationkey;
```

Rules for Deciding Whether to Use 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 nonaggregate 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.
  - Addition and multiplication are commutative.
  - COUNT(*) and COUNT(*non-null_expression*) are used interchangeably.
  - 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 in order to find a match. For example, the following expression can be converted to the expression following it in order to find a match.

```
SUM(CASE WHEN x1=1
         THEN 1
         ELSE 0)

CASE WHEN x1=1
THEN SUM(1)
ELSE SUM(0)
```

  - Predicate conditions are converted according to the following conversion table.

THIS predicate condition ...	IS converted to this equivalent expression ...
expression >= 'YYYY0101'	'EXTRACT(YEAR FROM *expression*) >= YYYY'
expression > 'YYYY1231'	'EXTRACT(YEAR FROM *expression*) > YYYY'
expression < 'YYYY0101'	'EXTRACT(YEAR FROM *expression*) < YYYY'
expression <= 'YYYY1231'	'EXTRACT(YEAR FROM *expression*) <= YYYY'
expression >= 'YYYYMM01'	'EXTRACT(YEAR FROM *expression*) > YYYY OR (EXTRACT(YEAR FROM *expression*) = YYYY AND EXTRACT(MONTH FROM *expression*) >= MM)'

THIS predicate condition …	IS converted to this equivalent expression …
expression > 'YYYYMMLD' where LD is the last day of the month specified by MM	'EXTRACT(YEAR FROM *expression*) > YYYY OR (EXTRACT(YEAR FROM *expression*) = YYYY AND EXTRACT(MONTH FROM *expression*) > MM)'
expression <= 'YYYYMMLD' where LD is the last day of the month specified by MM	'EXTRACT(YEAR FROM *expression*) < YYYY OR (EXTRACT(YEAR FROM *expression*) =YYYY AND EXTRACT(MONTH FROM *expression*) <= MM)'
expression < 'YYYYMM01'	'EXTRACT(YEAR FROM *expression*) < YYYY OR (EXTRACT(YEAR FROM *expression*) = YYYY AND EXTRACT(MONTH FROM *expression*) < MM)'

Further Information

See *Teradata RDBMS Database Design* for more examples and an extended discussion of the functions and features of join indexes.

CREATE MACRO
REPLACE MACRO

Purpose

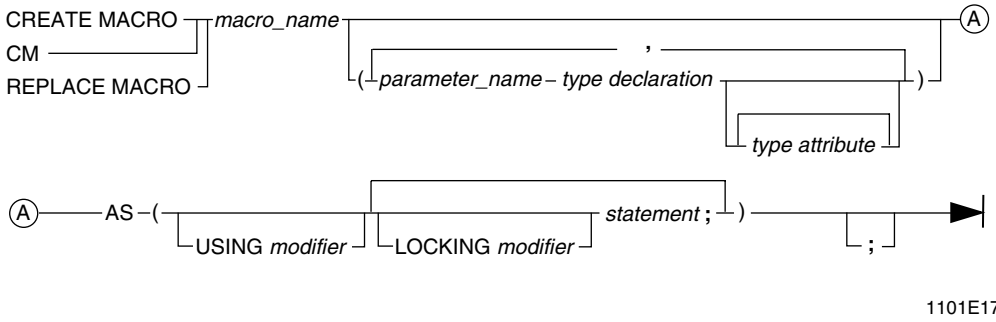
CREATE MACRO defines a set of statements that are frequently used or that perform a complex operation. The statements in the macro body are submitted when the macro is invoked by a subsequent EXECUTE statement.

REPLACE MACRO redefines an existing macro. If the specified macro does not exist, REPLACE MACRO creates a new macro with that name.

Invocation

Executable.

Syntax



1101E172

where:

Syntax Element …	Specifies …
macro_name	the name of the new macro. If a fully qualified name is not specified, the default database is used.
parameter_name	the name of a parameter that is replaced with a value during macro execution. When referenced in the macro body, a parameter name is prefixed by the COLON (:) character.
type_declaration	a data definition or default definition for a parameter. If a default value is not assigned, the user must specify a value for the parameter at EXECUTE time. For a list of data types see “Data Type Phrases” in the chapter “SQL Data Definition” in *Teradata RDBMS SQL Reference, Volume 3.*

Syntax Element …	Specifies …
type_attribute	the data type attributes, such as NOT NULL, UPPERCASE, or TITLE. For a list of data type attributes see "Core Data Type Attributes" under "Data Type Phrases" in the chapter "SQL Data Definition" in *Teradata RDBMS SQL Reference, Volume 3.*
USING *modifier*	one or more variable parameter names. A value is substituted for each parameter name when the request is processed. Also see "USING Row Descriptor" in the chapter "SQL Data Manipulation Language Statement Syntax," in *Teradata RDBMS SQL Reference, Volume 6*. This is an optional phrase. USING does not work in a macro when accessed via BTEQ. When running BTEQ, put USING in the EXECUTE statement. Including USING in a macro might require special programming. See your NCR Field support engineer or the NCR Support Center for help.
LOCKING *modifier*	a lock on a database, table, view or row. It overrides the default usage lock placed in response to a request. Also see "LOCKING Modifier" in the chapter "SQL Data Manipulation Language Statement Syntax," in *Teradata RDBMS SQL Reference, Volume 6*.
statement	Specifies an SQL statement. Every statement in the macro body must be terminated by a SEMICOLON character. Parameter names referenced in the macro body are prefaced by the COLON (:) character. The macro body can include EXECUTE statements to invoke other macros.

ANSI Compliance

CREATE MACRO and REPLACE MACRO are Teradata extensions to the ANSI SQL-99 standard.

Authorization: CREATE MACRO

You must have the CREATE MACRO privilege on the containing database or user in which the macro is to be created. The creator of a macro is automatically granted the DROP MACRO and EXECUTE privileges WITH GRANT OPTION on the macro.

Access to data via a macro is controlled by the access privileges of the macro owner, *not* by the privileges of its creator. This can be a security issue. See "GRANT (SQL Form)" on page 2-22 and to "Security Considerations With CREATE MACRO Privilege" on page 2-30 for details.

The user creating a macro must have the privileges for all statements to be performed by the macro. To allow other users to perform the macro, its creator must have the privileges necessary to perform each statement WITH GRANT OPTION.

Once a macro has been created, its owner is the database in which it exists, *not* the user who created it. The owning database must have all the appropriate privileges for executing the macro, including WITH GRANT OPTION.

The user who performs a macro need not be aware of the tables affected by its performance.

Authorization: REPLACE MACRO

You must have the DROP VIEW privilege to replace a macro.

You must have the CREATE VIEW privilege if the specified view or macro does not exist.

Why Use Macros?

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.

Limit On Request Size

The maximum size of the fully expanded text for a macro is 2 Mbyte.

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 to be unusable because of stack overflows in the Syntaxer at performance time. In this case, Teradata returns an error.

Semantic Errors Are Not Reported At the Time a Macro Is Created

Teradata checks the CREATE MACRO text for syntax errors, but not for semantic errors. 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.

Employ a USING Phrase to Retrieve Data

In designing a macro, employ the USING modifier to retrieve data from an external source such as a data file.

Use a Colon to Specify Value Insertion Variables

Use the COLON character to identify the variables whose values are to be inserted at performance time. For example, if the CREATE MACRO statement contained the following VALUES clause:

```
VALUES (:name, :city, :empno)
```

then the EXECUTE statement might contain a clause such as the following:

```
(name='Kimble G', city='Altadena', empno=8273)
```

Not All Data Attributes Are Supported in Macro Parameters

Not all of the data attributes that can be specified for table columns can be specified in macros.

For example, data type declarations and attributes have limited functionality as macro parameters.

The following data attributes are never valid as macro parameters.

- CHECK constraints
- COMPRESS phrase

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 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 tables.

When you perform a macro that contains an INSERT statement that inserts rows to a temporary table, and that table is not materialized in the current session, then an instance of the table is created when the INSERT statement performs.

When you perform a macro that contains a CREATE VOLATILE TABLE statement, 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 Performance

Performing a macro constitutes an implicit transaction. Therefore, in Teradata transaction mode, a macro need not be enclosed by a BEGIN TRANSACTION statement and an END TRANSACTION statement.

If the session performing a macro is in ANSI transaction mode, then unless the body of the macro ends with a COMMIT, the actions of the macro are uncommitted until a commit or rollback occurs in subsequent statements. If a user defines a macro with a COMMIT, it can be performed only by sessions running in ANSI mode.

Because macro execution implies a transaction, the user receives no response to individual statements contained in the macro body until all statements have been completed successfully. Any object that is referenced by a statement 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 (CREATE, DROP, MODIFY, etc.), it can contain no other statement. Note that 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 (e.g., databasename.tablename) in the macro body.

IF a macro is to be performed in this mode …	THEN …
ANSI only	add a COMMIT WORK statement to the end of its definition.
ANSI and Teradata	omit the COMMIT WORK statement from the definition and instruct users to specify it explicitly whenever they perform the macro in ANSI mode.

Resolution of Nonqualified Names

When the CREATE MACRO statement is performed, Teradata parses the text of statements in the macro and resolves names that are not fully qualified in those statements. Note that nonqualified names in DDL statements 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 is 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 DDL statement.

Because CREATE MACRO is a DDL statement, the database name is not resolved when the macro is created, but later, when the macro is performed.

The result is that if user XYZ performs Macro_1, table ABC.mambo is created, and if user LMN performs Macro_1, table DEF.mambo is created.

Now suppose that user XYZ creates macro 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 statement.

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 user ABC, LMN, or any other user.

To summarize, the following statements are true of nonqualified names when they are specified in macros:

- Nonqualified 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.
- Nonqualified names in a macro definition are *fully* resolved in DML statements when the macro is created.

Using the * Symbol in Macros

A macro defined using the * symbol is forever bound by the definition of the table as of the time the macro was created.

For example:

```
CREATE MACRO GetEmp AS
(
SELECT *
FROM Employee;
);
```

If a column is later added to or removed from the table, the following statement still returns the number of columns that existed in Employee when GetEmp was defined.

```
EXECUTE GetEmp;
```

If columns have been dropped or data types have been changed, performing the macro can result in an error message or unexpected behavior.

Using Dates in Macros

Validation of date strings in a macro takes place during its performance. 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 validation, do 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.

Naming Macros

The name of a macro must be unique within its database. A macro, table, view, trigger, hash or join index, or stored procedure can take the same name as the database in which it is created, but two identically named objects, for example, a macro named TEST and a table named TEST, cannot coexist in the same database.

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 “Macros and Tactical Queries” in the chapter “Design Issues for Tactical Queries” in *Teradata RDBMS Database Design* for further information about using macros to perform tactical queries.
- See “Stored Procedures and Tactical Queries“ in the chapter ”SQL Stored Procedures” in *Teradata RDBMS SQL Reference, Volume 6*, for a comparison of the relative efficiencies of macros and stored procedures for performing tactical queries.

Rules for Using Parameters in Macros

Parameters are values that are entered by the user into the EXECUTE statement for use by the macro during execution. Use of parameters is optional in a macro and, when required, are specified as part of the CREATE MACRO statement.

In defining parameters for a macro, follow the macro name in your CREATE MACRO statement 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 supply a parameter value in the EXECUTE statement that does not conform to the specified format, data type, or default value, performance halts with an error message.

The maximum number of parameters you can specify per macro is 2 048.

Macro Definitions and Character Data Types

A macro can contain explicit [VAR]CHARACTER(n) [CHARACTER SET …] clauses. If the character data type (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 character type of the creator/modifier of the macro (creator user default character type).

  For example, if the creator user default character type 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 character data type of the expression if the expression is typed as CHARACTER. Otherwise, the character data type is set to the default character type for the creator user.

  For example, if the creator user default character type 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 statement 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 character type of the user who performs it.

  For example, suppose the creator user default character type is Unicode. When the macro definition below is performed by a different user having a user default character type 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.

What REPLACE MACRO Does

In effect, the REPLACE MACRO statement operates as a DROP followed by a CREATE, except for the handling of access privileges. All access privileges that were granted directly on the original macro are retained.

If the specified macro does not exist, the REPLACE MACRO statement creates the view or macro. In this case, the REPLACE statement has the same effect as a CREATE MACRO statement.

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 statement (it is not dropped). This is analogous to a ROLLBACK on the operation.

Example 1

The following statement creates a macro that first inserts a row for a new employee in the Employee table, then performs a SELECT statement to verify that the information was entered correctly:

```
CREATE MACRO NewEmp1
 (number   INTEGER,
  name     VARCHAR(12),
  dept     INTEGER    DEFAULT 900,
  position VARCHAR(12)
  sex      CHARACTER,
  dob      DATE       FORMAT 'MMMbDDbYYYY',
  edlev    BYTEINT )  AS
(
```

```
INSERT INTO Employee (Empno, Name, DeptNo, JobTitle, Sex, DOB,
 EdLev)
VALUES (:number, :name, :dept, :position, :sex, :dob, :edlev);
--   The following select verifies the insert
SELECT *
FROM Employee
WHERE EmpNo = :number;
);
```

Use the -- construct to include comments in the macro. Text appearing after -- and up to the end of the line is not performed.

If this macro is performed in ANSI mode, the INSERT has not been committed. This is also true in Teradata mode when the macro is performed as part of an explicit transaction.

Example 2

This example creates a macro that also inserts a row for a new employee in the Employee table, but then performs an UPDATE statement rather than a SELECT statement. The UPDATE statement changes the Department table by incrementing the employee count in the row containing a department number that matches the value of the :dept parameter.

```
CREATE MACRO NewEmp2
 (number   INTEGER,
  name     VARCHAR(12),
  dept     INTEGER     DEFAULT 900,
  position VARCHAR(12),
  sex      CHARACTER,
  dob      DATE        FORMAT 'MMMbDDbYYYY',
  edlev    BYTEINT)   AS
(
INSERT INTO Employee (Empno, Name, DeptNo, JobTitle, Sex, DOB,
 EdLev)
VALUES (:number, :name, :dept, :position, :sex, :dob, :edlev) ;
UPDATE Department
SET EmpCount=EmpCount+1
WHERE DeptNo = :dept;
);
```

If this macro is performed in ANSI mode, the INSERT and UPDATE have not been committed. This is also true in Teradata mode when the macro is performed as part of an explicit transaction.

Example 3

This example replaces the INSERT statement with a DELETE statement, and the departmental count of employees in the Department table is decremented. The ABORT statement terminates macro execution if the row for the employee being deleted is not present in the Employee table.

The example shows a macro designed for use in ANSI mode, and for which the user wants to commit if the delete and update are successful. Note that the statements in the body of the macro are entered as one multistatement request. Therefore, if the WHERE condition of the ROLLBACK statement is met, the entire request is aborted and the value in EmpCount is protected.

```
CREATE MACRO DelEmp
 (num   SMALLINT    FORMAT '9(5)',
  dname VARCHAR(12),
  dept  SMALLINT    FORMAT '999') AS
(
ABORT 'Name does not exist'
WHERE :num NOT IN
 (SELECT EmpNo
  FROM Employee
  WHERE Name = :dname)
      ;
DELETE FROM Employee
WHERE Name = :dname
      ;
UPDATE Department
SET EmpCount = EmpCount - 1
WHERE DeptNo = :dept;
COMMIT WORK;
);
```

Example 4

You can include a condition for halting execution of a macro by incorporating an ABORT (ROLLBACK) statement into the macro. If the specified condition is encountered during execution, the macro is aborted (the transaction in process is concluded, locks on the tables are released, changes made to data are backed out, and any spooled output is deleted).

As an example, to restrict the NewEmp macro from being used to add employees to the Executive Office, department 300, incorporate an ABORT statement into the macro specification, as follows:

```
CREATE MACRO Personnel.NewEmp
 (Number    (SMALLINT     FORMAT '9(5)'),
  Name      (VARCHAR(12)),
  Dept      (SMALLINT     FORMAT '999'),
  Position  (VARCHAR(12)),
  Birthdate (DATE         FORMAT 'MMMbDDbYYYY'),
  Sex       (CHARACTER(1))
  Education (BYTEINT))   AS
(
ABORT 'Department 300 not valid'
WHERE :dept = 300;
```

```
INSERT INTO employee (empno, name, deptno, jobtitle, dob, sex,
 edlev)
VALUES (:number, :name, :dept, :position, :birthdate, :sex,
 :education);
);
```

Specify the text of an optional error message following the ABORT keyword and enclose it in APOSTROPHE characters. This message displays on the terminal screen if the macro is aborted for the specified condition. In this example, the abort condition, :dept = 300, is specified in the WHERE clause of the ABORT statement.

Example 5

The following example illustrates the use of REPLACE MACRO.

The following statement can be used to replace macro NewEmp (see "CREATE MACRO REPLACE MACRO" on page 1-142 and "SHOW HASH INDEX SHOW JOIN INDEX SHOW MACRO SHOW PROCEDURE SHOW TABLE SHOW TRIGGER SHOW VIEW" on page 4-79).

```
REPLACE MACRO NewEmp(name VARCHAR(12) NOT NULL,
 street CHARACTER(30),
 city   CHARACTER(20),
 number INTEGER NOT NULL,
 dept   SMALLINT DEFAULT 999) AS
(INSERT INTO Employee (Name, Street, City, EmpNo, DeptNo)
  VALUES (:name, :street, :city, :number, :dept);
UPDATE Department
SET EmpCount = EmpCount + 1
WHERE DeptNo = :dept ;);
```

CREATE PROCEDURE REPLACE PROCEDURE

Purpose

CREATE PROCEDURE directs the stored procedure compiler to create a procedure from the SQL statements in the remainder of the source text.

REPLACE PROCEDURE directs the stored procedure compiler to replace the definition of an existing stored procedure.

If the specified stored procedure does not exist, REPLACE PROCEDURE creates a new procedure by that name from the SQL statements in the remainder of the source text.

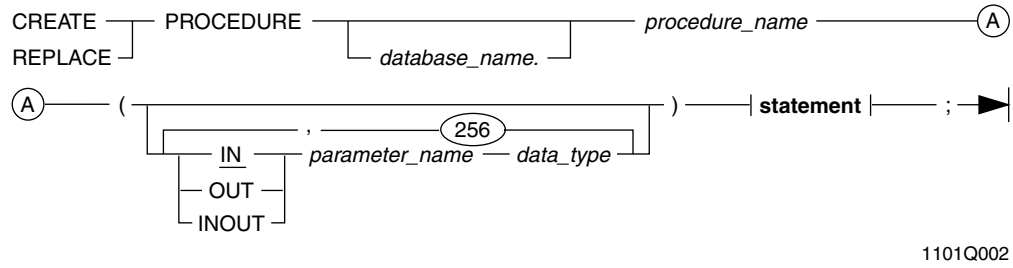
Invocation

Executable.

Stored procedures only.

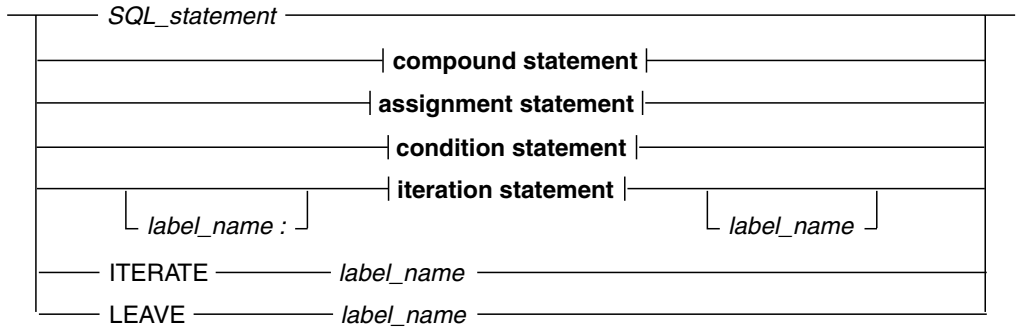
Not valid inside a stored procedure.

Syntax



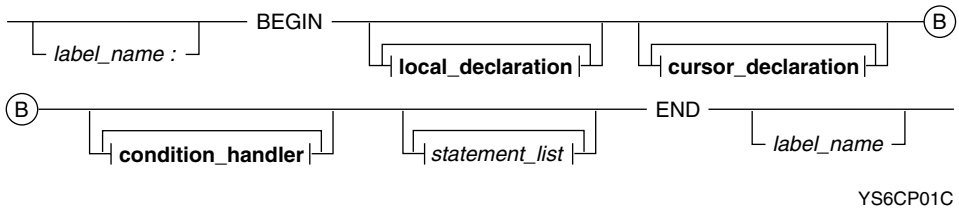
1101Q002

statement

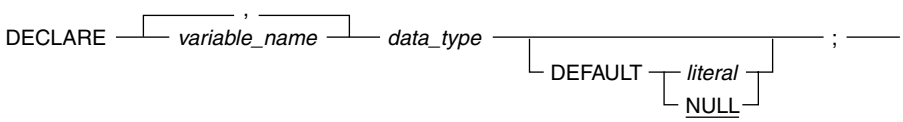


YS6CP01B

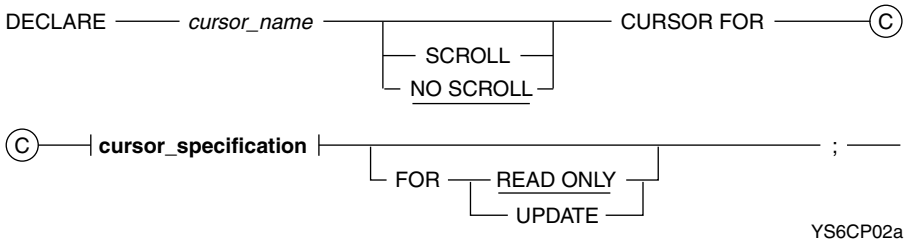
compound statement



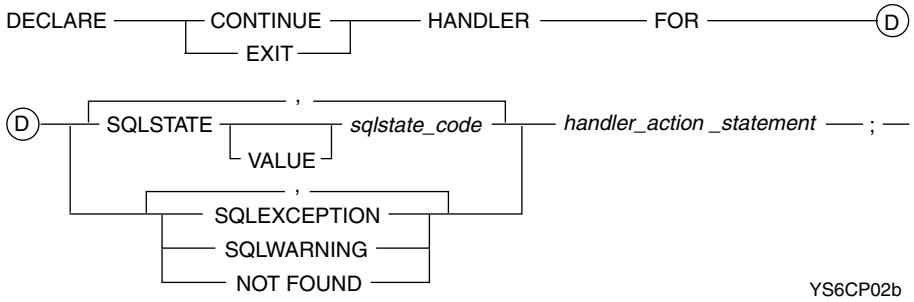
local_declaration



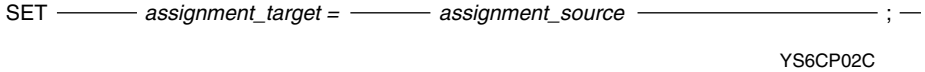
cursor_declaration



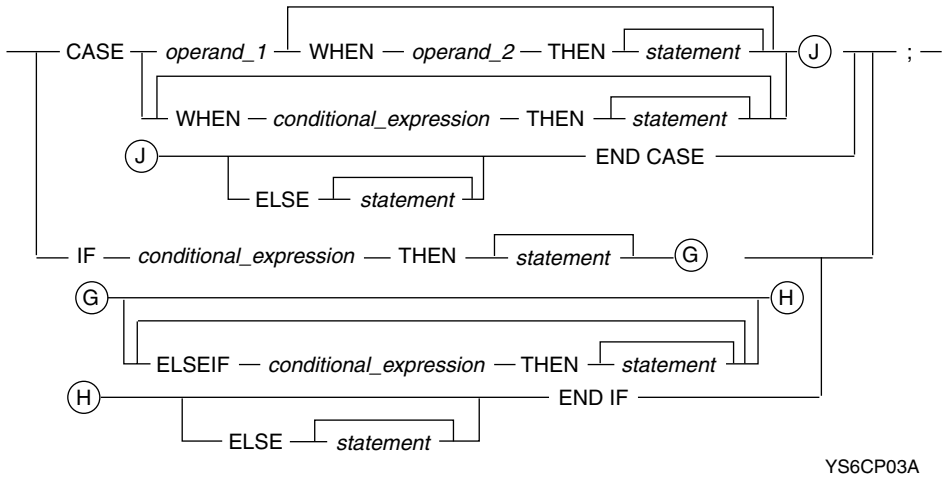
condition-handler



assignment statement

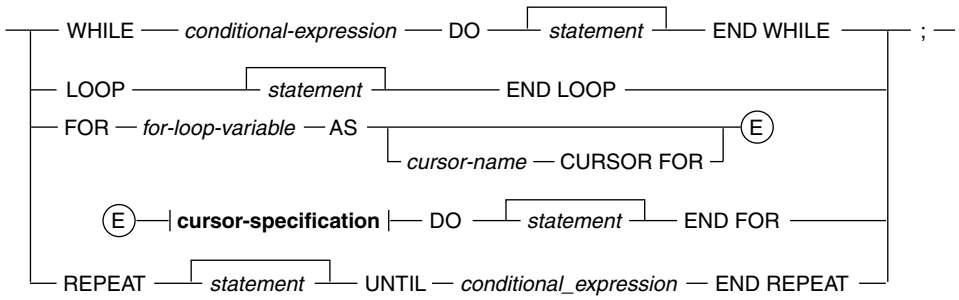


condition statement



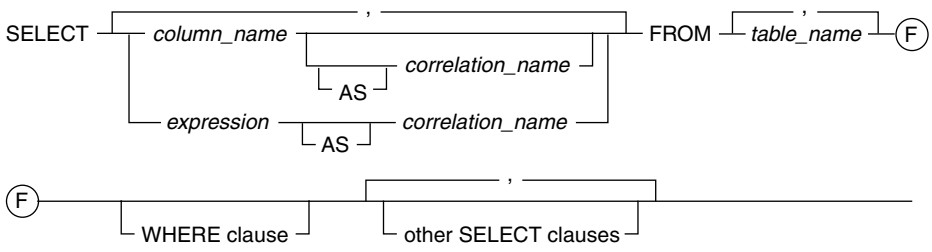
YS6CP03A

iteration statement



YS6CP03B

cursor-specification



YS6CP03C

where:

Syntax Element …	Specifies …
database_name	an optional qualifier for *procedure_name*. If the *database_name* is not specified, the current default database is used.
procedure_name	the name of the stored procedure to be created. *procedure_name* must be unique within the database.
IN OUT INOUT	type of the parameter being specified. • IN parameter is input only. • OUT parameter is output only. • INOUT can be both input and output. IN is the default value. If the parameter type is not specified, the parameter is assumed to be of the IN type.
parameter_name	the name of a parameter that is replaced with an argument during stored procedure execution. A maximum of 256 parameters can be specified with each stored procedure in a comma-separated list. The data type of the parameter must be specified with each *parameter_name.* When used in an SQL statement other than a control statement, a parameter must be prefixed with a COLON (:) character.
data_type	data definition for local variables or parameters. All Teradata-supported data types can be used. For parameters, you can specify CHARACTER SET and/or CASESPECIFIC or NOT CASESPECIFIC as an attribute of the character data type. >?parameter of the character data type after *data_type* for each parameter. If you specify both, you must specify them in the following order: • CHARACTER SET • CASTSPECIFIC or NOT CASESPECIFIC No other data type attribute, such as NOT NULL, UPPERCASE, or FORMAT, is legal for stored procedures. The keyword DEFAULT is not legal before the character set. For local variables, you can specify CHARACTER SET and NOT CASESPECIFIC in any order after data_type.

Syntax Element …	Specifies …
data_type (continued)	If CHARACTER SET is not specified, the character set will default to the character set of the user creating/compiling the stored procedure. See *Teradata RDBMS SQL Reference, Volume 3* for information about data types.
statement	the stored procedure body. You must specify either a single statement or a BEGIN-END compound statement. **Note:** You cannot specify the following as a single statement for stored procedure body: • local variable, cursor or handler declarations • cursor statements (OPEN, FETCH, and CLOSE)

Statement

The following statements can be specified *as* the stored procedure body.

You can also specify any of these statements in a *statement_list* within compound statements, condition statements (CASE and IF), iteration statements and in the handler action part of a condition handler declaration.

SQL_statement	one of the following: • SQL DML, DDL, or DCL statement supported by stored procedures. This includes a dynamic SQL statement. • control statement. The creator of the stored procedure must have the appropriate privileges on each of these statements. DDL and DCL statements, and dynamic SQL statements, are valid only if the creator of the stored procedure is also its immediate owner.
compound statement	the BEGIN-END statement enclosing a set of declarations and statements. You can specify local variable declarations, cursor declarations, condition handler declarations and SQL and control statements within a compound statement. Nesting of compound statements is allowed.
assignment statement	the SET statement.
condition statement	CASE statement or IF statement.
iteration statement	WHILE, LOOP, FOR or REPEAT statement.

Syntax Element …	Specifies …
ITERATE	keyword introducing the ITERATE statement to terminate the execution of the current iteration of a labeled iteration statement, and start the next iteration. The *label_name* references the iteration statement with which the label is associated, and whose current iteration is to be terminated. The next iteration is started conditionally in WHILE or FOR, and unconditionally in LOOP and REPEAT.
LEAVE	keyword introducing the LEAVE statement to continue performance outside an iterated iteration by leaving a labeled statement. The *label_name* references the iteration statement with which the label is associated, and on which LEAVE is to be performed.
Compound Statement	
label_name	the label for a BEGIN-END compound statement in the stored procedure, or for an iteration statement (WHILE, LOOP, FOR and REPEAT). The beginning label must be suffixed with a COLON character (:). An ending label is not mandatory, but if an ending label is specified, you must also specify an equivalent beginning label. The label name of the BEGIN-END compound statement cannot be reused in an iteration statement. A label name cannot be reused within a group of nested compound statements or nested iteration statements, but can be reused for different non-nested iteration statements or non-nested compound statements. Using label names for each BEGIN-END compound statement is recommended if you specify nested compound statements in a stored procedure. A local variable can be optionally qualified with the label of the corresponding compound statement in which the variable is declared. This helps in avoiding conflicts that might be caused by duplicate local variables in nested compound statements.

Syntax Element …	Specifies …
BEGIN	keyword introducing the compound statement which contains the stored procedure body, or a handler action. The compound statement defining the stored procedure body contains all the SQL statements and declaration statements of the stored procedure. You must terminate a compound statement with the END keyword.
statement_list	any of the following: • SQL DML, DDL, or DCL statements supported by stored procedures. These include dynamic SQL statements. You can also use the CALL statement invoking the *procedure_name* being created. • control statements, including the BEGIN-END compound statement. You can use any number of statements in a *statement_list*. Each must be terminated by a SEMICOLON character.
END	ending keyword of the BEGIN-END compound statement.
Local Declaration	
DECLARE	keyword introducing a local variable declaration, cursor declaration or condition handler declaration statement. DECLARE is followed by a list of local variables, cursor specification or a list of handler declarations. You can specify multiple local variable declarations, cursor declarations or condition handler declarations for each stored procedure. Each time the DECLARE keyword is specified, it must be followed by at least one local variable declaration, cursor declaration or condition handler declaration.
variable_name	name of the local variable being declared. Any number of local variables of the same data type can be specified as a comma-separated list. These variable names are replaced with their values during stored procedure execution. When used in an SQL statement other than a control statement, a local variable must be prefixed with a COLON (:) character.

Syntax Element …	Specifies …
DEFAULT	optional keyword for introducing a default value for the local variables. If more than one local variable is specified along with a default value, that value applies to all of the local variables in the list.
literal NULL	the default value for the variables. This must be a literal. A DEFAULT clause cannot contain an expression. The literal must be compatible with the data type specified. A variable is initialized to NULL if no default value is specified.
Cursor Declaration	
cursor_name	the name of the cursor.
SCROLL NO SCROLL	the cursor scrollability option. The SCROLL option allows the cursor to scroll forward to the next row or back to the first row of the response set. NO SCROLL is the default option. When NO SCROLL is specified or when no option is specified, the cursor can only scroll forward to the next row.
CURSOR FOR	keywords qualifying the *cursor_name* as the cursor for the statement set to follow.
cursor_specification	the SELECT statement specifying the column names or expressions from which to fetch data rows in the CURSOR FOR clause.
FOR READ ONLY FOR UPDATE	the updatability clause. The FOR UPDATE option makes the cursor updatable, and UPDATE or DELETE can be performed on the cursor. If FOR UPDATE is not specified, the cursor is read only. FOR READ ONLY is the default option and is implicit.
Condition Handler	
CONTINUE EXIT	the type of condition handler being requested.
HANDLER	keyword introducing the condition handler declaration.
FOR	keyword introducing the FOR clause.

Syntax Element …	Specifies …
SQLSTATE	keyword introducing an *sqlstate* value for which the exception or completion condition is to be handled. Any number of *sqlstate* values prefixed by "SQLSTATE [VALUE]" can be specified in a comma-separated list within each handler declaration.
VALUE	optional keyword prefixing the *sqlstate* value
sqlstate	the five-character string value that indicates the SQL statement condition. This is always specified within APOSTROPHE characters. A given *sqlstate* value cannot be a part of more than one DECLARE HANDLER statements and cannot be repeated within a FOR clause.
SQLEXCEPTION SQLWARNING NOT FOUND	keyword for generic conditions. You can specify these individually or in combination with one another, but you cannot repeat any within a given DECLARE HANDLER statement. SQLEXCEPTION indicates a generic exception condition. SQLWARNING indicates a generic completion condition. NOT FOUND indicates a generic completion condition when no data is found.
handler_action_ statement	either a single statement or a compound statement that serves as the handler action. A single statement can be any one of the following: • SQL DML, DDL, or DCL statement supported by stored procedures. This includes a dynamic SQL statement. • control statement. The following are not allowed as a single statement for *handler_action_statement*: local variable, cursor, or handler declarations.
Assignment Statement	
SET	keyword introducing the SET statement used for assigning a value to a variable or parameter.
assignment_target	the name of the variable or parameter to be assigned a value. If assigning to a parameter it must be an INOUT or OUT parameter. It cannot be an IN parameter. You cannot set values for status variables in stored procedures.

Syntax Element …	Specifies …
assignment source	arithmetic and string expressions that contain the value to be assigned to a variable. Stored procedure local variables, status variables, IN or INOUT parameters and FOR loop column and correlation names can be specified in the *assignment source,* also known as value expression.
Condition Statement	
CASE	keyword introducing the CASE conditional statement to qualify control statements. You can use either the Simple Case statement form or the Searched Case statement.
operand_1 *operand_2*	value expressions or arithmetic and string expressions. Stored procedure local variables, status variables, IN or INOUT parameters and FOR loop column and correlation names can be specified in the value expression.
THEN ELSE	keyword introducing the statements to be performed in the CASE statement.
statement	list of SQL and control statements.
conditional_expression	a boolean condition (also called search condition) used to evaluate whether a statement or statements embedded within an IF, CASE, REPEAT or WHILE should be performed. Arithmetical or logical expressions and status variables can be specified as conditional expressions. You cannot use IN and NOT IN operators if the conditional list contains any local variables, parameters, or cursor correlation names. OUT parameters and subqueries are not allowed in *conditional_expressio*n.
END CASE	keywords marking the end of the CASE statement
IF	keyword introducing a *conditional expression* to qualify SQL control statements. You can use the following variations of IF with additional clauses: IF-THEN-END IF IF-THEN-ELSE-END IF IF-THEN-ELSEIF-END IF IF-THEN-ELSEIF-ELSE-END IF

Syntax Element …	Specifies …
THEN ELSE ELSEIF	keyword introducing the THEN, ELSE, or ELSEIF clause in the IF statement.
END IF	keywords marking the end of the IF statement.
	Iteration Statement
WHILE	keyword introducing an iteration statement to repeat the execution of one or more statements within the defined WHILE statement. The associated condition is checked before each iteration (including the first) and if true, the statements are performed. Otherwise, the WHILE statement completes with no further iterations.
DO	keyword introducing the sequence of statements to be performed in the WHILE statement.
END WHILE	keywords indicating the ending of the sequence of statements in the WHILE statement.
LOOP	keyword introducing an iteration statement that repeats the execution of one or more statements embedded within the defined loop.
END LOOP	keywords ending the LOOP statement
FOR	keyword introducing an iteration statement to repeat the execution of one or more statements for each row fetched by the cursor.
for_loop_variable	the name of the FOR loop.
cursor_name	the name of the cursor.
CURSOR FOR	optional keywords qualifying the *cursor_name* as the cursor for the statement set to follow.
cursor_specification	the SELECT statement specifying the column names or expressions from which to fetch data rows in the CURSOR FOR statement.
DO	keyword introducing the sequence of statements to be performed in the CURSOR FOR statement.
END FOR	keywords ending the CURSOR FOR statement.
REPEAT	keyword introducing the iteration statement that repeats the execution of one or more statements in a loop until the conditional expression specified in the UNTIL clause evaluates to true.
UNTIL	keyword introducing a *conditional_expression* to qualify the statements in the loop.

Syntax Element …	Specifies …
END REPEAT	keywords ending the REPEAT statement.
Cursor Specification	
SELECT	that the column names or expressions specified next form the cursor.
column_name	name of the column from which to fetch data.
AS	an optional introduction to *correlation_name*.
correlation_name	an alias for the table that is referenced by *table_name*. *correlation_name* is also used to name expressions. The ANSI SQL standard refers to aliases as correlation names. They are also known as range variables.
table_name	the name of a table or view from which rows are to be fetched.
expression	any valid SQL expression.
WHERE clause	an optional search condition that must be satisfied by the row or rows to be fetched by the SELECT statement.
other SELECT clauses	clauses such as ORDER BY, GROUP BY and HAVING. These are allowed only in read-only cursors. None of these clauses is allowed in updatable cursors. WITH...BY clause is not allowed in stored procedure cursors.

ANSI Compliance

CREATE PROCEDURE is ANSI SQL-99-compliant with extensions.

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

Authorization: CREATE PROCEDURE

To use CREATE PROCEDURE, you need the CREATE PROCEDURE privilege on the database in which the procedure is created.

User DBC, or the owner of a database having the CREATE PROCEDURE privilege WITH GRANT OPTION on that database, can grant this privilege to you.

Authorization: REPLACE PROCEDURE

You must have DROP PROCEDURE permission on the named procedure or the containing database to perform REPLACE PROCEDURE on an existing stored procedure.

You must have CREATE PROCEDURE permission on the specified database or the current default database to perform REPLACE PROCEDURE to create a new stored procedure.

Utilities Supporting CREATE/REPLACE PROCEDURE

You can create a stored procedure through the COMPILE command in BTEQ and TeqTalk (DMTEQ). See *Teradata BTEQ Reference* for details.

You can also create a stored procedure using the CREATE PROCEDURE statement from CLIv2, ODBC, and JDBC applications, and from the QueryMan utility.

Stored procedure execution and DDL operations are supported by the following client utilities:

- BTEQ
- CLIv2
- JDBC
- ODBC
- PP2
- QueryMan
- TeqTalk (DMTEQ)

For details on the use of local variables and condition handlers in stored procedures, see the documentation for the DECLARE and DECLARE HANDLER statements in the chapter “SQL Stored Procedures” in *Teradata RDBMS SQL Reference, Volume 6*.

Stored Procedure Options

Stored procedures have one user-specified attribute: source text storage option, as detailed in the following table:

This option ...	Does the following ...
SPL	Stores the DDL definition source text of the stored procedure in Teradata. SPL is the default option.
NO SPL	Does not store the source text of the stored procedure in Teradata.

This option is specified in different ways in different client utilities and interfaces from which the stored procedure is created, as described in the following table. See the appropriate client manuals for details.

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.
JDBC	with the following call pairs: • `getSPLOption()` • `setSPLOption()`
ODBC	as a parameter specified in different ways depending on the platform.

FOR this platform ...	AS a parameter in the ...
UNIX	`.odbc.ini` file.
Windows	ODBC Setup Dialog box for ODBC Driver.

The user-specified option *[no] warning*, available with the ALTER PROCEDURE statement, enables suppression of all stored procedure compilation warnings. See "ALTER PROCEDURE" on page 1-3 for details.

Creating Stored Procedures

Use the following procedure to create a stored procedure. Steps 4, 5 and 6 apply only when a compound statement is specified.

Step	Action
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 three elements, in the following order: • Parameter type (optional) • Parameter name (mandatory) • Data type of the parameter (mandatory)

Step	Action
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 to complete the CREATE PROCEDURE statement. If you have started specifying 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 Stored Procedure Creation

On successful completion of the stored procedure object creation, a SUCCESS (or OK in field mode) parcel is returned to the client application with the activity count set to zero, indicating that there were no compilation errors.

The following statements need to be specified to complete the DDL operation of creating the stored procedure, depending on the session mode:

Session mode	Statements required
ANSI	An explicit COMMIT statement.
Teradata	An explicit END TRANSACTION statement if the transaction is explicitly open. No such statement is required if the transaction is implicit.

Aborting Creation of a Stored Procedure

The creation of a stored procedure in an open transaction can be aborted by using the SQL ABORT or ROLLBACK statement.

Processing of CREATE PROCEDURE Statement

A stored procedures compiler in Teradata performs syntax and semantic checks on the source text of the stored procedure.

The processing by the Parser involves the following steps, in addition to the routine for creation of any database object:

Step	Action
1	The Teradata server validates the SQL statements, including any expressions, inside the stored procedure body.
2	The errors or warnings encountered during validation are treated as stored procedure compilation errors or warnings. These are reported to the client utility as part of the SUCCESS or OK parcel.
3	On successful creation of a stored procedure, a SUCCESS (or OK in field mode) parcel is returned to the client application, with warning code = 0.
4	When a failure is encountered, a FAILURE parcel is returned. An error is treated as a failure. The current transaction is rolled back.

Retrieving a Stored Procedure Body

Use the SHOW PROCEDURE statement to display stored procedure source text. The text is returned in the same format as defined by the creator.

Use new line characters wherever possible in the source text to increase its readability.

What REPLACE PROCEDURE Does

REPLACE PROCEDURE effectively acts as a DROP PROCEDURE followed by a CREATE PROCEDURE. All privileges that were granted directly on the original stored procedure are retained.

If the specified stored 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 stored 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 stored procedure created in Teradata mode is being replaced in ANSI mode, it acquires the new session mode attributes and thereafter can only be performed in ANSI mode.

Similarly, if a stored procedure created in ANSI mode is replaced in Teradata mode, it acquires the new session mode attributes and thereafter can only be performed in Teradata mode.

You can replace a stored procedure created with one default character set with a different default character set.

If a stored procedure originally created on Teradata for NCR UNIX MP-RAS has been restored on Teradata for Windows, or vice versa, it can be replaced on the new platform.

Example 1

The following example creates a valid stored procedure named spSample1 with the following parameters and local variables:

Variable Type	Name
IN	ip
OUT	op
local variable	var1

Because the parameter type for ip is not specified, the default type IN applies. This stored procedure does not contain any condition handlers.

```
CREATE PROCEDURE spSample1(ip INTEGER, OUT op INTEGER)
BEGIN DECLARE var1 INTEGER;
  SELECT col1 INTO :var1
  FROM tab1
  WHERE col2 = :ip;
  SET op = var1 * 10;
END;
```

Example 2

The following example is for creating a valid stored procedure named spParams with six parameters of different data type combinations.

```
CREATE PROCEDURE spParams (
        INOUT iop1 SMALLINT,
        OUT op1 DECIMAL(10,3),
        IN ip2 NUMERIC,
        INOUT iop2 FLOAT,
        OUT op2 REAL,
        IN ip3 DOUBLE PRECISION)
BEGIN
     SET iop1=3;
     SET op1=5.2;
     SET iop2=10.2;
     SET op2=iop2;
END;
```

Example 3

The following example shows the CREATE PROCEDURE statement for creating a stored procedure named spSample3 with local variable declarations, cursor declarations, and condition handler declarations.

```
CREATE PROCEDURE spSample3(OUT p1 CHAR(80))
BEGIN
   DECLARE i INTEGER;
   DECLARE EmpCursor CURSOR FOR
      SELECT EmpName, Salary FROM EmpDetails ORDER BY DeptCode;
   DECLARE DeptCursor CURSOR FOR
      SELECT DeptName from Department;

   DECLARE EXIT HANDLER
      FOR sqlstate '42000'
      BEGIN
       OPEN EmpCursor;
       SET p1='FAILED TO INSERT RECORD';
      END;
   DECLARE CONTINUE HANDLER
      FOR SQLSTATE value '23505'
      BEGIN
       SET p1='FAILED TO INSERT RECORD';
      END;
   DECLARE CONTINUE HANDLER
      FOR SQLEXCEPTION
      BEGIN
        SET p1='FAILED TO INSERT RECORD';
      END;

   INSERT INTO table1 VALUES(1000,'aaa');
END;
```

Example 4: Stored Procedures With a Single Statement

The following are examples of stored procedures with a single statement that is not enclosed in a BEGIN - END compound statement.

- A stored procedure with a single SET statement:

```
CREATE PROCEDURE spSample4a (IN InParam INTEGER,
                             OUT OutParam INTEGER)
SET OutParam = InParam + 1;
```

- A stored procedure with a WHILE statement:

```
CREATE PROCEDURE spSample4b (IN InParam INTEGER,
                             OUT OutParam INTEGER)
WHILE (InParam < 20)
    SET OutParam = InParam + 1;
END WHILE;
```

Example 5

The following example shows nesting of compound statements.

The local variable and cursor name of the outer compound statement labeled L1 are used in the inner compound statement labeled L2.

```
CREATE PROCEDURE spSample1(IN pName CHAR(30), INOUT pAmt INTEGER)
L1: BEGIN
   DECLARE cvar1, vAmt INTEGER;
   DECLARE cursor1 CURSOR FOR
       SELECT c1 as c_c1, c2 as c_c2 FROM temp;
   L2: BEGIN
       DECLARE vName CHAR(30);
       DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
          BEGIN
          OPEN cursor1;
          INSERT INTO temp VALUES (vName, vAmt);
          END;
       SET vName = pName;
                -- tab1 table does not exist
       INSERT INTO tab1 VALUES (vName, vAmt);
        FETCH cursor1 INTO cvar1, pAmt;
   END L2;
END L1;
```

Related Information

See the following chapter and statements for related information:

- "SQL Stored Procedures" in *Teradata RDBMS SQL Reference, Volume 6*
- "ALTER PROCEDURE" on page 1-3

CREATE PROFILE

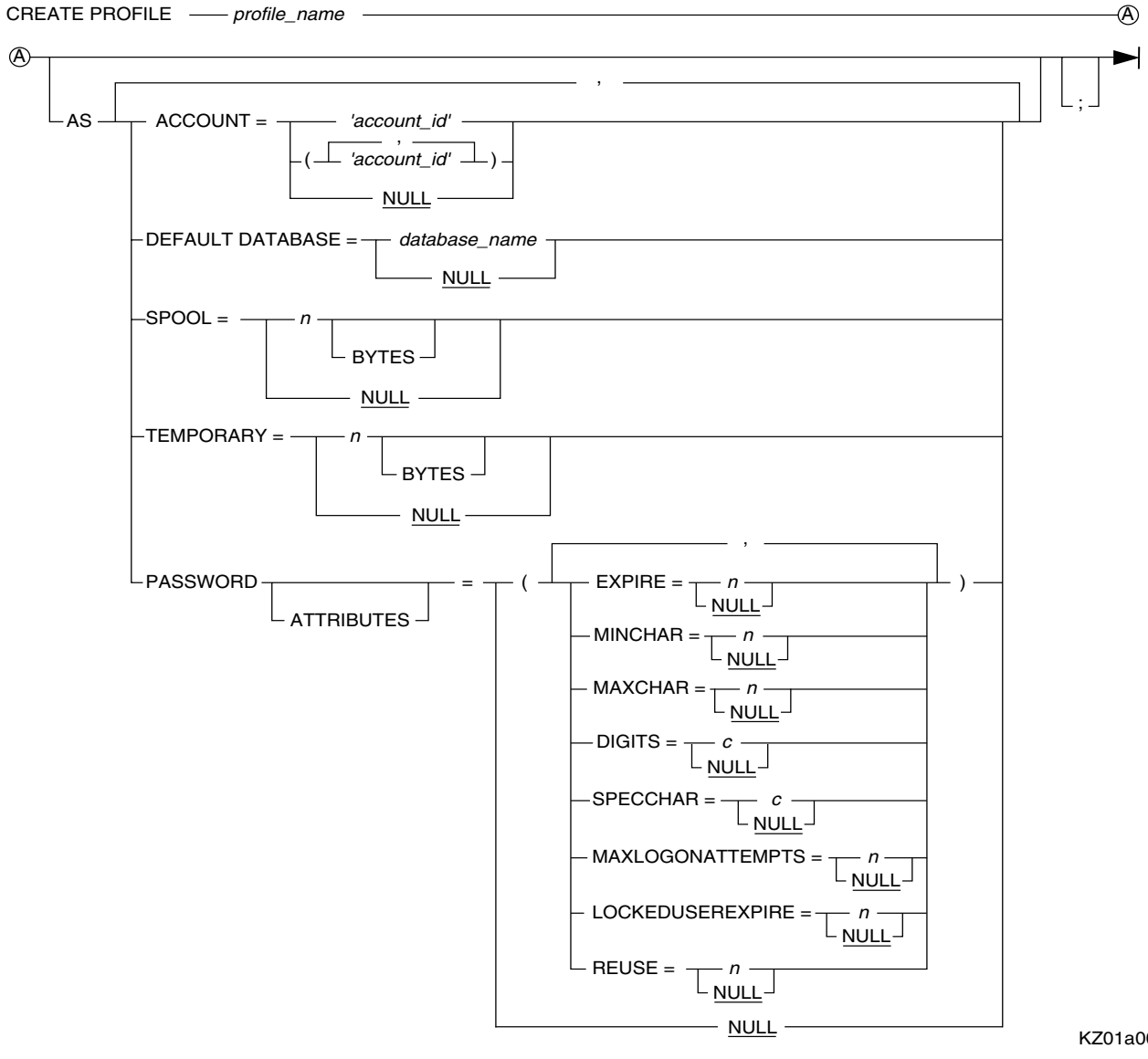
Purpose

Creates a new profile that defines system parameters that a group of users can share.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
profile_name	the name of the new profile. A profile name must be unique among profiles.
ACCOUNT	a keyword allowing you to specify the account for a user with this profile. If no account is defined in the profile assigned to a user, Teradata uses the setting defined for the user.
account_id	an APOSTROPHE character-quoted identifier or a list of identifiers of the accounts to charge for the space used by users with this profile. The first (or only) identifier becomes the default account for the user. If *account_id* is null or is not defined in the profile assigned to a user, Teradata uses the setting defined for the user. Each *account_id* can be up to 30 characters and must be enclosed by APOSTROPHE characters. A list of *account_ids* must be separated by commas and enclosed by LEFT PARENTHESIS and RIGHT PARENTHESIS. Accounts in a profile assigned to a user override any other accounts the user might have. You can associate a Priority Scheduler performance group with each account ID you specify. This assignment specifies the relative service priority for the account based on the Priority Scheduler parameters specified for it by the system administrator. Each performance group for an account is named using the syntax `$group_name$`. *group_name* is limited to 16 single-byte characters (or 16 bytes for multibyte character sets). Validation of a performance group name occurs only at logon. When you log on, Teradata validates the performance group name you specify against existing performance group names. If no match is found, Teradata assigns one to you by default. The default is one of the following, depending on the situation. • First account string defined for the profile. • First account string defined for the user. • Performance group M if the user has no account string with a defined performance group.

Syntax Element …	Specifies …

account_id (continued)	The following interactive mode BTEQ logon commands are valid examples of how to specify a performance group.
	`.LOGON elnino/GMK '$L2$&D&TacctPUB'`
	`.LOGON elnino/KGM '$L&D&TacctPUB'`
	Performance group names, their relative levels of service, and other related parameters are defined either by default or as individually customized variables by your system administrator. For more information about performance groups and other Priority Scheduler parameters, see "Priority Scheduler Facility" in *Teradata RDBMS Utilities*.

DEFAULT DATABASE	a keyword allowing you to specify the default database for a user with this profile. If no default database is defined in the profile assigned to a user, Teradata uses the setting defined for the user.
database_name	the name of the default database established each time a user with this profile logs onto Teradata. *database_name* need not be an existing database. Teradata returns an error when a user with this profile tries to create or reference an object within a nonexistent database. If the default database is NULL or is not defined in the profile assigned to a user, Teradata uses the setting defined for the user.
SPOOL	a keyword allowing you to specify the bytes to allocate to the spool file. If no spool space is defined in the profile assigned to a user, Teradata uses the setting defined for the user.
n [BYTES]	the number of bytes (*n*) to allocate for spool files. The default is NULL, which causes Teradata to use the setting defined for the user. *n* cannot exceed the spool space parameter in the profile of the creator. If no spool space is defined for that profile, then the spool space defined for the user setting of the creator is the criterion. *n* can be an integer, a decimal value, or a floating point value. *n* refers to bytes, whether or not the optional BYTES keyword is specified.

Syntax Element …	Specifies …
TEMPORARY	a keyword allowing you to define how much space to allocate for creating temporary tables. If no temporary space is defined in the profile assigned to a user, Teradata uses the setting defined for the user. Temporary space is reserved prior to spool space for any user defined with this characteristic. Disk usage for a materialized temporary table is charged to the temporary space allocation of the user who referenced the table.
n BYTES	the number of bytes (*n*) to allocate for temporary space. The default is NULL, which causes Teradata to use the setting defined for the user. *n* cannot exceed the temporary space parameter in the profile of the creator. If no temporary space is defined for that profile, then the temporary space defined for the user setting of the creator is the criterion. *n* can be an integer, a decimal value, or a floating point value. *n* refers to bytes, whether or not the optional BYTES keyword is specified.
PASSWORD [ATTRIBUTES]	a keyword allowing you to define password security attributes. The default value is NULL for password attributes that are not defined in the profile for a user, which causes Teradata to use the system setting. Password security attributes defined in a profile take effect the next time a user logs on.
EXPIRE = *n*	the number of days to elapse before the password expires. A value of 0 for *n* indicates the password never expires.
MINCHAR = *n*	the minimum number of characters in a password string. The valid range for *n* is 1 to 30.
MAXCHAR = *n*	the maximum number of characters in a password string. The valid range for *n* is 1 to 30.

Syntax Element …	Specifies …
DIGITS = *c*	whether digits can appear after the first character in a password string. Valid values for *c* are Y, y, N, or n.
SPECCHAR = *c*	whether special characters can appear in a password string. Valid values for *c* are Y, y, N, or n.
MAXLOGONATTEMPTS = *n*	the number of incorrect logon attempts allowed before locking the user. A value of 0 for *n* indicates never to lock the user.
LOCKEDUSEREXPIRE = *n*	the number of minutes to elapse before unlocking a locked user. A value of 0 for *n* unlocks the user immediately. A value of -1 for *n* locks the user indefinitely.
REUSE = *n*	the number of days to elapse before a password can be reused. A value of 0 for *n* allows the password to be reused immediately.

ANSI Compliance

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

Authorization

To use CREATE PROFILE, you must have the CREATE PROFILE privilege.

New users do not implicitly have the CREATE PROFILE privilege. User DBC or a user who has the CREATE PROFILE privilege WITH GRANT OPTION can grant this privilege to another user.

Ownership

The concept of ownership and ownership hierarchy that applies to database objects does not apply to profiles.

Using Profiles to Manage System Parameters for Groups

Profiles define system parameters. Assigning a profile to a group of users makes sure that all group members operate with a common set of parameters.

To manage system parameters for groups, a database administrator can:

- Create a different profile for each user group, based on system parameters that group members share.

  You can define values for all or a subset of the parameters in a profile. If you do not set the value of a parameter, Teradata uses the setting defined for the user in a CREATE USER or MODIFY USER statement.
- Assign profiles to users.

  The parameter settings in a user profile override the settings for the user in a CREATE USER or MODIFY USER statement.

Example 1

The following statement creates a profile called human_resources that defines the default database to be personnel:

```
CREATE PROFILE human_resources AS
  DEFAULT DATABASE = personnel;
```

The following statement assigns the human_resources profile to a new user:

```
CREATE USER marks AS
  PROFILE = human_resources;
```

Controlling User-Level Password Security

An administrator can create profiles that define values for the following user-level password security attributes:

- Minimum and maximum number of characters in a password string
- Whether or not to allow digits and special characters in a password string
- Number of incorrect logon attempts to allow before locking a user
- Number of minutes before unlocking a locked user
- Number of days before a password can be used again

The password attribute settings in a user profile override the corresponding system-level password settings. Teradata uses the system-level password setting for password attributes that are not defined in the profile for a user.

Changes to password attribute settings in a profile do not require a system restart to take effect. The new settings take effect the next time users who are assigned the profile log on.

Example 2

The following statement defines user-level password security attributes for a profile called human_resources:

```
CREATE PROFILE human_resources AS
  DEFAULT DATABASE =  personnel
  ,PASSWORD =
  (EXPIRE =           90
  ,MAXLOGONATTEMPTS = 10
  ,LOCKEDUSEREXPIRE = -1);
```

The following statement assigns the human_resources profile to user marks:

```
MODIFY USER marks AS
  PROFILE = human_resources;
```

For More Information

FOR more information on …	SEE …
assigning profiles to users	• "CREATE USER" on page 1-296 • "MODIFY USER" on page 1-364
changing system parameters for a profile	"MODIFY PROFILE" on page 1-359
obtaining the CREATE PROFILE privilege	"GRANT (SQL Form)" on page 2-22
system-level password attributes	"Controlling Access to Teradata RDBMS" in *Teradata RDBMS Security Administration*.

CREATE ROLE

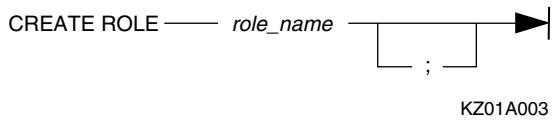
Purpose

Creates a new role for managing user access privileges on database objects.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
role_name	the name of the new role. A role can share the same name as a profile, table, column, view, macro, trigger, or stored procedure. However a role name must be unique among users, databases, and roles.

ANSI Compliance

CREATE ROLE is ANSI SQL-99-compliant.

Authorization

To use CREATE ROLE, you must have the CREATE ROLE privilege.

New users do not implicitly have the CREATE ROLE privilege. User DBC or a user who has the CREATE ROLE privilege WITH GRANT OPTION can grant this privilege to another user.

The role creator is automatically granted membership to the newly created role WITH ADMIN OPTION, which makes the role creator a member of the role who can grant membership to the role to other users and roles.

Ownership

The concept of ownership and ownership hierarchy that applies to database objects does not apply to roles.

Using Roles to Manage User Access Privileges

Roles define access privileges on database objects. Assigning a default role to a user gives the user access to all the objects that the role has privileges to. A default role that has members that are roles gives the user additional access to all the objects that the nested roles have privileges to.

To manage user access privileges, a database administrator can:

- Create different roles for different job functions and responsibilities.

  A newly created role does not have any associated privileges until grants are made to it.
- Grant specific privileges on database objects to the roles.
- Add members to the role.

  Members of a role can be users or other roles.

  Roles can only be nested one level. Thus, a role that has a role member cannot also be a member of another role.
- Assign default roles to users.

Example

The following statements create roles called management and administration:

```
CREATE ROLE management;
CREATE ROLE administration;
```

For More Information

FOR more information on ...	SEE ...
granting privileges on database objects to roles	"GRANT (SQL Form)" on page 2-22
obtaining the CREATE ROLE privilege	
adding members to roles	"GRANT (Role Form)" on page 2-19
assigning default roles to users	• "CREATE USER" on page 1-296 • "MODIFY USER" on page 1-364
changing the current role for a session	"SET ROLE" on page 1-382

CREATE TABLE

Purpose

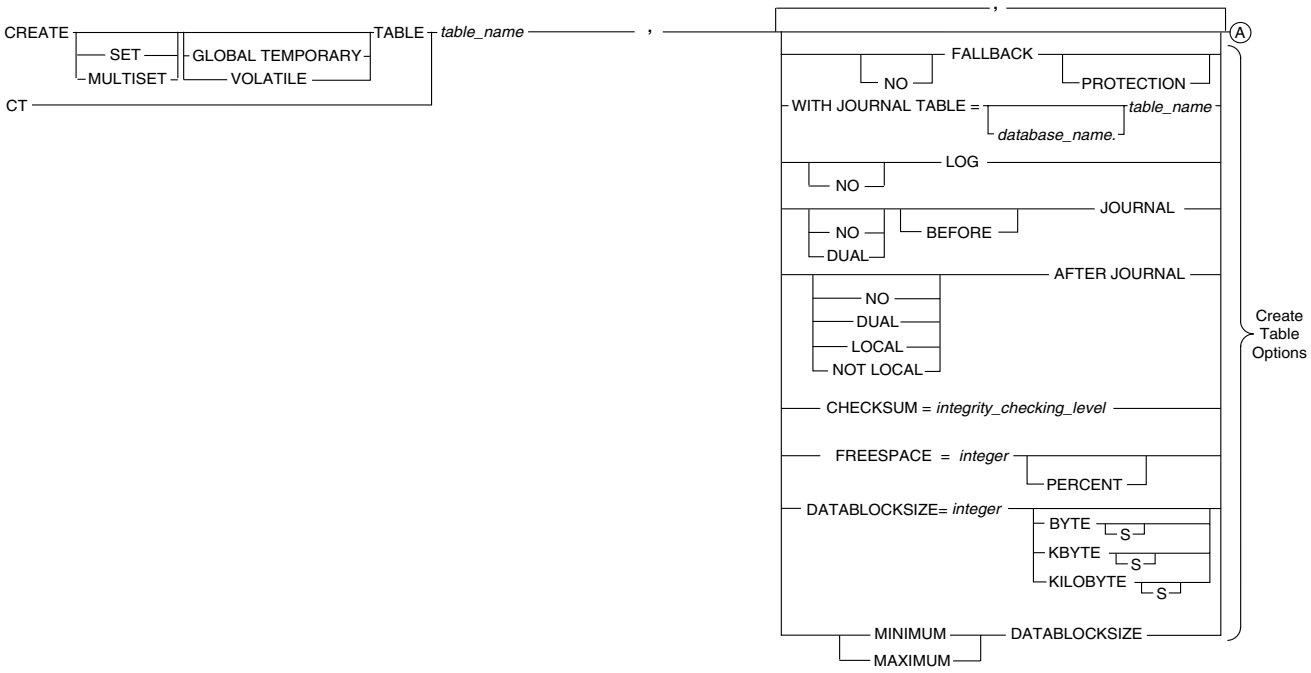
Defines the columns, primary and secondary indexes, constraints, and other attributes of a new table.

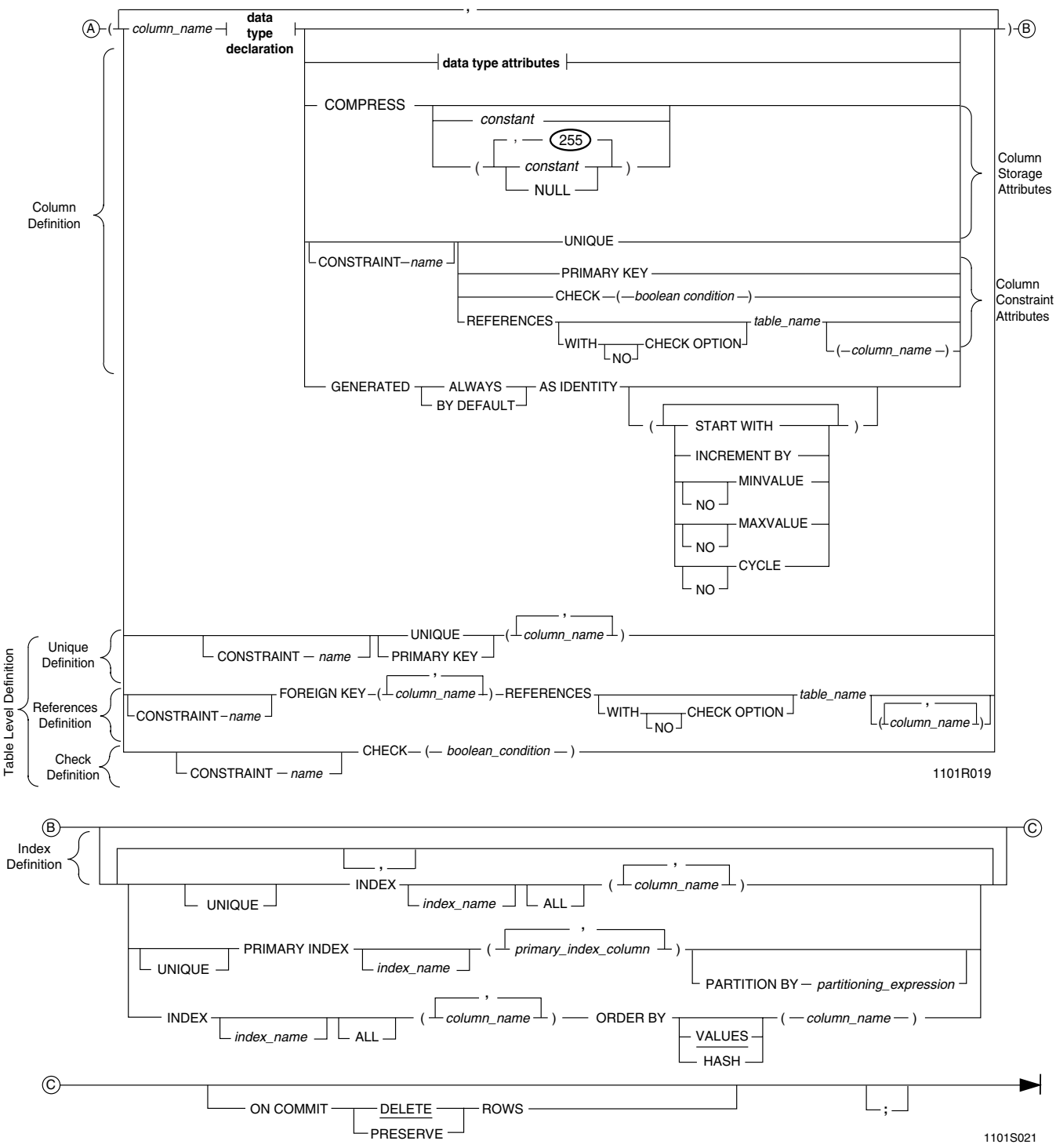
The Copy Table form copies selected column definitions, and, optionally data, from an existing table to a new table.

Invocation

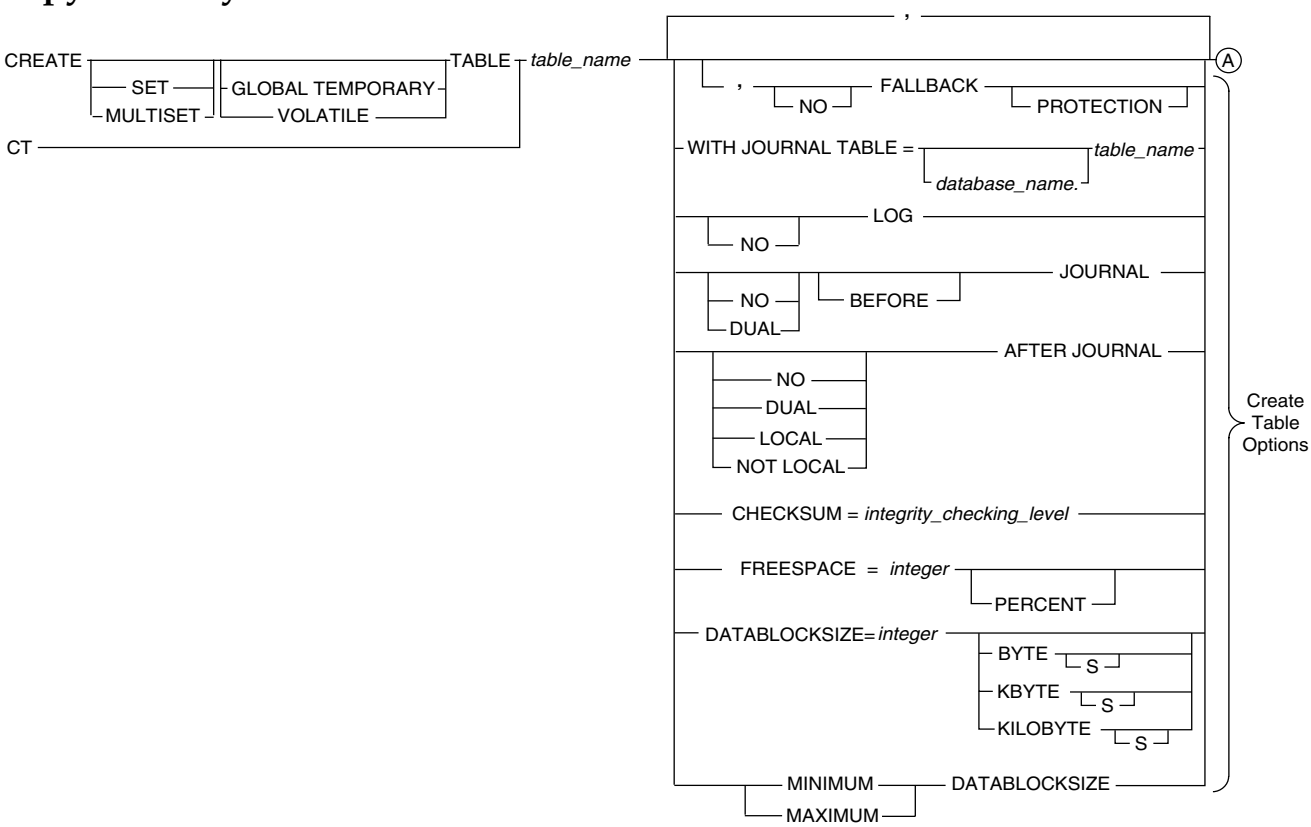
Executable.

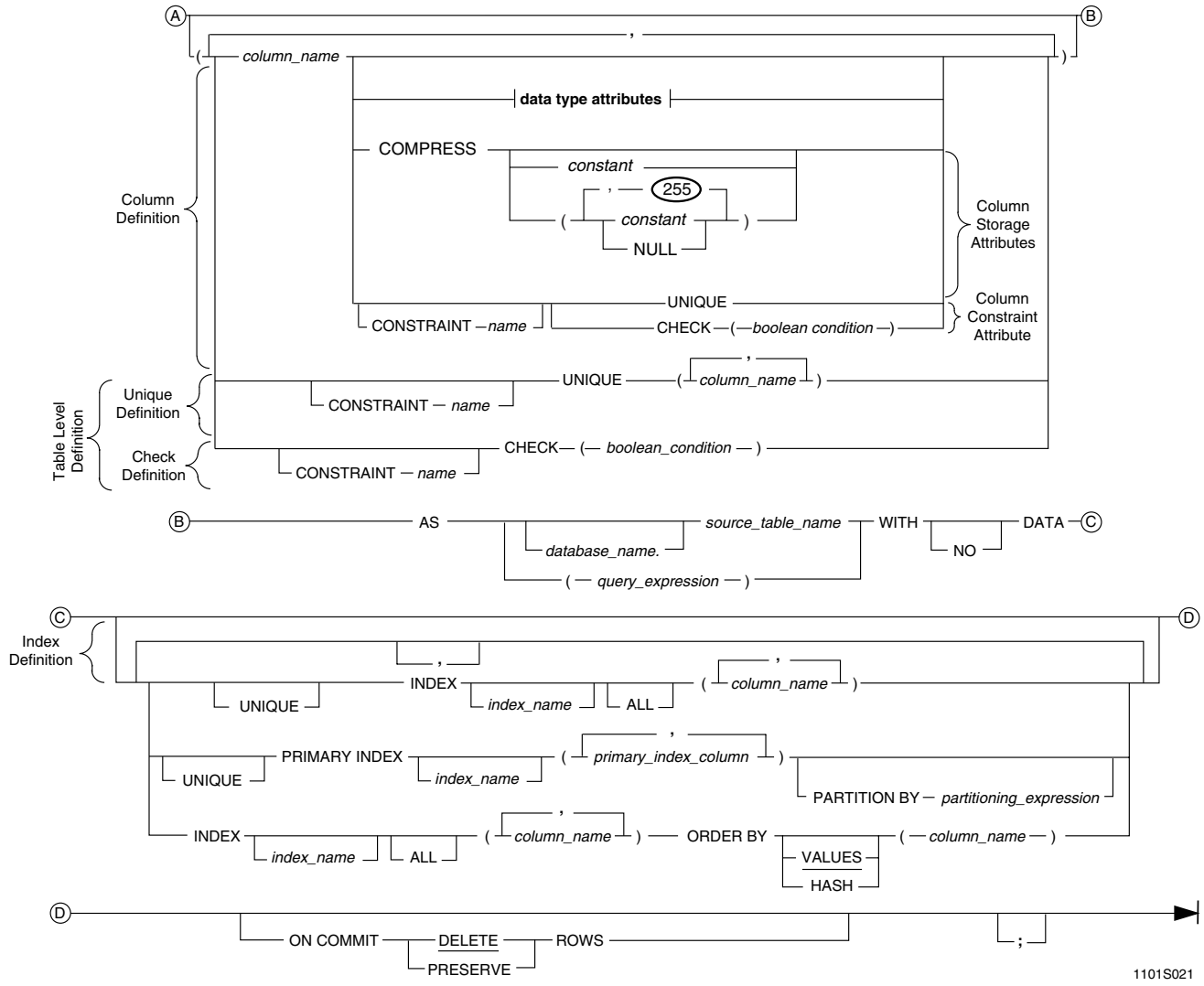
Create Table Syntax



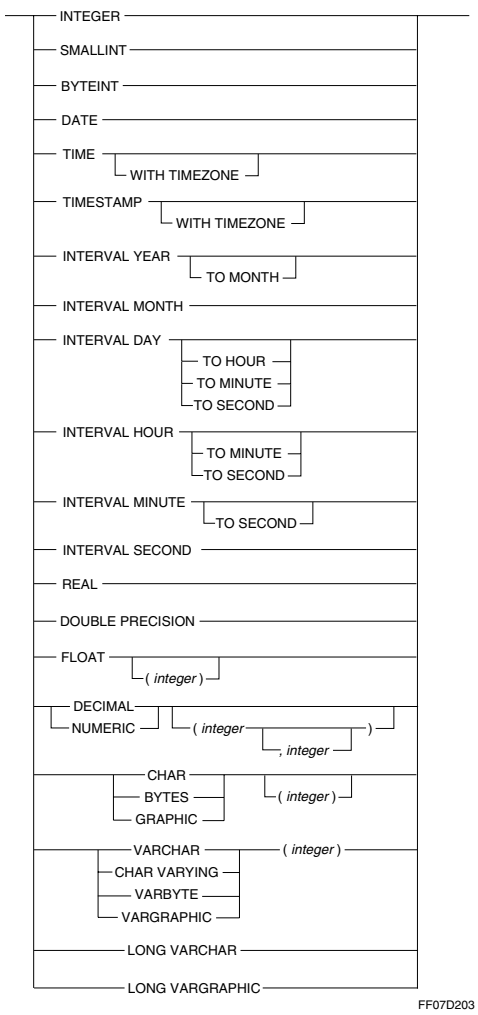

Column Definition
column_name
data type declaration
data type attributes
COMPRESS
constant
255
NULL
Column Storage Attributes
UNIQUE
CONSTRAINT name
PRIMARY KEY
CHECK (boolean condition)
REFERENCES
table_name
WITH NO CHECK OPTION
column_name
Column Constraint Attributes
GENERATED ALWAYS BY DEFAULT AS IDENTITY
START WITH
INCREMENT BY
MINVALUE
MAXVALUE
CYCLE
Table Level Definition
Unique Definition
References Definition
Check Definition
CONSTRAINT name
UNIQUE
PRIMARY KEY
FOREIGN KEY
REFERENCES
CHECK (boolean_condition)
1101R019
Index Definition
INDEX
index_name
ALL
PRIMARY INDEX
primary_index_column
PARTITION BY partitioning_expression
ORDER BY
VALUES
HASH
ON COMMIT
DELETE
PRESERVE
ROWS
1101S021

Copy Table Syntax

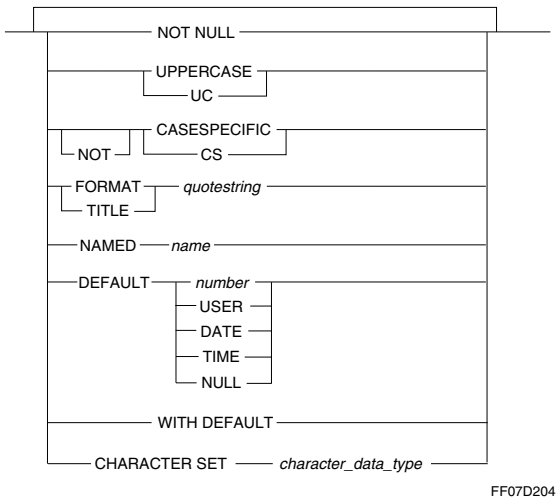



A
B
,
(column_name)
data type attributes
COMPRESS
constant
, 255
(constant)
NULL
Column Definition
Column Storage Attributes
UNIQUE
CONSTRAINT name
CHECK (boolean condition)
Column Constraint Attribute
Table Level Definition
Unique Definition
UNIQUE (column_name)
CONSTRAINT name
Check Definition
CHECK (boolean_condition)
CONSTRAINT name
B
AS
source_table_name
database_name.
(query_expression)
WITH
NO
DATA
C
Index Definition
D
UNIQUE
INDEX
index_name
ALL
(column_name)
PRIMARY INDEX
primary_index_column
PARTITION BY partitioning_expression
ORDER BY
VALUES
HASH
ON COMMIT
DELETE
PRESERVE
ROWS
;
1101S021

Data Type Declarations



Data Type Attributes



where:

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td colspan="2">Table Kind Clause</td></tr>
<tr><td>MULTISET
SET</td><td>duplicate row control.

If there are uniqueness constraints on any column or set of columns in the table definition, then the table cannot have duplicate rows even if it is declared as MULTISET.

Some client utilities have restrictions with respect to MULTISET tables. See “SET and MULTISET” on page 1-201.

<table>
<tr><th>IF this keyword is specified ...</th><th>THEN duplicate rows are ...</th></tr>
<tr><td>SET</td><td>not allowed.

This is compatible with Teradata tables from prior releases.</td></tr>
<tr><td>MULTISET</td><td>allowed, in compliance with the ANSI SQL-99 standard.</td></tr>
<tr><td>none</td><td>a check is run on the mode the session is running in and the defaults described in the following table are assigned according to the session mode.</td></tr>
</table>

<table>
<tr><th>For this session mode ...</th><th>The default table kind is ...</th></tr>
<tr><td>ANSI</td><td>MULTISET</td></tr>
<tr><td>Teradata</td><td>SET</td></tr>
</table>
</td></tr>
<tr><td>GLOBAL TEMPORARY
VOLATILE</td><td>whether the table to be created is a global temporary table or a volatile table.

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

VOLATILE specifies that a volatile table be created, with its definition retained in memory only for the course of the session in which it is defined.</td></tr>
<tr><td colspan="2">Table Name Clause</td></tr>
<tr><td>table_name</td><td>the name of the new table.

If the name is not fully qualified, then Teradata assigns the name of the default database for the current session.</td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td colspan="2">Table Options Clause</td></tr>
<tr><td>[NO] FALLBACK</td><td>duplicate copy protection (or not) for the table.

When FALLBACK is chosen, then duplicate copies of rows in the table are created and stored.

The default for this option is established by a CREATE DATABASE, CREATE USER, LOCKING Modifier, or MODIFY USER statement for the database in which the table is to be created.</td></tr>
<tr><td>PROTECTION</td><td>a dummy word that can be specified after the FALLBACK keyword.</td></tr>
<tr><td>WITH JOURNAL TABLE = database_name.
table_name</td><td>the journal table for the data table being created. table_name need not reside in the same database as the data table.

<table><tr><th>IF a database is …</th><th>THEN …</th></tr><tr><td>specified</td><td>it must exist and table_name must have been defined as its default journal name.</td></tr><tr><td>not specified</td><td>the default database for the current session is assumed and table_name must have been defined as the default journal table.</td></tr></table>
table_name can reside in the same database as the table being created or in a different database.

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

If a default journal table was defined for the database, then this clause can be used to override the default</td></tr>
<tr><td>[NO] LOG</td><td>whether logging occurs.

LOG specifies that any updates, inserts, or deletes made to the global temporary or volatile table be logged in the transaction journal. This is the default.

NO LOG specifies that transaction journal logging is not to be performed. Because Teradata does not do any transaction journaling, performance is greatly enhanced.

If any sort of abort or restart occurs and the table is defined as NO LOG, then any updates, inserts, or deletes made to the temporary or volatile table cannot be recovered.

If the table is defined as NO LOG, contents of any materialized temporary table or any volatile table are emptied when a transaction aborts.

This option pertains to global temporary and volatile tables only.</td></tr>
</table>

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td>[NO] BEFORE JOURNAL

DUAL BEFORE JOURNAL</td><td>the number of before change images to be maintained.

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

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

Permanent journaling is not supported for temporary or volatile tables.</td></tr>
<tr><td>[NO] AFTER JOURNAL

DUAL AFTER JOURNAL

LOCAL AFTER JOURNAL

[NOT] LOCAL AFTER JOURNAL</td><td>the type of after-image to be maintained for the table; any existing images are not affected until the table is updated.

NO, DUAL, LOCAL, or NOT LOCAL: the NO and DUAL options specify the number of after-change images to be maintained for the table.

NOT LOCAL and LOCAL specify whether single after-image journal rows for non-fallback data tables are written on the same virtual AMP (LOCAL) as the changed data rows, or on another virtual AMP in the cluster (NOT LOCAL).

Permanent journaling is not supported for global temporary or volatile tables.

Also see “CREATE DATABASE” (“Local Journaling” on page 1-83) and “ALTER TABLE” (“LOCAL Journaling” on page 1-29).</td></tr>
<tr><td>[BEFORE] JOURNAL

AFTER JOURNAL</td><td>the type of image to be maintained for the table.

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

Permanent journaling is not supported for global temporary or volatile tables.

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

<table>
<tr><th>FOR this keyword ...</th><th>THIS type of journal image is maintained ...</th></tr>
<tr><td>BEFORE or AFTER</td><td>only the default for that type is overridden.

For example, if AFTER is specified, before-image journaling remains at the default setting.</td></tr>
<tr><td>BEFORE and AFTER</td><td>both, but the two specifications must not conflict with one another.</td></tr>
<tr><td>neither</td><td>both.</td></tr>
</table>
</td></tr>
<tr><td>FREESPACE integer</td><td>the percent of free space that will remain on a cylinder during loading operations.

Specifies the value of the percent freespace attribute to the specified value integer.

If the specified value does not fall within the allowable range (0 to 75 percent), an error message is generated.</td></tr>
<tr><td>PERCENT</td><td>a noise keyword. It has no effect.</td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>DATABLOCKSIZE=
integer BYTES/KBYTES/
KILOBYTES</td><td>the maximum data block size for blocks that contain multiple rows as the value integer.
BYTES can be abbreviated as BYTE. KILOBYTES can be abbreviated as KBYTE.
If DATABLOCKSIZE is not specified, then data blocks default to the sizes set in the PermDBSize and JournalDBSize fields of the DBSControl record.
<table><tr><th>The default value for this type of data block …</th><th>Is specified using this field of the DBSControl record …</th></tr><tr><td>permanent</td><td>PermDBSize</td></tr><tr><td>transient and permanent journal</td><td>JournalDBSize</td></tr></table>
See Teradata RDBMS Utilities, Volume 1 A-K for more detailed information about setting default data block sizes.
For details about data block size, see “DATABLOCKSIZE” on page 1-209.</td></tr>
<tr><td>MINIMUM
DATABLOCKSIZE</td><td>the smallest possible DATABLOCKSIZE setting for the table.
MINIMUM DATABLOCKSIZE sets the maximum data block size for blocks that contain multiple rows to the minimum legal value of either 6144 or 7168 bytes (12 or 14 sectors), depending on the cylinder size set for your system.
See “DATABLOCKSIZE” on page 1-209.
You can abbreviate MINIMUM as MIN.</td></tr>
<tr><td>MAXIMUM
DATABLOCKSIZE</td><td>the largest possible DATABLOCKSIZE setting for the table: 130 560 bytes (255 sectors).
You can abbreviate MAXIMUM as MAX.</td></tr>
<tr><th colspan="2">Column Definition Clause</th></tr>
<tr><td>column_name</td><td>specifies the name of one or more columns, in the order in which they and their attributes are to be defined for the table. Up to 2 048 columns can be defined for a table.</td></tr>
<tr><td>data_type_declaration
data_type_attributes</td><td>one or more data definition phrases that define data for the column.
A single data type must be specified for each column_name variable except for those defined in a CREATE TABLE … AS statement. It is not valid to specify data types for such columns.
Column data attribute specifications are optional. If you specify attributes for a column, you should define its data type prior to defining the attributes.
If you do not specify explicit formatting, a column assumes the default format for the data type, which can be specified by a custom data formatting specification (SDF) defined by the tdlocaledef utility (see Teradata RDBMS Utilities). Explicit formatting applies both to the parsing and to the retrieval of character strings.
Data types and data type attributes are the subject of Teradata RDBMS SQL Reference, Volume 3.
For information about converting data between data types, see the chapter ”Data Type Conversions” in Teradata RDBMS SQL Reference, Volume 5.</td></tr>
</table>

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td colspan="2">Column Storage Attributes Clause
Column storage attributes are a Teradata extension to the ANSI SQL-99 specification.</td></tr>
<tr><td>COMPRESS</td><td>a specified set of distinct values in a column that is to be compressed to zero space.
See the chapter "Database-Level Considerations" in Teradata RDBMS Database Design for a detailed description of value compression.
See the chapter “Default Value Control Phrases” in Teradata RDBMS SQL Reference, Volume 3 for more information about the COMPRESS attribute.</td></tr>
<tr><td>NULL</td><td>that nulls are compressed.
You can only specify NULL once per column.</td></tr>
<tr><td>constant</td><td>that nulls and the value are compressed.
Constants are typically quoted character strings. For numeric columns, constants are numeric.
The most space a numeric value can take is eight bytes.
Using COMPRESS on fixed-length character data can save space depending on the percentage of rows for which the compressed value is assigned.
See "COMPRESS Phrase" in the chapter "Default Value Control Phrases" in Teradata RDBMS SQL Reference, Volume 3 for information about limits for this value.</td></tr>
<tr><td colspan="2">Identity Column Specification
The identity column is ANSI SQL-99-compliant.</td></tr>
<tr><td>ALWAYS</td><td>that identity column values are always system-generated.
You cannot insert values into, nor can you update, an identity column defined as GENERATED ALWAYS AS IDENTITY.</td></tr>
<tr><td>BY DEFAULT</td><td>that identity column values can be system-generated or user-inserted, depending on the circumstance.
<table>
<tr><th>IF this is specified for insertion into the column ...</th><th>THEN ...</th></tr>
<tr><td>a null</td><td>Teradata generates an identity column value.</td></tr>
<tr><td>a value</td><td>the user-specified value is inserted into the identity column.</td></tr>
</table></td></tr>
<tr><td colspan="2">Identity Column Parameters
The five identity column parameters are optional and can be specified in any order.</td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>START WITH</td><td>Defines the lowest number in the system-generated numeric series for an identity column. The default is 1.

The value can be any exact negative or positive whole number within the range of the data type for the column as long as it is less than MAXVALUE for an incremental series or greater than MINVALUE for a decremental series.</td></tr>
<tr><td>INCREMENT BY</td><td>Defines the (possibly negative) interval on which to increment system-generated numbers. The default is 1.

The value can be any whole number less than or equal to the value of DBSControl field IdCol Batch Size except 0.</td></tr>
<tr><td>[NO] MINVALUE</td><td>Defines the minimum value to which a system-generated numeric series can decrement.

The value can be any whole number with a value less than the value specified for START WITH.

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

<table>
<tr><th>IF …</th><th>THEN …</th></tr>
<tr><td>INCREMENT BY is positive and CYCLE is specififed</td><td>renumbering begins from MINVALUE when MAXVALUE is reached.</td></tr>
<tr><td>INCREMENT BY is negative</td><td>MINVALUE, if specified, must be a whole number such that MINVALUE < START WITH.

MINVALUE also cannot be smaller than the minimum value for the data type assigned to the identity column.</td></tr>
<tr><td>NO CYCLE is specified</td><td>MINVALUE is not applicable for positive increments.

No warning or error is returned if you specify a MINVALUE with NO CYCLE.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td>[NO] MAXVALUE</td><td>Defines the maximum value to which a system-generated numeric series can increment.

Can be any whole number with a value greater than the value specified for START WITH.

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

If the sum of the specified values for START WITH and INCREMENT BY exceeds the specified value for MAXVALUE, and cycling is not enabled, then only one number is generated before the maximum limit is exceeded.

<table>
<tr><th>IF ...</th><th>THEN ...</th></tr>
<tr><td>INCREMENT BY is positive</td><td>MAXVALUE, if specified, must be a whole number such that MAXVALUE > START WITH.

MAXVALUE also cannot be larger than the maximum value for the data type assigned to the identity column.</td></tr>
<tr><td>INCREMENT BY is negative and CYCLE is specified</td><td>renumbering begins with MAXVALUE when MINVALUE is reached.</td></tr>
<tr><td>NO CYCLE is specified</td><td>MAXVALUE is not applicable for negative increments.

No warning or error is returned if you specify a MAXVALUE with NO CYCLE.</td></tr>
</table>
</td></tr>
<tr><td>[NO] CYCLE</td><td>Defines whether system-generated values can be recycled when their minimum or maximum is reached.

The default is NO CYCLE.</td></tr>
</table>

Column Constraint Attributes Clause

A column listed as UNIQUE or PRIMARY KEY must be declared NOT NULL.

CONSTRAINT *name*	the name for a named constraint. See "Constraint Names" on page 1-224.
UNIQUE	that no two rows in the table can have the same value in the field. A unique primary or secondary index will be used to enforce this constraint. Used on a single column. To specify UNIQUE on multiple columns, use the UNIQUE definition clause. Teradata implicitly creates a unique secondary index on any column specified as UNIQUE.

Syntax Element …	Specifies …
PRIMARY KEY	that no two rows in the table can have the same value for the defined column set. A unique primary or secondary index is used to enforce this constraint. Only one primary key can be specified per table. To specify candidate primary keys for referential integrity relationships with other tables, use the UNIQUE definition clause. Teradata implicitly creates a unique secondary index on any column set specified as PRIMARY KEY if it is not also defined to be the primary index for the table.
CHECK *(boolean condition)*	a Boolean conditional expression, including scalar comparison predicates using, for example, any of the following: • = • <> • > • >= Also see the chapter "Comparison Operators" in *Teradata RDBMS SQL Reference*, Volume 5. When CHECK is part of the column definition, then a search condition cannot reference any columns other than the one being defined, nor are aggregate functions allowed. Also see "BETWEEN … AND … Constraints" on page 1-238.
REFERENCES *table_name column_name* list	that each reference column in the REFERENCES clause identifies a *column_name* of the referenced table *table_name*. *table_name* must be a base table; it can*not* be a view. Referential integrity constraints are not supported for global temporary or volatile tables. Also see "REFERENCES" on page 1-226.
WITH CHECK OPTION	that referential integrity is checked only when the entire batch of actions of which it is a component completes. If any of the rows in the batch are found to violate the Referential Integrity rule, then the entire batch is rolled back. This clause is a Teradata extension to the ANSI SQL-99 standard.
WITH NO CHECK OPTION	that referential integrity is not to be enforced for the specified primary key-foreign key relationship. See "Referential Constraints" on page 1-230 for further information about this option.

Syntax Element …	Specifies …
UNIQUE Definition Clause	
CONSTRAINT *name*	the name for a named constraint. UNIQUE constraints are not permitted for volatile tables.
UNIQUE	that no two rows in the table can have the same field values. A unique primary or secondary index is used to enforce this constraint.
PRIMARY KEY	the column set that ensures that no two rows in the table can be duplicates. Only one primary key can be specified per table. A unique primary or secondary index is used to enforce this constraint.
column_name	the column set to be used as the primary key or as unique. If more than one *column_name* is specified, the PRIMARY KEY or UNIQUE column set is based on the combined values of the column named. A maximum of 16 columns can be specified for a primary key. Columns listed in a UNIQUE or PRIMARY KEY table constraint must all be defined as NOT NULL.
REFERENCES Definition Clause	
CONSTRAINT *name*	the name for a named constraint. REFERENCES constraints are not permitted for global temporary or volatile tables.
FOREIGN KEY	a foreign key for the table. Foreign keys and their use is described in "FOREIGN KEY" on page 1-226. FOREIGN KEY is a table-level constraint. A maximum of 64 foreign keys can be defined for a single table.
column_name	a name for the defined column.
REFERENCES	an integrity reference to the parent table named in *name*. Referential integrity constraints are not supported for global temporary and volatile tables. See "REFERENCES" on page 1-226.
table_name	the name of the referenced parent table used in the referential integrity constraint definition. If not in the same database as the child table, it must be qualified.
column_name	a column set that comprises the Parent table PRIMARY KEY or UNIQUE candidate key columns. Optional column name in *table_name*, used in the referential integrity constraint definition.

Syntax Element …	Specifies …
CHECK Definition Clause	
CONSTRAINT *name*	the name for a named constraint. Constraints are not permitted for volatile tables.
CHECK (*boolean_condition*)	a conditional expression, including scalar comparison predicates using, for example, any of the following comparison operators. • = • <> • > • >= For more information about these operators, see the chapter "Comparison Operators" in *Teradata RDBMS SQL Reference, Volume 5.* When CHECK is part of the column definition, then its defined search condition cannot reference any other columns in its table, nor are set specifications allowed. Valid column-level CHECK constraints that reference candidate keys of other tables *are* permitted. See also "BETWEEN … AND … Constraints" on page 1-238 and the chapter "Designing for Database Integrity" in *Teradata RDBMS Database Design*. A table-level CHECK constraint can compare any columns defined for its table, both against each other and against constant values.
AS Clause (Copy Table Syntax)	
source_table_name	the name of the source table whose definitions and, optionally, data are to be copied to *table_name*.
query_expression	an optional clause used to select a subset of the column definitions and, only when a WITH NO DATA clause is specified, data from *existing_table_name* for copying to *table_name*.
WITH NO DATA	that all (or a user-defined subset) of the definitions for the source table are to be copied to a new table. Indicate column subsets using a subquery; otherwise all column definitions are copied to the target table.
WITH DATA	that all (or a user-defined subset) of the definitions and the data for the source table are to be copied to a new table. You cannot create global temporary tables using the WITH DATA option. Indicate column subsets using a subquery; otherwise all column definitions and data are copied to the target table.

Syntax Element …	Specifies …
INDEX Definition Clause	
UNIQUE	that the named column must be unique. Any secondary indexes and the primary index can be defined to be unique.
PRIMARY	the primary index. The primary index is used by the hashing algorithm to partition table rows across the AMPs. If no primary index is specified, the first column in the table is selected to be the primary index. This index is nonunique (NUPI) by default except for the case of a single column table defined with the SET (no duplicate rows permitted) option.
INDEX	a keyword used to define the primary index and secondary indexes for the table. The INDEX list is an extension to ANSI SQL. Unlike the indexes created by the UNIQUE and PRIMARY KEY constraint definitions, indexes defined by the index list can have nullable fields.
name	an optional name for the index.
ALL	that a NUSI should retain row ID pointers for each logical row of a join index (as opposed to only the compressed physical rows). ALL also ignores the NOT CASESPECIFIC attribute of data types so a NUSI can include case-specific values. ALL enables a NUSI to cover a join index, enhancing performance by eliminating the need to access a join index when all values needed by a query are in the secondary index. However, ALL might also require the use of additional index storage space. Use this keyword when a NUSI is being defined for a join index and you want to make it eligible for the Optimizer to select when covering reduces access plan cost. ALL can also be used for an index on a table, however. You cannot specify multiple indexes that differ only by the presence or absence of the ALL option.
primary_index_column	the column set that defines a partitioned primary index.
PARTITION BY	that the primary index is partitioned by the partitioning expression that follows.
partitioning_expression	an SQL expression used to define the partition to which a partitioned primary index row is assigned when it is hashed to its AMP. See "CASE_N" and "RANGE_N" in *Teradata RDBMS SQL Reference, Volume 5* for documentation of functions that are particularly useful for use as partitioning expressions.
INDEX	a keyword to introduce a secondary index definition using an ORDER BY clause to order the index on a single column. The INDEX list is an extension to ANSI SQL. Unlike the indexes created by the UNIQUE and PRIMARY KEY constraint definitions, indexes defined by the index list can include nullable fields.
name	an optional name for the index.

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>ALL</td><td>that a NUSI should retain row ID pointers for each logical row of a join index (as opposed to only the compressed physical rows).

ALL also ignores the NOT CASESPECIFIC attribute of data types so a NUSI can include case-specific values.

ALL enables a NUSI to cover a join index, enhancing performance by eliminating the need to access a join index when all values needed by a query are in the secondary index. However, ALL might also require the use of additional index storage space.

Use this keyword when a NUSI is being defined for a join index and you want to make it eligible for the Optimizer to select when covering reduces access plan cost. ALL can also be used for an index on a table, however.

You cannot specify multiple indexes that differ only by the presence or absence of the ALL option.</td></tr>
<tr><td>column_name</td><td>one or more columns whose values are to be used as the basis for a primary or secondary index.

If more than one name is specified, the index is based on the combined values of each column named. A maximum of 64 columns can be specified for an index, and a maximum of 32 secondary indexes can be created for one table.</td></tr>
<tr><td>ORDER BY</td><td>row ordering on each AMP by a single NUSI column: either value-ordered or hash-ordered.

Rules for using an ORDER BY clause are shown in the following table.
<table>
<tr><th>IF you specify ORDER BY …</th><th>THEN …</th></tr>
<tr><td>with VALUES</td><td>the ORDER BY column_name must be numeric with four bytes or less</td></tr>
<tr><td>without HASH or VALUES</td><td>VALUES is assumed.</td></tr>
<tr><td>with a column_name</td><td>the ORDER BY column_name must be one of the columns in the INDEX list of columns.</td></tr>
<tr><td>without a column_name</td><td>the ORDER BY column_name is assumed to be the first column in the INDEX list of columns.</td></tr>
</table></td></tr>
<tr><td>VALUES</td><td>value-ordering for the ORDER BY column.

Select VALUES to optimize queries that return a contiguous range of values, especially for a covering index or a nested join.</td></tr>
<tr><td>HASH</td><td>hash-ordering for the ORDER BY column.

Select HASH to limit hash-ordering to one column, rather than all columns (the default).

Hash-ordering a multicolumn NUSI on one of its columns allows the NUSI to participate in a nested join where join conditions involve only that ordering column.</td></tr>
</table>

Syntax Element …	Specifies …
column_name	a column in the INDEX list of columns that specifies the sort order to be used. Supported data types for a value-ordered, four-bytes-or-less *column_name* are: • DATE • BYTEINT • INTEGER • DECIMAL • SMALLINT
Temporary/Volatile Table Preservation Clause	
ON COMMIT DELETE/ PRESERVE ROWS	to delete or preserve the contents of a global temporary or volatile table when a transaction completes. DELETE is the default. This parameter is valid for global temporary and volatile tables only.

ANSI Compliance

CREATE TABLE is ANSI SQL-99-compliant with extensions.

Global temporary tables are ANSI SQL-99-compliant.

Volatile tables are a Teradata extension to the ANSI SQL-99 standard.

Authorization

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

No privileges are granted on newly created volatile tables.

For other table types, the creator receives all the following privileges on the newly created table:

- CREATE TRIGGER (WITH GRANT OPTION)
- DELETE
- DROP TABLE
- DUMP
- INDEX
- INSERT
- REFERENCE
- RESTORE
- SELECT
- UPDATE

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

Table Definition Storage

The table structure definition (for all table types except volatile tables) is stored in the data dictionary.

Column Data Types and User Default Character Types

Column character type definitions assume the type assigned as the default for the user with CREATE USER and MODIFY USER.

You can explicitly override the default character type definitions by using the CHARACTER SET clause (see "CHARACTER SET Clause" in the chapter "Character Data Types," *Teradata RDBMS SQL Reference, Volume 3).*

For example, if the DEFAULT CHARACTER SET defined by CREATE USER is UNICODE for user Hon, then whenever Hon creates or alters a table, the declaration CHARACTER(n) in CREATE/ALTER TABLE is equivalent to CHARACTER(n) CHARACTER SET UNICODE. The same is true, the necessary changes being made, for the other column data types.

If no default character type is defined for a user, then the default character set used for a new table is UNICODE with the exceptions noted in the following table.

FOR this kind of table …	The default character set when no default is defined for the user creating the table is …
system tables	LATIN for non-Japanese systems.
	KANJI1 for Japanese systems.
user tables created prior to release 3.0.0	LATIN for non-Japanese systems.
	KANJI1 for Japanese systems.

Note that these character type definitions apply only to how the data is stored on the Teradata server; they do not affect the client, or session, character set.

Lock Placed When a Table Is Created

When you submit a CREATE TABLE statement, an EXCLUSIVE lock is placed on the table being created.

CREATE TABLE (Table Kind Clause)

Introduction

The Table Kind clause establishes two characteristics of the table:

- Are duplicate rows permitted?
- Is the defined object a standard base table, a global temporary table, or a volatile table?

SET and MULTISET

SET tables obey set theory semantics and the relational data model by not permitting duplicate rows in a table.

MULTISET tables obey bag, or multiset, semantics. They violate set theory semantics and the relational data model by permitting duplicate rows in a table.

THIS keyword ...	Means that duplicate rows are ...
SET	not permitted for the table.
MULTISET	permitted for the table.

If neither SET nor MULTISET is specified explicitly, then assignment is made according to the rules stated in the following table.

IF the session creating the table is in this mode ...	THEN this keyword is implied ...
ANSI	MULTISET
Teradata	SET

Some client utilities have restrictions with respect to MULTISET tables. For specific information on these restrictions, see the appropriate client utility manual.

- *Teradata FastLoad Reference*
- *Teradata Archive/Recovery Utility Reference*
- *Teradata Parallel Data Pump Reference*

GLOBAL TEMPORARY

Global temporary tables have a persistent definition but do not have persistent contents across sessions.

Space usage is charged to the login user temporary space.

A single session can materialize up to 2 000 global temporary tables at one time.

You materialize a temporary table locally by referencing it in a data manipulation statement. To materialize a temporary table, you must have the appropriate privilege on the base temporary table or on the containing database as required by the statement that materializes the table.

Any number of different sessions can materialize the same table definition, but the contents are different depending on the DML statements made against the individual materialized tables during the course of a session.

You can log temporary table updates. To do this, specify LOG immediately after the table name in the CREATE TABLE statement. LOG is the default.

You can save the contents of a temporary table across transactions. To do this, specify ON COMMIT PRESERVE as the last keywords in the CREATE TABLE statement. The default is *not* to preserve table contents after a transaction completes (DELETE).

The following options are not permitted for global temporary tables:

- Referential integrity constraints
- Permanent journaling

Access checking is not done on the materialized instances of any global temporary tables because those tables exist only for the duration of the session in which they are materialized.

VOLATILE

Volatile tables do not have a persistent definition; they must be created with a CREATE TABLE statement each time you need to use them. The table definition is cached only for the duration of the session in which it is created.

Because volatile tables are private to the session that creates them, their creation, access, modification, and drop privileges are not checked.

Both the contents and the definition of a volatile table are destroyed when a system reset occurs.

Space usage is charged to the login user spool space.

A single session can materialize up to 1 000 volatile tables at one time.

The following options are not permitted for volatile tables:

- Referential integrity constraints
- CHECK constraints
- Permanent journaling
- Compressed column values
- DEFAULT clause
- TITLE clause
- Named indexes

Otherwise, the options for volatile tables are identical to those for global temporary tables.

Volatile Tables and the Default Database

When you create an unqualified volatile temporary table, the login user space is used as the default database for the table, whatever the default database is currently specified to be.

When you create a qualified volatile table, the database name you specify must be identical to that of the login user database. Otherwise an error is returned.

Volatile and Global Temporary Tables, Teradata Transaction Mode, and DML Performance

When a session uses Teradata mode transaction semantics, you need to be aware of certain critical performance issues with the LOG/NO LOG and DELETE/PRESERVE ON COMMIT options for global temporary and volatile tables. The problem relates to whether the DML statements DELETE, INSERT, and UPDATE are grouped within explicit transaction boundaries or not. Recall that in Teradata transaction mode, each SQL statement is treated as a transaction unless you explicitly define transaction boundaries with the BEGIN TRANSACTION and END TRANSACTION statements.

Suppose you define the following four volatile tables so you can examine 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_nld
 NO 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_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.

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.[7]

For example, consider the following data for deletes, inserts, and updates to the previously defined volatile tables:

Table Name	Journal Step Performed?[a]	Deletion Step Performed?[b]	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

a. If yes, performed once *per table* per transaction.
b. If yes, performed once per transaction.

7 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;
```

If inserts are done to these tables using a multistatement request in Teradata mode like the following, the overhead is only one journal step and one delete step:

```
INSERT vt1 (1,1)
;INSERT vt2 (1,1)
;INSERT vt1 (2,2)
;INSERT vt3 (1,1)
;INSERT vt2 (2,2);
```

If inserts are done to these tables using a single statement request in an explicit transaction like the following, the overhead is one journal step for each table and one delete step for all of them.

```
BT;
 INSERT vt1 (1,1); ← one journal step for vt1
 INSERT vt2 (1,1); ← one journal step for vt2
 INSERT vt1 (2,2);
 INSERT vt3 (1,1); ← one journal step for vt3
 INSERT vt2 (2,2);
ET;                ← one delete step for all NO LOG or ON COMMIT DELETE ROWS
                     volatile tables referenced in the transaction.
```

Because the implicit transaction semantics of Teradata mode treat each SQL statement as an individual transaction, a large number of all-AMPs operations occurs with a significantly negative impact on overall CPU and Disk I/O performance unless you explicitly group DML statements into explicit transactions.

For example, consider the following comparison test of 200 individual Teradata session mode inserts made to:

- A permanent table.
- A volatile table with ON COMMIT DELETE and 200 implicit transactions.
- A volatile table with ON COMMIT DELETE and a single explicit transaction defined to encapsulate all 200 individual inserts.

Table Type	Transaction Boundary Semantics	Sum(CPUTime)	Sum(DiskI/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, but the ratios between values are comparable across configurations and hold for both 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 both CPU time and disk I/O measures where other factors are held constant.

Resolving Tables References

When encountering an unqualified table name, Teradata looks for a table 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 "ambiguous table name" error message results if the table name exists in more than one 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 login user database and not a table in another database, the reference defaults to the volatile table.

  Thus,

  ```
  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 login user database, not the current default database, DB1.
- If the unqualified table name is a volatile table in the login user database and also a table in another database, the reference is ambiguous and an "ambiguous table name" error results.

  Thus,

  ```
  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 volatile_table_1 (1,1)
  ```

  The INSERT results in an error message because the table, `volatile_table_1`, already exists in the login user database, U1.

- If the unqualified table name is both a volatile table in the login user database and also a permanent table in the login user database (the same database), the reference defaults to the volatile table.

  This could 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.

CREATE TABLE (Table Options Clause)

Introduction

Table options are an extension to ANSI SQL.

Specifying Data Protection

Specific protection for the data in a table is provided by the FALLBACK and JOURNAL options.

FALLBACK causes a duplicate copy (the fallback copy) to be maintained by Teradata. 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 data that changed as a result of any insert, update, or delete.

These journals are used either to restore a table or to reverse the changes that were made to a table.

Assigning a Journal to the Table

To assign a journal to a new data table, the named journal table must exist in the database being used. If it does not, a journal table can be added or created by using MODIFY DATABASE or MODIFY USER. Then CREATE TABLE can be resubmitted, or ALTER TABLE can be used to assign the new journal table to the existing data table.

Permanent journaling is not supported for global temporary or volatile tables.

Providing for Fallback Data

If you have specified FALLBACK (either explicitly or by default) in creating the table, Teradata automatically maintains a duplicate copy of the data in the table. This fallback copy is used if the primary copy becomes unavailable.

To create a fallback copy of a new table, include the FALLBACK clause 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 new table. Alternatively, if you want to accept the default, do not specify the option for the table.

Unless specified as part of the CREATE TABLE statement, protection for the data in a new table defaults to that specified for the database in which the table is created.

Providing for a Permanent Journal

A permanent journal table is created when the default JOURNAL table option and a journal table name are specified as part of the CREATE/MODIFY DATABASE/USER statement. The named table resides in the database or user space that defined it. In addition, 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 journaling defaults by specifying the JOURNAL options as part of the CREATE TABLE statement. Thus you can specify the journal table, the type of image (before-change or after-change, or both) and whether dual images are to be maintained for the created table, no matter 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, including revalidation
- 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.

Permanent journals are not supported for global temporary or volatile tables.

LOG and NO LOG

Global temporary and volatile tables permit you to define whether their activity is logged to the transaction 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.

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. Rows are not split across data blocks.

If DATABLOCKSIZE is not specified, then the default size is used.

The default DATABLOCKSIZE settings depend on the cylinder size set for your system. Cylinder size is a system-level value shown as SectsPerCyl with the Filer utility or Number of Disk Blocks Per Cylinder with the pdeconfig utility.

The following table shows the smallest, largest, and default DATABLOCKSIZE values correlated to the cylinder size settings.

Cylinder Size (Sectors)	DATABLOCKSIZE (Bytes)		
	Minimum	Default	Maximum
1488 (pre-Release 3 default)	6 144	32 256	130 560
3872	7 168	130 560	130 560

A sector is 512 bytes. If the computed value for the DATABLOCKSIZE attribute does not fall within the allowable range, an error message is returned.

IF this keyword is specified …	THEN …
BYTE[S]	the value of the data block size is the result of rounding *n* to the nearest multiple of the sector size in bytes. Rounding is upward when *n* is exactly halfway between two consecutive multiples.
KILOBYTE[S] KBYTE[S]	the value of the data block size is the result of rounding 1024*n* to the nearest multiple of the sector size in bytes.
none	BYTES is assumed.

At any give time, the DATABLOCKSIZE value returned in response to a SHOW TABLE statement is the value specified in the CREATE TABLE or most recently entered ALTER TABLE statement.

For additional information, see the following chapters in other manuals in the Teradata documentation suite:

- "Database Considerations" in *Teradata RDBMS Database Design*
- "Disk Space" in *Teradata RDBMS Performance Optimization*

MINIMUM and MAXIMUM DATABLOCKSIZE

Because of the range of data block sizes that Teradata allows, the following equivalences are defined:

This keyword sequence ...	Is identical to this specification ...
MINIMUM DATABLOCKSIZE	DATABLOCKSIZE = 6144 BYTES or DATABLOCKSIZE = 7168 BYTES depending on the cylinder size set for your system: either 1488 or 3872 sectors, respectively.
MAXIMUM DATABLOCKSIZE	DATABLOCKSIZE = 130560 BYTES

FREESPACE

FREESPACE = *n* 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 this table. The PERCENT keyword is noise and does not affect the semantics of the specification.

The rules are as follows.

- The percent freespace attribute must be in the range 0 to 75 percent.
- If *n* is a decimal or floating point constant, an integer value is obtained by taking the result of truncating *n*.
- The freespace value returned in response to a SHOW TABLE statement is the value specified in the CREATE TABLE or most recently entered ALTER TABLE statement. Also see the descriptions of "ALTER TABLE" on page 1-12 and "SHOW HASH INDEX SHOW JOIN INDEX SHOW MACRO SHOW PROCEDURE SHOW TABLE SHOW TRIGGER SHOW VIEW" on page 4-79.

CREATE TABLE (Column Definition Clause)

Introduction

This clause defines the table column elements.

A name and a data type must be specified for each column defined for a table. Each data type can be further defined with one or more attribute definitions.

The following optional subclauses are also elements of the SQL column definition clause:

- Data Type declaration
- Data Type Attribute declaration
- Column Storage Attributes Clause
- Column Constraint Attributes Clause
- UNIQUE Definition Clause
- REFERENCES Definition Clause
- CHECK Constraint Definition Clause

Column Storage Attributes and Constraint Names

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 (see *Teradata RDBMS Utilities*). Explicit formatting applies both to the parsing and to the retrieval of character strings.

Constraint names are specified by the ANSI SQL-99 standard.

Table Size and Maximum Columns Per Table

If a CREATE TABLE statement defines more than 2 048 columns, the table creation attempt aborts and Teradata returns an error message.

Identity Columns

Identity columns are used mainly to ensure row uniqueness by taking a system-generated unique value. They are valuable for generating 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.

Note that 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" on page 1-221 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 restrictions:

- Support singleton INSERT statements only.
- Support the following bulk insert operations:
  - Single statement INSERT requests through multiple concurrent sessions.
    Example: parallel BTEQ imports into the same table.
  - Multistatement INSERT requests through multiple concurrent sessions.
    Example: Teradata TPump inserts.
  - INSERT … SELECT statements.
- Do *not* support the following bulk insert operations:
  - MultiLoad
  - Teradata Warehouse Builder
  - FastLoad
- If a triggered event is an INSERT into a table with an identity column, then the triggered action statement 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 value list.

    Example: `INSERT table (1,NULL);`
  - Implicit specification of NULL by the omission of a value in the value list when no column name list is provided.

    Example: `INSERT table (1, );`
  - Omission of a column name from the column name list.

    Example: `INSERT 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.
- Cannot be null.
- Cardinality of the 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), DECIMAL(15,0), NUMERIC(18,0), or NUMERIC(15,0), depending on the DBSControl MaxDecimal flag setting for your system.
- Inserts into GENERATED BY DEFAULT identity columns using TPump or a USING statement modifier cannot reuse TPump field variables or USING host 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);
```

Given this table definition, the following INSERT statements fail because they reuse USING host variables[8]:

```
USING (uc1 INTEGER, uc2 INTEGER)
INSERT INTO test01 VALUES (:uc1, :uc1);

USING (uc1 INTEGER, uc2 INTEGER)
INSERT INTO test01 VALUES (:uc2, (:uc2 + :uc2));
```

Parallel examples using TPump look like this:

```
.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 INSERT statements succeed because they do not reuse USING host or TPump field variables:

```
USING (uc1 INTEGER, uc2 INTEGER)
INSERT INTO test01 VALUES (:uc1, :uc2);

USING (uc1 INTEGER, uc2 INTEGER)
INSERT INTO test01 VALUES (:uc2, (:uc1 + :uc1));

.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));
```

8 The constant values in the client system data record must match, item for item, the USING or TPump description of those values. Also, the constants in the data record must appear in the order in which they are defined in the USING statement modifier or TPump .FIELD commands.

- Insert operations into GENERATED BY DEFAULT and GENERATED ALWAYS identity columns using TPump field variables or a USING statement modifier cannot specify USING host or TPump field variables if the value inserted into the identity column is derived from an expression that includes a USING host variable or TPump field variable.

  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);
  ```

  Given this table definition, the following INSERT statements fail because they attempt to insert values into an identity column that are derived from a USING expression that includes a host variable:

  ```
  USING (uc1 INTEGER, uc2 INTEGER)
  INSERT INTO test01 VALUES ((:uc1 + :uc2), :uc2);

  USING (uc1 INTEGER, uc2 INTEGER)
  INSERT INTO test01 VALUES ((:uc1 + 1), :uc2);
  ```

  The same examples generalize to TPump inserts that use field variables.

  The following INSERT succeeds because the expression from which the value inserted into the identity column is derived does not use a host variable:

  ```
  USING (uc1 INTEGER, uc2 INTEGER)
  INSERT INTO test01 VALUES ((1 + 2), :uc2);
  ```

- Inserts into both GENERATED BY DEFAULT and GENERATED ALWAYS identity columns using TPump field variables or a USING statement modifier cannot insert into the identity column of more than one identity column table.

  The following examples are based on these table definitions:

  ```
  CREATE MULTISET TABLE test01 (
    a1 INTEGER GENERATED BY DEFAULT AS IDENTITY
       (START WITH 1
        INCREMENT BY 20
        MAXVALUE 1000
       ),
    a2 INTEGER);
  ```

```
CREATE MULTISET TABLE test02 (
  a1 INTEGER,
  a2 INTEGER GENERATED BY DEFAULT AS IDENTITY
     (START WITH 1
      INCREMENT BY 20
      MAXVALUE 1000
     ));

CREATE MULTISET TABLE test03 (
  a1 INTEGER,
  a2 INTEGER);
```

Given these table definitions, the following multistatement INSERT operation fails because it attempts to insert into the identity column of more than one identity column table using values taken from the same USING statement modifier:

```
USING (uc1 INTEGER, uc2 INTEGER)
INSERT INTO test01 VALUES (:uc1, :uc2)
;INSERT INTO test03 VALUES (:uc1, :uc2)
;INSERT INTO test02 VALUES (:uc1, :uc2);
```

Columns test01.a1 and test02.a2 are both identity columns; therefore, the multistatement insert fails.

The same example generalizes to TPump inserts that use field variables.

The following multistatement INSERT operations succeed because they do not attempt to insert into the identity column of more than one identity column table using values taken from the same USING statement modifier:

```
USING (uc1 INTEGER, uc2 INTEGER)
INSERT INTO test01 VALUES (:uc1, :uc2)
;INSERT INTO test03 VALUES (:uc1, :uc2);

USING (uc1 INTEGER, uc2 INTEGER)
INSERT INTO test01 VALUES (:uc1, :uc2)
;INSERT INTO test03 VALUES (:uc1, :uc2)
;INSERT INTO test01 VALUES (:uc1, :uc2);
```

This multistatement insert succeeds because it only uses USING modifier values to insert into the identity column of the test01 identity column table.

The same example generalizes to TPump inserts that use field variables.

The GENERATED … AS IDENTITY keywords introduce a clause that specifies:

- The column is an identity column.
- Teradata generates values for the column when a new row is inserted into the table except in certain cases described later.

IF you specify GENERATED …	THEN Teradata …
ALWAYS	always generates a unique value for the column when a new row is inserted into the table and NO CYCLE is specified.[a]
BY DEFAULT	generates a unique value for the column when a new row is inserted into the table only if the INSERT statement does not specify a value for the column.

a. Note, however, that 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. This means that some preprocessing must still be performed on rows to be loaded into identity column tables if real world uniqueness is a concern.

IF you are using the identity column for this purpose …	THEN you should specify this option …
to ensure a UPI, USI, PK, or some other row uniqueness property	ALWAYS
• to load data into or unload data from a table • to copy rows from one table to another table • to fill in gaps in the sequence • to reuse numbers that once belonged to now-deleted rows	BY DEFAULT

Process for Generating Identity Column Numbers

Teradata allocates identity column numbers differently depending on whether an operation is 1) a singleton or USING clause-based insert or 2) an INSERT … SELECT insert.

FOR this type of insert operation …	Identity column numbers are cached on the …
• Singleton • USING clause[a]	PE.
INSERT … SELECT	AMP.

a. Examples are BTEQ imports and Teradata TPump inserts.

Teradata 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, Teradata 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 field in the DBSControl record.

Once a range of numbers has been reserved, Teradata stores the first number in the range in 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.

Rules for Specifying Identity Columns

The following rules apply to identity columns:

- 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.
- An identity column cannot be part of any of the following types of index:
  - Composite primary index
  - Composite secondary index
  - Join index
  - Hash index
  - Partitioned primary 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.
  The following are examples of valid data types:
  - BYTEINT
  - DECIMAL(n,0)
  - INTEGER
  - NUMERIC(n,0)
  - SMALLINT

  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
…	…	…
DECIMAL(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

  The following are examples of non-valid data types:
  - All non-numeric types, including DateTime and INTERVAL types
  - DECIMAL(n,m) where m is not 0
  - DOUBLE PRECISION
  - FLOAT
  - NUMERIC(n,m) where m is not 0
  - REAL
-

WHEN this type of table …	HAS an IDENTITY column specified to be …	THEN it …
any	GENERATED ALWAYS	cannot have duplicate rows.
MULTISET	GENERATED BY DEFAULT	can have duplicate rows.

- If you insert values into a GENERATED BY DEFAULT IDENTITY column in a SET table, the inserted row (but not the inserted value for the IDENTITY column) must be unique or Teradata 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, USI, PK, or UNIQUE NOT NULL.
-

A column specified to be …	Has this response to user-defined insert values …
GENERATED ALWAYS	Always rejects them.
GENERATED BY DEFAULT	Always accepts them.

  A column specified to be GENERATED BY DEFAULT AS IDENTITY only generates a unique column value for inserted rows that contain a null IDENTITY column.
- You cannot specify any of the following column attributes for an IDENTITY column:
  - DEFAULT
  - BETWEEN
  - COMPRESS
  - CHECK constraints
  - REFERENCES constraints

Identity Columns, Duplicate Column Values, and Duplicate Rows

The use of identity columns presents two 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.

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 without identity columns, but that are specified as SET tables or have a UPI do not permit the insertion of duplicate rows.
- TPump 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.

Caution 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 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 two employee rows that are identical except for their different employee number values. While this is an error from the perspective of the enterprise, the two 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

The only forms of bulk inserts supported for identity column tables are the following:

- Single statement INSERT statements made through multiple concurrent sessions, for example parallel BTEQ imports into the same table.
- Multistatement INSERT statements made through multiple concurrent sessions, for example TPump inserts.

Teradata Warehouse Builder, MultiLoad, and FastLoad are not supported for identity column tables.

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 TPump and BTEQ imports receive the same warning for each row inserted.

Rows bulk-inserted into a table defined with a GENERATED BY DEFAULT identity column are not guaranteed to be unique.

Optimizing Bulk Inserts Into an Identity Column

The performance of bulk inserts into a table with an identity column depends on the DBSControl setting for the IdColBatchSize variable (see *Teradata RDBMS Utilities, Volume 1* for information about DBSControl). 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 in 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 IdColBatchSize setting enhances bulk insert performance, it also means that more identity column numbers are lost if a system restart occurs.[9]

The optimal batch size for your system is a function of the number of AMPs. The IdColBatchSize 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.

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 two factors: the next available value stored in DBC.IdCol.AvailValue and the defined incremental identity column batch size for the system. This quantity is specified by the DBSControl flag IdColBatchSize (see the chapter "DBS Control Utility" in *Teradata RDBMS Utilities*, and "Optimizing Bulk Inserts Into an Identity Column" on page 1-223 for further information).

When the pool of available identity column numbers in the cache is exhausted, Teradata 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 on the table,[10] 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.

---

9 Reserved identity column numbers are memory-resident. If Teradata must be restarted, the batch insert transaction is rolled back, and those reserved identity column numbers are permanently lost.

10 The SELECT privilege on DBC.IdCol is restricted to user DBC and any user explicitly granted the privilege by DBC.

Compressing Columns

NULL and as many as 255 distinct values can be compressed per column. You can compress an unlimited number of columns per table.

IF this type of compression is used …	THEN …
single-valued	the default is null if no value is specified explicitly.
multivalued	• 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. • There is no essential ordering for specifying values to be compressed for a given column. • The same value cannot be specified more than once in the compression value list for a given column.

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

Compressed columns cannot be used for fast path optimization in an INSERT … SELECT operation.

Compression is not supported for any column that is a member of the primary index column set for a table.

Compression is not supported for volatile tables or identity columns.

See the chapter “Default Value Control Phrases” in *Teradata RDBMS SQL Reference, Volume 3* for more information about the COMPRESS phrase.

See the chapter "Database-Level Considerations" in *Teradata RDBMS Database Design* for detailed information about value compression.

Constraint Names

Constraint names must conform to the rules for a Teradata name (30 characters maximum) and be unique among all other constraint, primary index, and secondary index names specified in the table.

The characters used can be any of the following:

- Uppercase and lowercase alphabetic characters
- Integers
- Any of the following special characters:
  - -
  - #
  - $

Unnamed constraints are not assigned system-generated names.

PRIMARY KEY

The primary key of a table is a column set that uniquely identifies each row of that table (see "Identifying Primary Keys" in the chapter "The Normalization Process" in *Teradata RDBMS Database Design*). Primary keys, which are a logical construct in the relational model, are frequently implemented physically as the unique primary index for a table. When a table has a nonunique primary index, Teradata strongly recommends defining a USI on the primary key. Primary and other alternate keys are used with foreign keys to enforce referential integrity relationships between tables.

Surrogate Keys

Situations 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 entities 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 entities in a meaningful way: they are simply an arbitrary method to distinguish among them.

Surrogate keys are typically arbitrary system-generated sequential integers. See "Identity Columns" in the chapter "Database-Level Considerations" in *Teradata RDBMS Database Design* and "Generating Surrogate Keys" on page 1-225, for information about how to generate surrogate keys in Teradata.

Generating Surrogate Keys

You can generate surrogate keys in Teradata using the identity column feature (see "Identity Columns" on page 1-212). 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 or DECIMAL specified with the largest possible width for your site: either (15,0) or (18,0), depending on the value of the DBSControl field MaxDecimal for your site.

UNIQUE Constraint

UNIQUE constraints are used to ensure that the uniqueness of alternate keys is enforced by Teradata. The physical implementation of UNIQUE constraints is identical to that of UPI, USI, and any other uniquely constrained column sets. Columns with UNIQUE constraints often have referential integrity relationships with other tables.

PRIMARY KEY Versus UNIQUE PRIMARY INDEX

UNIQUE and PRIMARY KEY can be defined as constraints on a column. The internal Teradata implementation of both constraints is as UNIQUE NOT NULL indexes.

Teradata also supports the related constraints UNIQUE INDEX and UNIQUE PRIMARY INDEX. Nonunique primary and secondary indexes (NUPIs and NUSIs), join indexes, and hash indexes are also supported.

The ANSI SQL-99 standard does not describe or define DDL for indexes because it regards them as an implementation issue.

FOREIGN KEY

If a FOREIGN KEY is specified in a table-constraint, the FOREIGN KEY *column_name_list* must see columns defined within the CREATE TABLE statement for the target table, and the same column name should not be included in the foreign key definition more than once.

The *column_name_list* should be identical to a list of columns in the referenced table which are defined either as the primary key columns or as a set of columns with the unique attribute. *Table-name* refers to the referenced table which must be a base table, and not a view.

The following rules apply to foreign keys:

- A maximum of 64 foreign keys can be defined on or about a table.

  This limit includes both references to and from the table. For example, if 10 tables (t_1, t_2, … t_{10}) reference table t_n, then only 54 (64 - 10 = 54) foreign keys can be defined on table t_n.

  Column-level CHECK constraints that reference alternate keys in other tables do not count against this limit.
- Each individual foreign key can be defined on a maximum of 16 columns.
- A maximum of 100 table-level constraints can be defined for any table.

REFERENCES

Each column in the REFERENCES clause should identify a column of the referenced table, and the same column name should not be identified more than once.

The following rules apply to both column- and table-level REFERENCES constraints.

- Each column in the REFERENCES clause should correspond to a column of the referenced table, and the same column name should not be specified more than once.
- The referencing column list should contain the same number of column names as the referenced column list. The ith column of the referencing list corresponds to the ith column identified in the referenced list. The data type of each referencing column should be the same as the data type of the corresponding referenced column.
- The user issuing the CREATE TABLE statement containing a REFERENCES clause must either have the REFERENCE privilege on the referenced table or on all specified columns of the referenced table.
- If REFERENCES is specified in a *column_constraint*, then *table_name* defines the referenced table. *table_name* must be a base table, not a view.
- You cannot specify REFERENCES constraints for global temporary or volatile tables.
- You cannot specify REFERENCES constraints for identity columns.
- A REFERENCES clause can apply to three different forms of referential constraint[11] as described in the following table:

Referential Constraint Type	DDL Definition Clause	Does It Enforce Referential Integrity?	Level of Referential Integrity Enforcement
Referential integrity constraint	REFERENCES	Yes	Row
Batch referential integrity constraint	REFERENCES WITH CHECK OPTION	Yes	Implicit transaction
Referential constraint	REFERENCES WITH NO CHECK OPTION	No	None

11 When printed in lowercase characters, the phrase referential constraint refers to the general class of referential constraints including standard referential integrity, batch referential integrity, and Referential Constraints.
When printed with initial capital characters, the phrase Referential Constraint refers specifically to a constraint that defines a referential relationship but instructs Teradata not to enforce it. See "Referential Constraints" on page 1-230 for more information.

The various types have different applications as described in the following table:

Referential Constraint Type	Application
Referential integrity constraint (see "Standard Referential Integrity" on page 1-228)	• Tests each individual inserted, deleted, or updated row for referential integrity. • 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 referential integrity constraint (see "Batch Referential Integrity" on page 1-229)	• Tests an entire insert, delete, or update batch operation[a] for referential integrity. • If insertion, deletion, or update of any row in the batch violates referential integrity, then parsing engine software rolls back the entire batch and returns an abort message. • Permits special optimization of certain queries.
Referential constraint (see "Referential Constraints" on page 1-230)	• Does not test for referential integrity. • Assumes that the user somehow enforces referential integrity in a way other than the normal declarative referential integrity constraint mechanism. • Permits special optimization of certain queries.

a. In this context, a batch operation is defined as an implicit SQL transaction.

The following rules apply to column-level REFERENCES constraints only.

- If *column_name* is omitted, then the referenced table must have a single column primary key, and the specified foreign key column references that primary key column of the referenced table.
- If *column_name* is specified, then it must refer to the single column primary key of the referenced table or to a single column alternate key in the referenced table defined as UNIQUE.

The following rule applies to table-level REFERENCES constraints only.

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

Standard Referential Integrity

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 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 Warehouse Builder on 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.

Standard referential integrity is the only form of referential constraint that is specified both by the relational model (see "The Referential Integrity Rule" in the chapter "The Normalization Process" in *Teradata RDBMS Database Design*) and by the ANSI SQL-99 standard.

Batch Referential Integrity

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 (the entire implicit transaction must complete successfully) 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 Teradata 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 "REFERENCE Indexes" in the chapter "Teradata Indexes: The Basics" in *Teradata RDBMS Database Design* 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 implicit transaction or SQL statement.

This form of RI is tested by joining the relevant parent and child table rows. If there is an inconsistency in the result, then Teradata rolls back the entire implicit transaction.

In some situations, there can be a tradeoff for this enhanced performance because when an RI violation is detected, the entire implicit transaction rolls back instead of one integrity-breaching row.

In many instances, this is no different than the case for standard RI because for most implicit transactions, only one row is involved.

The difference with batch RI is when multiple update operations are involved: for example, an INSERT … SELECT of 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 trigger RI violations.

You cannot use bulk data loading utilities like FastLoad, MultiLoad, or Teradata Warehouse Builder on tables defined with batch referential integrity.

Batch referential integrity is a Teradata extension to the ANSI SQL-99 standard.

Referential Constraints

In some circumstances, the Teradata query 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 permits you to take advantage of these optimizations without incurring the overhead of enforcing the suggested referential constraints.

Caution The Referential Constraint feature presumes a high level of trust between the Teradata database and its users, assuming that users never violate any of the defined Referential Constraints even though Teradata does not enforce those constraints. As a result, 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 Tables In an Referential Constraint Relationship" on page 1-234). 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. See "Scenario for Data Corruption With Referential Constraints" on page 1-236 for an example.

Referential Constraint relationships are defined by specifying the WITH NO CHECK OPTION phrase for a REFERENCES constraint. When you specify this phrase, the database does not enforce the defined RI constraint. This means that a row in a child table having a non-null value for a referencing column can exist in a table even when no equivalent parent table value exists. 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 Warehouse Builder are permitted against columns defined with SRI constraints without warning.

Referential Constraints are a Teradata extension to the ANSI SQL-99 standard.

Example of How the Optimizer Takes Advantage of Referential Constraints

To understand how the Optimizer can produce more performant 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;

 *** Help information returned. 33 rows.
 *** Total elapsed time was 1 second.
```

```
Explanation
-----------------------------------------------------------------------------
  1) First, we lock a distinct TESTCOST."pseudo table" for read on a
     RowHash to prevent global deadlock for TESTCOST.tb3.
  2) Next, we lock a distinct TESTCOST."pseudo table" for read on a
     RowHash to prevent global deadlock for TESTCOST.tb1.
  3) We lock TESTCOST.tb3 for read, and we lock TESTCOST.tb1 for read.
  4) We do an all-AMPs RETRIEVE step from TESTCOST.tb3 by way of an
     all-rows scan with a condition of ("NOT (TESTCOST.tb3.b IS NULL)")
     into Spool 4 (all_amps), which is redistributed by hash code to
     all AMPs.  Then we do a SORT to order SORT to order Spool 4 by row
     hash.  The size of Spool 4 is estimated with no confidence to be 3
     rows.  The estimated time for this step is 0.03 seconds.
  5) We do an all-AMPs JOIN step from TESTCOST.tb1 by way of a RowHash
     match scan with no residual conditions, which is joined to Spool 4
     (Last Use).  TESTCOST.tb1 and Spool 4 are joined using a merge
     join, with a join condition of ("TESTCOST.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 no confidence to be 3 rows.
     The estimated time for this step is 0.20 seconds.
  6) We do a SUM step to aggregate from Spool 3 (Last Use) by way of an
     all-rows scan, and the grouping identifier in field 1.  Aggregate
     Intermediate Results are computed locally, then placed in Spool 5.
     The size of Spool 5 is estimated with low confidence to be 3 rows.
     The estimated time for this step is 0.17 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 SORT to order Spool 1 by
     the sort key in spool field1.  The size of Spool 1 is estimated
     with low confidence to be 3 rows.  The estimated time for this
     step is 0.17 seconds.
  8) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

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;

 *** Help information returned. 19 rows.
 *** Total elapsed time was 1 second.
```

```
Explanation
-----------------------------------------------------------------------------
  1) First, we lock a distinct TESTCOST."pseudo table" for read on a
     RowHash to prevent global deadlock for TESTCOST.tb2.
  2) Next, we lock TESTCOST.tb2 for read.
  3) We do a SUM step to aggregate from TESTCOST.tb2 by way of an
     all-rows scan with a condition of ("NOT (TESTCOST.tb2.b IS NULL)"),
     and the grouping identifier in field 1.  Aggregate Intermediate
     Results are computed locally, then placed in Spool 3.  The size of
     Spool 3 is estimated with no confidence to be 3 rows.  The
     estimated time for this step is 0.15 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 SORT to order Spool 1 by
     the sort key in spool field1.  The size of Spool 1 is estimated
     with no confidence to be 3 rows.  The estimated time for this step
     is 0.17 seconds.
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.
```

Rules for Using Referential Constraints

Other than not enforcing referential integrity, most of the rules for Referential Constraints are identical to those documented by "FOREIGN KEY" on page 1-226 and "REFERENCES" on page 1-226.

The exceptions are documented by the following set of rules that apply specifically to the specification and use of Referential Constraints:

- You can specify standard RI, batch RI, and Referential Constraints in the same table, but not for the same column.
- 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.
- 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.

Validating the Integrity of Tables In an Referential Constraint Relationship

If you decide to specify a Referential Constraint for a table, the responsibility for ensuring or validating the referential integrity of that table is yours alone. Teradata 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 ensurance, 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 might 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 the chapter "Designing for Database Integrity" in *Teradata RDBMS Database Design*. There is the additional likelihood that actively firing triggers will 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 particular 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:

```
SELECT DISTINCT childtable.*
FROM childtable, parenttable
WHERE childtable.fk NOT IN
  (SELECT pk FROM parenttable)
AND childtable.fk IS NOT NULL;
```

The exclusion of foreign key nulls is included in the query because it is not possible to determine what values they represent.

For this scenario, the following set of correspondences is defined:

Generic Query Element	Specific Query Element
childtable	softri_tbl1
parentable	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 nonmatching rows must be deleted from the child table in order to maintain the referential integrity of the database.

You should perform this validation query regularly, particularly after any substantial updates are made to the tables in the implied relationship.

Scenario for Data Corruption With Referential Constraints

Always exercise caution when manipulating data within a Referential Constraints environment. If you define referential relationships using Referential Constraints, then INSERT, DELETE, and UPDATE statements can corrupt data in the parent tables of those relationships if the DML statement 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 two 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 the following rows into table_1 and table_2:

```
INSERT INTO table_1 (1,2);
INSERT INTO table_1 (2,2);
INSERT INTO table_1 (3,2);

INSERT INTO table_2 (5,6);
INSERT INTO table_2 (6,7);
INSERT INTO table_2 (7,8);
```

The populated tables now look like this:

table_1

PK	
col_1	**col_2**
1	2
2	2
3	2

table_2

	FK
col_1	**col_2**
5	6
6	7
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 Teradata 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 predicate filter, and then incorrectly updates all rows in table table_2 with a col_1 value of 1, corrupting the database.

CHECK Constraints

Teradata supports the CHECK constraint both at the column table levels. Constraints that reference one or more fields in a row are also permitted.

The following rules apply to CHECK constraints.

- CHECK constraints can be defined 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 specify CHECK constraints for volatile tables.
- You cannot specify CHECK constraints for identity columns.
- 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.

- Unnamed CHECK constraints with identical wording and case are considered to be duplicates, and Teradata returns an error when you specify them.

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

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

  The following CREATE TABLE statement is not valid and returns an error.

  ```
  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.

The following rules apply to column-level CHECK constraints:

- You can specify multiple constraints on a single column.

  If more than one *unnamed* CHECK constraint is defined for a column, then Teradata combines them into one column constraint.

  However, each *named* column-level CHECK constraint is handled 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 to table-level CHECK constraints:

- A table-level constraint must reference at least two columns from its table.
- Table-level CHECK constraint predicates cannot reference columns from other tables.
- A maximum of 100 table-level constraints can be defined for any table.

BETWEEN … AND … Constraints

The following non-ANSI Teradata construct is supported (except for volatile tables and identity columns).

```
BETWEEN number AND number
```

Teradata treats it as if it were the following ANSI-compliant statement.

```
CHECK (field_name BETWEEN number AND number)
```

Newly written applications should use the ANSI-compliant CHECK constraint syntax.

CREATE TABLE (Index Definition Clause)

Introduction

This clause creates a primary index and optional secondary indexes for the table.

Every table must have one and only one primary index, which can optionally be partitioned.

You should *always* define the primary index for your tables explicitly.

If you do not explicitly specify a column set to define the primary index, but do define a primary key or a unique column constraint for the table, then the first such constraint defined is assigned to be the primary index by default.

If you do not define a primary index, primary key, or any unique constraints, then the first column defined for the table is selected to be its *nonunique* primary index by default.[12]

Because Teradata assigns rows to AMPs based on their primary index, it is important to select a column that distributes table rows evenly when a nonunique primary index is defined for a table. This is critical whether the primary index is partitioned or not. When you allow a column to be selected as the primary index for a table by default, you have no control over the smoothness of its distribution across the AMPs and in all likelihood will have a table that produces significant 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" on page 1-100).

To create a join index incorporating this table, use the CREATE JOIN INDEX statement (see "CREATE JOIN INDEX" on page 1-108).

To create a hash index on a table, use the CREATE HASH INDEX statement (see "CREATE HASH INDEX" on page 1-86).

12 The exception being a single column SET table, for which the primary index is unique by default.

Index Types

Index determination and definition is an important aspect of database design.

Teradata supports eight index types:

- Unique nonpartitioned primary
- Unique partitioned primary
- Nonunique nonpartitioned primary
- Nonunique partitioned primary
- Unique secondary
- Nonunique secondary
- Join
- Hash

For additional information about Teradata indexes, see the chapters "Teradata Indexes: The Basics," "Primary Indexes," "Secondary Indexes," and "Join and Hash Indexes" in *Teradata RDBMS Database Design* and the following topics in the chapter “SQL Data Definition, Control, and Manipulation” in *Teradata RDBMS SQL Reference, Volume 1*:

- Keys Versus Indexes
- Primary Indexes
- Secondary Indexes
- Join and Hash Indexes

These topics are described in greater detail in *Teradata RDBMS Database Design*.

Primary Indexes

The primary index is hashed to determine on which AMPS of the Teradata server rows are stored. 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 columns that are members of the primary index column set.

Partitioned and Nonpartitioned Primary Indexes

Primary indexes can be partitioned or nonpartitioned.

A nonpartitioned primary index (NPPI) 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. Partition assignment is based on how *partitioning_expression* is defined. The functions CASE_N and RANGE_N (see *Teradata RDBMS SQL Reference, Volume 5*) are designed to support simple partitioning expressions, but you can write any valid SQL expression as a partitioning expression with the following exclusions:

- Expressions that require comparisons of CHARACTER or GRAPHIC data to be evaluated. Such comparisons use the following operators in their evaluation:
  - =
  - >
  - <
  - >=
  - <=
  - BETWEEN
  - LIKE
- CHARACTER and GRAPHIC comparison functions such as the following:
  - CHAR2HEXINT
  - INDEX
  - LOWER
  - MINDEX
  - POSITION
  - TRANSLATE
  - TRANSLATE_CHK
  - TRIM
  - UPPER
  - VARGRAPHIC
- Aggregate functions
- Ordered analytical functions
- Built-in functions
- The RANDOM function
- The HASHAMP and HASHBAKAMP functions[13]
- The ROWID and PARTITION system-derived columns
- The UPPERCASE attribute
- Set operators
- Subqueries

"Example 15: Partitioned NUPI Specification With a USI On the Same Column Set and Partitioned By a RANGE_N Expression" on page 1-264 through "Example 21: Illegal Specification of UPI Without All Partitioning Columns Included in Primary Index Definition" on page 1-267 provide several examples of valid partitioning expressions.

13 HASHROW and HASHBUCKET *are* permitted.

Restrictions for Partitioned Primary Indexes

The following restrictions apply to partitioned primary indexes:

- You cannot define a PPI on a global temporary or volatile table.
- You cannot define a PPI using an identity column.
- You can only define a PPI as UNIQUE if all the columns referenced in *partitioning_expression* are also referenced in the column list that defines the primary index.
- 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.[14]
- 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.
- Columns referenced in *partitioning_expression* must be drawn from the set of columns defined for the table on which the PPI is defined.
- If the data type for the result of *partitioning_expression* is not INTEGER, then the value is implicitly cast to the INTEGER type.
- If you attempt to insert or update a row of a table with a PPI, and the partitioning expression does not produce a non-null result in the range 1 - 65 535, even after casting to INTEGER if the result is not already typed as INTEGER, then an error is returned for the insert or update.

  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.
- Because DATE values can be cast to INTEGER, they are valid types for partitioning expressions; however, only values between DATE '1900-01-01' and DATE '1906-12-31' have values between 1 and 65 535 when cast to INTEGER.

14 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.

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 65 533 (see the description of CASE_N in *Teradata RDBMS SQL Reference, Volume 5*).
- If *partitioning_expression* is only a RANGE_N function, then the number of ranges defined for the function must be less than or equal to 65 533[15] to allow for the NO RANGE [OR UNKNOWN] and UNKNOWN options.

  This restriction is enforced even if those options are not specified in order to make it possible to alter the index definition later to include them.

  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 RDBMS SQL Reference, Volume 5*) for additional information.
- Teradata derives a table-level CHECK constraint from the partitioning expression. The text for this derived constraint cannot exceed 8 192 characters.

See *Teradata RDBMS Database Design* for more information about primary indexes.

See "Redefining the Primary Index" on page 1-40 for information about how to use the ALTER TABLE statement to modify the primary index definition for a table.

Memory Limitations Related to Partitioned Primary Indexes

You might encounter several different memory limitations when working with PPI tables. The following table addresses these limitations by documenting the relevant error message numbers and providing explanations or workarounds for them.

15 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 65 533 conditions.

<table>
<tr><th>Message Number</th><th>Problem</th><th>Possible Cause</th><th>Remedy</th></tr>
<tr><td rowspan="3">3708</td><td rowspan="3">Table header size limit exceeded (see “Table Header Format” in the chapter “Database-Level Considerations” in Teradata RDBMS Database Design).

The limit is 64Kbytes.</td><td>Too many columns in the table.</td><td>If possible, reduce the number of columns in the table.</td></tr>
<tr><td>Too many distinct values compressed in the table.</td><td>If possible, reduce the number of compressed values.</td></tr>
<tr><td>Partitioning expression is too complex.</td><td>Simplify the partitioning expression.

If possible, use RANGE_N instead of CASE_N.</td></tr>
<tr><td rowspan="2">3710</td><td rowspan="2">Parser memory size limit exceeded.

The limit is variable and is determined by the value of the MaxParseTreeSegs field in DBSControl.</td><td>Current values for DBSControl record fields do not allocate enough parser memory to process the request.</td><td>Change the values for the following DBSControl fields to the specified settings:

<table><tr><th>IF the problem occurs with PPI tables …</th><th>THEN change the value for …</th></tr><tr><td>without hash or join indexes</td><td>MaxParseTreeSegs to 2000.</td></tr><tr><td>with hash or join indexes</td><td>• MaxParseTreeSegs to 2000.

and

• OptMaxGlobalJI to 8.</td></tr></table>
You might find that your query workloads are such that these values need to be increased still further.

OptMaxGlobalJI is an internal DBSControl field. You must contact a Teradata support representative to change its value.</td></tr>
<tr><td>Partitioning expression is too complex.</td><td>Simplify the partitioning expression.

If possible, use RANGE_N instead of CASE_N.</td></tr>
<tr><td>3891</td><td>Check constraint text size limit exceeded.

The limit is 8 192 characters.

The check constraint text can be explicit or derived from the partitioning expression for the PPI.</td><td>Text for the partitioning expression in the PPI definition is too long.</td><td>Reduce the partitioning expression text to 8 172[a] or fewer characters.</td></tr>
</table>

a. 20 characters of the 8 192 character limit are allocated to the text string "<partitioning_expression> BETWEEN 1 AND 65535" by default.

Load Utilities and PPI Tables

Load utilities are supported on PPI tables with the following restrictions and advisories:

- MultiLoad does not support tables with USIs. Many PPI tables have USIs.
- 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 secondary indexes of any kind.
- Neither MultiLoad nor FastLoad support hash and join indexes.
- You should specify values for the partitioning column set when performing TPump deletes and updates to avoid lock contention problems that can degrade performance.
- You should avoid updating primary index and partitioning columns with TPump to minimize performance degradation.

Archive/Restore and PPI Tables

You cannot restore or copy an archive that contains PPI tables to a pre-V2R5.0 system.

There is neither upward nor downward compatibility between permanent journals logged prior to release V2R5.0 or after.

If you restore or copy a PPI table to another system, you should revalidate the table headers on the target system after the restore or copy is completed.

Because many MODIFY PRIMARY INDEX 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/Restore 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.

Secondary Indexes

Up to 32 secondary and join indexes (in any combination) can be defined for one table, and any secondary index can be unique or nonunique.

Secondary indexes can be added to an existing table using the CREATE INDEX statement.

As a general guideline for decision support applications, whenever you define the primary index for a table to be a NUPI, you should define a USI on its primary key. USIs are generally *not* recommended for OLTP applications.

Each secondary index has an associated secondary index subtable. Each row in a secondary index subtable contains the indexed column value and one or more row IDs that point to the base table data rows containing that value.

Rows for this type of secondary index …	Are stored in a subtable on …
USI	a different AMP from the rows they reference.
NUSI	the same AMP as the rows they reference.

Secondary indexes can greatly reduce table search time for set selections. However, updates (including delete, insert, and update) to tables with secondary indexes require more I/O because of the required transient journaling, the possibility that secondary index subtables must be updated (if the indexed column 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.

You should always test the performance of a secondary index before implementing it.

PRIMARY KEY and UNIQUE Constraints Versus Primary Indexes

You can define the primary index for a table using PRIMARY KEY and UNIQUE constraints syntax 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 one primary key and *must* have one, but only one, primary index.
- 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 (USI) by default.
- If a primary key is specified in a CREATE TABLE statement, but a primary index is not, then the primary key is mapped to a unique primary index (UPI) by default.
- If neither primary index nor primary key is specified in a CREATE TABLE statement then the first UNIQUE constraint is defined as the unique primary index (UPI).

- If there is no primary index, primary key, or UNIQUE constraint defined in a CREATE TABLE statement, then the first column of the table is used as the primary index. This index is nonunique (NUPI) by default except for the case of a single column table defined with the SET (no duplicate rows permitted) option, in which case it is unique (UPI).
- Columns defined as either of the following *cannot* be defined to be nullable:
  - PRIMARY KEY
  - UNIQUE
- Columns defined as any of the following can be defined as nullable; however, you should *not* define any of these table elements to be nullable unless you have an extremely compelling reason to do so:
  - PRIMARY INDEX
  - UNIQUE PRIMARY INDEX
  - INDEX
  - UNIQUE INDEX
- You cannot define a PRIMARY KEY or UNIQUE constraint with the same column set as an INDEX defined on the same table.
- You cannot define a UNIQUE constraint explicitly on the same columns as a PRIMARY KEY. Note that internally, Teradata does define each primary key set with a UNIQUE NOT NULL constraint.

Nullable Columns Are Valid for Unique Indexes

While columns defined as PRIMARY KEY or UNIQUE cannot be nullable, nullable columns are valid for unique and unique primary index column definitions. This is not good database design practice, and you should only permit nullable columns in index definitions under extraordinary conditions.

IF an index declared as UNIQUE or UNIQUE PRIMARY consists of …	THEN …
a single column	only one row can exist in the table with a null for that column.
multiple columns	the table cannot have more than one row in which all the non-null values are identical because duplicate rows are not permitted when uniqueness is defined.

CREATE TABLE (Temporary/Volatile Table Preservation Clause)

Introduction

This clause instructs the database manager 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.

ON COMMIT DELETE/PRESERVE ROWS

The following table explains how transaction mode affects the timing of the ON COMMIT action.

In this transaction mode …	The ON COMMIT action is applied at this time …
ANSI	when the application issues a COMMIT statement.
Teradata	• the end of each request (implicit transaction). • when the ET statement is processed (explicit transactions). If you are using nested BT statements, then ON COMMIT is applied when the ET for the outermost BT is processed.

CREATE TABLE (AS Clause)

Introduction

The capability for rapidly creating work or test tables that are identical to, or based on, base tables in a database is frequently useful. For example, you can use work tables with the same structure as base tables to speed up processing because it is much faster to insert rows into an empty 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.

Similarly, it is often useful to create test tables that are identical to some or all of the base tables used by live applications for prototyping new applications on a dummy 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 definitions (or a subset) to the new table, while the AS … WITH DATA clause copies both the source table definitions (or a subset) and its data to the new table.

Copying Table Definitions

The AS clause copies definitions for an existing table to a new table. You can copy either all or a selected subset of the columns from the existing table.

The difference between the AS options is explained in the following table.

Use this option …	To copy from an existing source to a new target table …
AS … WITH NO DATA	all or selected column definitions.
AS … WITH DATA	all or selected column definitions *and* the data associated with those columns.

To copy only a subset of the source table, 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.

Authorization

Access privileges are identical to those for the standard CREATE TABLE statement.

Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table

If you do not specify an explicit table kind, the table kind of the target table defaults to the default kind for the session, *not* to the default kind of the source table. The target table does *not* inherit the table kind of its source table by default.

For example, if you copy a SET source table to a target table in ANSI session mode, 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.

The following table restates this information.

IF you copy a source table of this table kind to a target table …	WHILE in this session mode …	THEN the target table has this table kind unless you explicitly specify a different table kind …
SET	ANSI	MULTISET
	Teradata	SET
MULTISET	ANSI	MULTISET
	Teradata	SET

Exclusions from Standard Syntax

The following list of attributes is excluded from the AS syntax for CREATE TABLE:

- Column definitions *cannot* contain either of the following attributes:
  - Data types
  - Referential integrity constraints
- Column definitions *can* contain valid data type attributes.
- 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:

- If you do not explicitly specify a column set to define the primary index, primary key, or a unique constraint for a table, then the first column defined for the table is selected to be its *nonunique* primary index by default (see "Introduction" on page 1-239).
- A column that maps to an expression assumes the data type of the expression result.

- When an existing table (not a subquery) is specified in the AS ... WITH NO DATA clause, then the new table always assumes the following table-level characteristics of the source table:
  - Column structures
  - Fallback options
  - Journaling options (except for the case of the new table being a global temporary or volatile table, in which case permanent journaling is *not* copied)
  - All defined indexes except join indexes.
- 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 right-to-left order as they appear in the copied table or subquery select list.
- If a partitioned primary index definition is copied to a new target table and no index definitions are specified, 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.

Attributes Not Copied To The Target Table

The following list describes attributes that are not copied to a target table from the source table 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
- Source table column statistics are not copied to the target table for the following reasons:
  - They are not attributes of a table
  - Their reliability depends on their recency
- When an existing 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.
- The table kind (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

Using Subqueries To Customize An AS Clause

Subqueries are a powerful means for customizing the definition and data for a target table by excluding selected attributes of the source table from the copy.

The following list itemizes important properties and rules for use of subqueries expressed in an AS clause:

- Any 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.
  - You cannot SELECT on a join index.
- You must specify explicit column names if the select list contains an expression not named by an AS or NAMED clause. The number of columns defined must be equal to the number of expressions used in the subquery.

IF ...	THEN ...
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.

-

IF a subquery contains ...	THEN ...
only 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.
a 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:
  - All column-level attributes are copied.
  - No table-level attributes, including indexes, are copied.
  - The following list of standard user or database default attributes is copied from the source table to the target table unless you specify explicit values are specified 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	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
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 a 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" on page 1-12) 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 Teradata returns a constraint violation error to the requestor.

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 nonequivalent 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 on *source_table* to define the columns in *target_table*.

Note that the second statement is coded with a WHERE clause that always tests FALSE. This rather inelegantly coded clause ensures that only the table definition for source_table, and not its data, is 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 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;
```

Nonequivalence 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" on page 1-252).

```
CREATE TABLE target_table AS
 source_table
WITH NO DATA;

CREATE TABLE target_table AS
 (SELECT *
  FROM source_table
 )
WITH NO DATA;
```

Nonequivalence Set 2

This result parallels Nonequivalence 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" on page 1-252).

```
CREATE TABLE target_table AS
 source_table
WITH DATA;

CREATE TABLE target_table AS
 ( SELECT *
    FROM source_table
 )
 WITH DATA;
```

CREATE TABLE (Examples)

Introduction

The following examples illustrate the use of CREATE TABLE:

Example 1

The following CREATE TABLE statement defines the Employee table.

```
CREATE TABLE Employee
(EmpNo SMALLINT FORMAT '9(5)'
  CHECK (EmpNo >= 10001 AND EmpNo <= 32001) NOT NULL,
 Name VARCHAR(12) NOT NULL,
 DeptNo SMALLINT FORMAT '999'
  CHECK (DeptNo >= 100 AND DeptNo <= 900),
 JobTitle VARCHAR(12),
 Salary DECIMAL(8,2) FORMAT 'ZZZ,ZZ9.99'
  CHECK (Salary >= 1.00 AND Salary <= 999000.00),
 YrsExp BYTEINT FORMAT 'Z9'
  CHECK(YrsExp >= -99 AND YrsExp <=99),
 dob DATE FORMAT 'MMMbDDbYYYY' NOT NULL,
 Sex CHAR UPPERCASE NOT NULL
  CHECK (Sex IN ('M','F')),
 Race CHAR UPPERCASE,
 MStat CHAR UPPERCASE
  CHECK (MStat IN ('S','M','D','U')),
 EdLev BYTEINT FORMAT 'Z9'
  CHECK (EdLev >=0 AND EdLev <=22) NOT NULL,
 HCap BYTEINT FORMAT 'Z9'
  CHECK (HCap >= -99 AND HCap <= 99)
UNIQUE PRIMARY INDEX (EmpNo),
INDEX (Name);
```

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

For display purposes, the values for Sex, Race, and MStat are defined as UPPERCASE. The Name column is not defined as a unique index because it is possible for two or more employees to have the same name.

Example 2: Use of DATABLOCKSIZE and FREESPACE

The following example creates the same Employee table as in the above example but includes options to specify a maximum data block size of 16 384 bytes (32 sectors) and 10 percent freespace:

```
CREATE TABLE Employee,
 DATABLOCKSIZE = 16384 BYTES,
 FREESPACE = 10 PERCENT,
 (EmpNo SMALLINT FORMAT '9(5)'
   CHECK (EmpNo >= 10001 AND EmpNo <= 32001) NOT NULL,
  Name VARCHAR(12) NOT NULL,
  DeptNo SMALLINT FORMAT '999'
   CHECK (DeptNo >= 100 AND DeptNo <= 900),
  JobTitle VARCHAR(12),
  Salary DECIMAL(8,2) FORMAT 'ZZZ,ZZ9.99'
   CHECK (Salary >= 1.00 AND Salary <= 999000.00),
  YrsExp BYTEINT FORMAT 'Z9'
   CHECK (YrsExp >= -99 AND YrsExp <=99),
  dob DATE FORMAT 'MMMbDDbYYYY' NOT NULL,
  Sex CHAR UPPERCASE NOT NULL,
  Race CHAR UPPERCASE,
  MStat CHAR UPPERCASE,
  EdLev BYTEINT FORMAT 'Z9'
   CHECK(EdLev >=0 AND EdLev <= 22) NOT NULL,
  HCap BYTEINT FORMAT 'Z9'
   CHECK (BCap >= -99 AND HCap <= 99)
UNIQUE PRIMARY INDEX (EmpNo),
INDEX (Name);
```

Example 3: Multivalued Compression

The following example compresses all EmpName values of "Smith," "Wong," and "Rodriguez" and all DOB values of "1972-02-29," "1976-02-29" and null.

```
CREATE TABLE Employee
  (EmpNo   INT      NOT NULL,
   EmpName CHAR(30) NOT NULL COMPRESS ('Smith',
                              'Wong','Rodriguez')
   …
   DOB     DATE               COMPRESS (NULL, DATE '1972-02-29',
                              DATE '1976-02-29')
   …
);
```

Example 4: Identity Column

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

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

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

No CYCLE parameter is specified for this example.

IF you …	THEN the …
do not specify CYCLE	specification defaults to NO CYCLE.
specify CYCLE	numbering sequence generated is as follows: 1 000, 1 010, 1 020 … 300 000, 0, 10, 20, 30 …

No warning message is returned when numbers are cycled.

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

Example 5: Identity Column

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

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

Example 6: Referential Integrity Constraints

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

This specific reference in the example ...	Is this level constraint ...
explicit foreign key columns (field_1, field_2) in table_1 defining a foreign key to table_2. For this to work, there must be columns identically typed to the (field_1,field_2) columns in table_1 (call them upi_field_1 and upi_field_2) that constitute the unique primary index columns for table_2. In other words, columns upi_field_1 and upi_field_2 in table_2 must both be defined as type INTEGER and must constitute the unique primary index for table_2. Otherwise, an error is returned.	Table
implicit foreign key column field_1 in table_1 explicitly referencing column field_1 in table_3. table_1(field_1) and table_3(field_1) must both be typed INTEGER, but table_3(field_1) need not be the unique primary index for table_3.	Column

```
CREATE TABLE table_1
 (field_1 INTEGER NOT NULL REFERENCES table_3(field_1),
  field_2 INTEGER,
FOREIGN KEY (field_1, field_2) REFERENCES table_2)
PRIMARY INDEX (field_1);
```

Example 7: Batch Referential Integrity Constraint

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

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

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

Example 8: Column-Level Referential Constraint

The first statement creates a column-level referential integrity constraint on column A1 with column B1 in table B. Referential integrity is enforced for this relationship.

The second statement specifies a column-level Referential Constraint relationship on column A1 with column B1 in table B. Referential integrity is not enforced for this relationship.

```
CREATE TABLE A (
 A1 INTEGER REFERENCES B(B1),
 A2 CHAR(10))
PRIMARY INDEX (A1);

CREATE TABLE A (
 A1 INTEGER REFERENCES WITH NO CHECK OPTION B(B1),
 A2 CHAR(10))
PRIMARY INDEX (A1);
```

Example 9: Table-Level Referential Constraint

The first statement creates a table-level standard referential integrity constraint on foreign key column A2 with column D1 in table D. Referential integrity is enforced for this relationship.

The second statement creates a table-level Referential Constraint relationship on column A2 with column D1 in table D. Referential integrity is not enforced for this relationship.

```
CREATE TABLE A (
 A1 INTEGER,
 A2 CHAR(10),
 A3 INTEGER,
FOREIGN KEY (A2) REFERENCES D(D1))
PRIMARY INDEX (A1);
```

```
   CREATE TABLE A (
    A1 INTEGER,
    A2 CHAR(10),
    A3 INTEGER,
   FOREIGN KEY (A2) REFERENCES WITH NO CHECK OPTION D(D1))
PRIMARY INDEX (A1);
```

Example 10: Column-Level Named Constraints

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

```
CREATE TABLE good_1
 (field_1 INTEGER NOT NULL
   CONSTRAINT primary_1 PRIMARY KEY,
  field_2 INTEGER NOT NULL
   CONSTRAINT unique_1 UNIQUE,
  field_3 INTEGER
   CONSTRAINT check_1
    CHECK (field_3 > 0),
  field_4 INTEGER
   CONSTRAINT reference_1
    REFERENCES parent_1);
```

Example 11: Table-Level Named Constraints

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

```
CREATE TABLE good_2
 (field_1 INTEGER NOT NULL
  field_2 INTEGER NOT NULL,
  field_3 INTEGER NOT NULL,
  field_4 INTEGER NOT NULL,
  field_5 INTEGER,
  field_6 INTEGER,
   CONSTRAINT primary_1
 PRIMARY KEY (field_1, field_2),
   CONSTRAINT unique_1
    UNIQUE (field_3, field_4),
   CONSTRAINT check_1
    CHECK (field_3 > 0 OR field_4 IS NOT NULL),
   CONSTRAINT reference_1
 FOREIGN KEY (field_5, field_6)
  REFERENCES parent_1 (field_2, field_3));
```

Example 12: Table- and Column-Level Named and Unnamed Constraints

The statement in this example defines named and unnamed constraints at both the column and table levels.

```
CREATE TABLE good_3
 (field_1 INTEGER NOT NULL
   CONSTRAINT primary_1
 PRIMARY KEY,
  field_2 INTEGER NOT NULL
   CONSTRAINT unique_1
    UNIQUE
   CONSTRAINT check_1
    CHECK (field_2 <> 3)
   CONSTRAINT reference_1
    REFERENCES parent_1
    CHECK (field_2 > 0)
    REFERENCES parent_1 (field_4),
  field_3 INTEGER NOT NULL
  field_4 INTEGER NOT NULL
  field_5 INTEGER,
   CONSTRAINT unique_2
    UNIQUE (field_3),
   CONSTRAINT check_2
    CHECK (field_3 > 0 AND field_3 < 100),
   CONSTRAINT reference_2
 FOREIGN KEY (field_3)
  REFERENCES parent_1 (field_5),
    UNIQUE (field_4),
    CHECK (field_4 > field_5),
 FOREIGN KEY (field_4, field_5)
  REFERENCES parent_1 (field_6, fielt_7));
```

Example 13: Named and Unnamed CHECK Constraints at Both Levels

The statement in this example combines the three unnamed CHECKs for `field_1`. Constraint `check_0` and each of the named CHECKs for `field_2` are treated as table constraints.

```
CREATE TABLE good_4
(field_1 INTEGER
   CHECK (field_1 > 0)
   CHECK (field_1 < 999)
   CHECK (field_1 NOT IN (100,200,300)
  CONSTRAINT check_0
   CHECK (field_1 IS NOT NULL),
 field_2 INTEGER
  CONSTRAINT check_1
   CHECK (field_2 > 0)
   CHECK (field_2 < 999));
```

Example 14: Primary Index and Primary Key Specification

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

The Primary Key (`field_1`) is mapped to a unique secondary index. The UNIQUE constraint on `field_3` also is mapped to a USI. These two secondary indexes cannot be null. The Unique Primary Index (`field_2`) is the hashing index and can be null.

```
CREATE TABLE good_5
 (field_1 INTEGER NOT NULL PRIMARY KEY,
  field_2 INTEGER,
  field_3 INTEGER NOT NULL UNIQUE,
  field_4 INTEGER
)
UNIQUE PRIMARY INDEX (field_2);
```

Example 15: Partitioned NUPI Specification With a USI On the Same Column Set and Partitioned By a RANGE_N Expression

The following example creates a NUPPI on o_orderkey and a USI on the same column set. The statement bases its partitioning expression on the RANGE_N function on o_orderdate.

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

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

Example 16: Partitioned NUPI Specification Without a USI and Partitioned By a RANGE_N Expression

The following example creates a NUPPI on l_orderkey and bases its partitioning expression on the RANGE_N function on l_shipdate. No secondary indexes are defined.

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

Example 17: Partitioned UPI Specification Over a Narrow Range Using a RANGE_N Expression

The following example creates a UPPI on storeid, productid, and salesdate and bases its partitioning expression on the RANGE_N function on salesdate. No secondary indexes are defined.

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

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

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

Example 18: Partitioned UPI Specification Over a Broad Range Using a RANGE_N Expression

The following example is similar to "Example 17: Partitioned UPI Specification Over a Narrow Range Using a RANGE_N Expression" on page 1-265 except that it provides partitions that are one week wide over a 4 year interval.

```
CREATE TABLE SalesHistory
     (storeid      INTEGER NOT NULL,
      productid    INTEGER NOT NULL,
      salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      totalrevenue DECIMAL(13,2),
      totalsold    INTEGER,
      note         VARCHAR(256))
   UNIQUE PRIMARY INDEX (storeid, productid, salesdate)
    PARTITION BY RANGE_N(
     salesdate BETWEEN DATE '1997-01-01' AND DATE '2000-12-31'
     EACH INTERVAL '7' DAY);
```

Example 19: Partitioned UPI Specification Using an EXTRACT Expression

The following example creates a UPPI on storeid, productid, and salesdate and bases its partitioning expression on a simple EXTRACT function on salesdate. No secondary indexes are defined.

```
CREATE TABLE SalesByMonth
     (storeid      INTEGER NOT NULL,
      productid    INTEGER NOT NULL,
      salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      totalrevenue DECIMAL(13,2),
      totalsold    INTEGER,
      note         VARCHAR(256))
   UNIQUE PRIMARY INDEX (storeid, productid, salesdate)
     PARTITION BY EXTRACT(MONTH FROM salesdate);
```

Example 20: Partitioned UPI Specification Using a Simple Column Expression

The following example creates a UPPI on storeid, productid, and salesdate and bases its CASE_N partitioning expression on totalrevenue. Because totalrevenue is not defined in the primary index column set, the primary index on this table cannot be defined as unique.

```
CREATE TABLE StoreRevenue
     (storeid      INTEGER NOT NULL,
      productid    INTEGER NOT NULL,
      salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      totalrevenue DECIMAL(13,2),
      totalsold    INTEGER,
      note         VARCHAR(256))
   PRIMARY INDEX (storeid, productid, salesdate)
    PARTITION BY CASE_N(totalrevenue < 10000, totalrevenue <
    100000,totalrevenue < 1000000, NO CASE, UNKNOWN);
```

Example 21: Illegal Specification of UPI Without All Partitioning Columns Included in Primary Index Definition

The following example attempts to create a UPPI on storeid, product id, and salesdate, but fails because totalrevenue, on which the table is partitioned, is not included in the UPPI definition. A PPI cannot be unique unless all its partitioning columns are included in the index definition.

```
CREATE TABLE StoreRevenue
     (storeid      INTEGER NOT NULL,
      product id   INTEGER NOT NULL,
      salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      totalrevenue DECIMAL(13,2)
      totalsold    INTEGER
      note         VARCHAR(256))
   UNIQUE PRIMARY INDEX (storeid, productid, salesdate)
     PARTITION BY CASE_N(totalrevenue < 10000, totalrevenue <
       100000, totalrevenue < 1000000, NO CASE, UNKNOWN);
```

Example 22: Partitioned UPI Specification Using a Simple Arithmetic Expression

The following example creates a UPPI on storeid, productid, and salesdate and bases its partitioning expression on a simple arithmetic expression on storeid and productid. No secondary indexes are defined.

```
CREATE TABLE StoreProduct
     (storeid      INTEGER NOT NULL BETWEEN 0 AND 64,
      productid    INTEGER NOT NULL BETWEEN 0 and 999,
      salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      totalrevenue DECIMAL(13,2),
      totalsold    INTEGER,
      note         VARCHAR(256))
   UNIQUE PRIMARY INDEX (storeid, productid, salesdate)
     PARTITION BY storeid*1000 + productid + 1;
```

Example 23: Primary Key Specification

The statement in this example specifies a primary key only, so `field_2` is mapped to a unique primary index (UPI). In this case the PRIMARY index cannot be null.

```
CREATE TABLE good_6
 (field_1 INTEGER,
  field_2 INTEGER NOT NULL PRIMARY KEY);
```

Example 24: Neither Primary Index Nor Primary Key Is Specified

The statement in this example specifies neither a primary index nor a primary key. Teradata maps the first unique field, `field_2`, to a UPI, which cannot be null.

```
CREATE TABLE good_7
 (field_1 INTEGER,   field_2 INTEGER NOT NULL UNIQUE,
  field_3 INTEGER NOT NULL
   UNIQUE);
```

Example 25: Non-Valid Use of CHECK Constraint

The statement in this example is not valid because it contains duplicate constraint names (`dup_name`).

```
CREATE TABLE bad_1
 (field_1 INTEGER CONSTRAINT dup_name CHECK (field_1 > 0),
  field_2 INTEGER NOT NULL
   CONSTRAINT dup_name
    UNIQUE);
```

Teradata returns the error message:

```
Constraint with the same name 'dup_name' already
exists in table.
```

Example 26: Non-Valid Use of Primary Key Constraint

The statement in this example is not valid because it defines more than one primary key constraint for the table.

```
CREATE TABLE bad_2
 (field_1 INTEGER NOT NULL CONSTRAINT primary_1
 PRIMARY KEY,
  field_2 INTEGER NOT NULL,
 PRIMARY KEY (field_2));
```

Teradata returns the error message:

```
More than one primary index or primary key specified.
```

Example 27: Non-Valid Use of Constraint Name

The statement in this example is not valid because the constraint name exceeds the limit of 30 characters.

```
CREATE TABLE bad_3
 (field_1 INTEGER NOT NULL
   CONSTRAINT ThisIsAVeryLongNamedUniqueConstraint
    UNIQUE);
```

Teradata returns the following error message:

```
Name is longer than 30 characters.
```

Example 28: Local Journaling

As an example, given the following CREATE TABLE statements:

```
CREATE TABLE cyk.test_1, NO FALLBACK, AFTER JOURNAL
(x INTEGER, y INTEGER) PRIMARY INDEX(x);

CREATE TABLE cyk.test_2, NO FALLBACK,NOT LOCAL AFTER JOURNAL
(x INTEGER, y INTEGER) PRIMARY INDEX(x);

CREATE TABLE cyk.test_3, NO FALLBACK, LOCAL AFTER JOURNAL
(x INTEGER, y INTEGER) PRIMARY INDEX(x);
```

SHOW TABLE produces the following output:

```
CREATE TABLE cyk.test_1, NO FALLBACK,
NO BEFORE JOURNAL,
NOT LOCAL AFTER JOURNAL
(x INTEGER, y INTEGER) PRIMARY INDEX(x);

CREATE TABLE cyk.test_2, NO FALLBACK,
NO BEFORE JOURNAL,
NOT LOCAL AFTER JOURNAL
(x INTEGER, y INTEGER) PRIMARY INDEX(x);

CREATE TABLE cyk.test_3, NO FALLBACK,
NO BEFORE JOURNAL,
LOCAL AFTER JOURNAL
(x INTEGER, y INTEGER) PRIMARY INDEX(x);
```

Example 29: AS … WITH DATA With Default Column Names

The following CREATE TABLE … AS … WITH DATA statement selects all columns from source_table for inclusion in target_table and retains their names. The statement also copies the data from the selected columns of source_table into target_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table AS
 (SELECT *
  FROM source_table
 )
WITH DATA;
```

Example 30: AS … WITH NO DATA With Default Column Names

The following CREATE TABLE … AS … WITH NO DATA statement selects a subset of the columns from source_table for inclusion in target_table and retains their names.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table AS
 (SELECT column_1, column_2
  FROM source_table
 )
WITH NO DATA;
```

Example 31: AS … WITH DATA With Column Constraints

The following CREATE TABLE … AS … WITH DATA statement specifies column constraints. Note the absence of referential integrity constraints, which are not valid for AS clauses (see "Exclusive AS … WITH DATA Clause Rules" on page 1-254).

The statement also copies the data from the selected columns of source_table into target_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table (column_1 NOT NULL DEFAULT 0,
column_2) AS
 (SELECT column_x, column_y
  FROM source_table
 )
WITH DATA;
```

Example 32: AS … WITH DATA With Explicit Column Names

The following CREATE TABLE … AS … WITH DATA statement specifies column names in target_table that differ from their names in source_table. The statement also copies the data from the selected columns of source_table into target_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table (column_x, column_y) AS
 (SELECT column_1, column_2
  FROM source_table
 )
WITH DATA;
```

Example 33: AS … WITH NO DATA With Explicit Column Names

The following CREATE TABLE … AS … WITH NO DATA statement specifies column names for target_table as correlation names in the subquery that defines source_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table AS
 (SELECT column_x AS column_1, column_y AS column_2
  FROM source_table
 )
WITH NO DATA;
```

Example 34: AS … WITH NO DATA With Data Type Attributes

The following CREATE TABLE … AS … WITH NO DATA statement specifies data type attributes and uses an expression to define a column in target_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table (column_1 TITLE 'max_x', column_2)
AS
 (SELECT MAX(column_x), column_y
  FROM source_table
  GROUP BY 2
 )
WITH NO DATA;
```

Example 35: AS … WITH NO DATA With Data Type Attributes

The following CREATE TABLE … AS … WITH NO DATA statement creates the same target_table as Example 20 but codes it differently.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table AS
 (SELECT MAX(column_x) (TITLE 'max_x' NAMED column_1),
         column_y AS column_2
  FROM source_table
  GROUP BY 2
 )
WITH NO DATA;
```

Example 36: AS … WITH DATA With Unnamed Expressions

The following CREATE TABLE … AS … WITH DATA statement creates target_table from a subquery on source_table having unnamed expressions that define both columns for target_table. Note that the affected columns are named explicitly in the target_table definition to compensate. The statement also copies the data from the selected columns of source_table into target_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table (column_x, column_y) AS
 (SELECT column_x + 1, column_y + 1
  FROM source_table
 )
WITH DATA;
```

Example 37: AS … WITH DATA With an Unnamed Expression

The following CREATE TABLE … AS … WITH DATA statement creates target_table from a subquery having an unnamed aggregate expression. Note that the affected column is named explicitly in the target_table definition to compensate. The statement also copies the data from the selected columns of source_table into target_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table (column_x, column_y) AS
 (SELECT MAX(column_x), column_y
  FROM source_table
  GROUP BY 2
 )
WITH DATA;
```

Example 38: AS … WITH NO DATA With Named Expressions

The following CREATE TABLE … AS … WITH NO DATA statement creates target_table from a subquery on source_table where all column expressions are named explicitly. No column names are defined for target_table because it is unnecessary.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table AS
 (SELECT column_x + 1 AS column_x, column_y + 1 AS column_y
  FROM source_table
 )
WITH NO DATA;
```

Example 39: AS … WITH DATA With Index

The following CREATE TABLE … AS … WITH DATA statement creates a new INDEX on column_2 of target_table. The statement also copies the data from the selected columns of source_table into target_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table (column_1, column_2) AS
 (SELECT *
  FROM source_table
 )
WITH DATA
INDEX (column_2);
```

Example 40: AS … WITH NO DATA With Index

The following CREATE TABLE … AS … WITH NO DATA statement creates a new INDEX on column_y of target_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE TABLE target_table AS
 (SELECT column_x, column_y
  FROM source_table
 )
WITH DATA
INDEX (column_y);
```

Example 41: GLOBAL TEMPORARY TABLE … AS … WITH NO DATA

The following CREATE TABLE … AS … WITH NO DATA statement creates a new global temporary table using all the column definitions from source_table with default definitions for all table options clause attributes (see "Using Subqueries To Customize An AS Clause" on page 1-252).

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE GLOBAL TEMPORARY TABLE target_table AS
 (SELECT *
  FROM source_table
 )
WITH NO DATA;
```

Example 42: CREATE VOLATILE TABLE … AS … WITH DATA

The following CREATE TABLE … AS … WITH DATA statement creates a new volatile table using all the column definitions and data from source_table.

Because no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of source_table (see "Target Table Kind Defaults Are Based On The Current Session Mode, Not On The Table Kind of the Source Table" on page 1-250).

```
CREATE VOLATILE TABLE target_table AS
 source_table
WITH DATA;
```

CREATE TRIGGER
REPLACE TRIGGER

Purpose

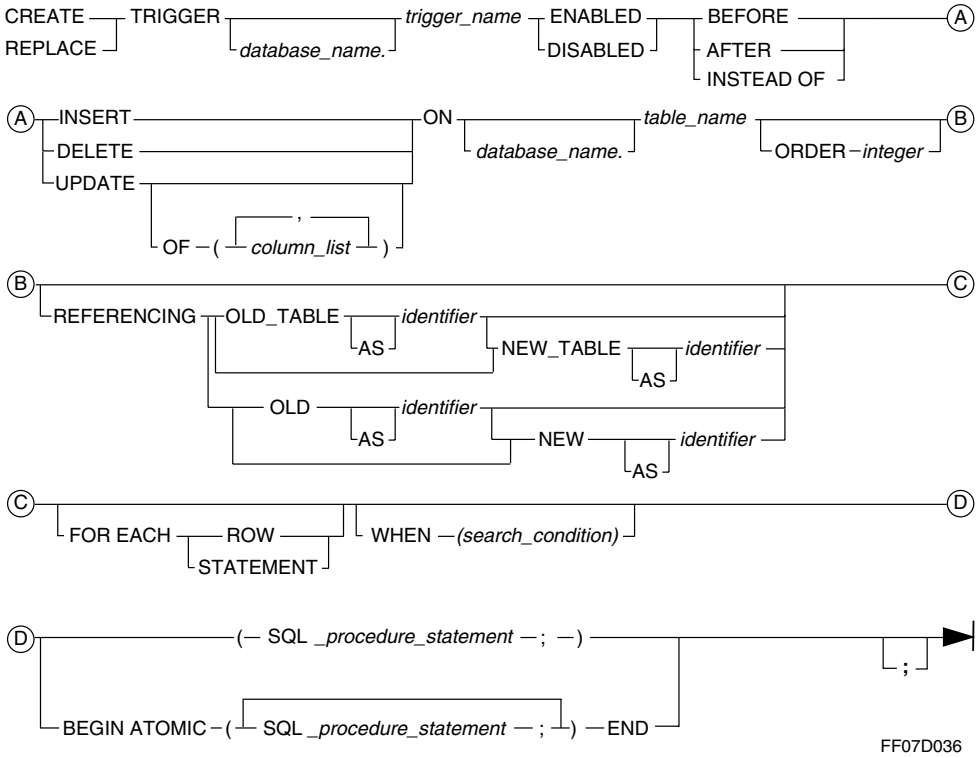
CREATE TRIGGER creates a new trigger.

REPLACE TRIGGER changes the definition for a trigger without having to drop and recreate it.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
database_name	an optional qualifier for the *trigger_name*. A *trigger_name* must be unique within the database.
trigger_name	the name of the trigger being created or replaced.
ENABLED	the keyword that enables a trigger to fire.

Syntax Element …	Specifies …
DISABLED	the keyword that disables a trigger from firing. A disabled trigger must be enabled before it can fire (see "ALTER TRIGGER" on page 1-65).
BEFORE	that the trigger fires *before* the triggering statement performs. If you create a BEFORE statement trigger or a BEFORE row trigger that contains data-changing statements, Teradata returns a warning message (see "ANSI Compliance" on page 1-280).
AFTER	that the trigger fires after the triggering statement performs.
INSTEAD OF	that the trigger fires instead of the triggering statement. In this case, the triggering statement never acts on the subject table. If you create an INSTEAD OF trigger, Teradata returns a warning message (see "ANSI Compliance" on page 1-280).
INSERT	that the triggering statement for this trigger is an INSERT or INSERT … SELECT.
DELETE	that the triggering statement for this trigger is a DELETE.
UPDATE	that the triggering statement for this trigger is an UPDATE.
column_list	the list of column names on which an UPDATE statement fires this trigger. The trigger will fire on the update of any column in the list. If no list is specified, the default is all columns. The list cannot contain duplicate column names. The column list does not apply to INSERT or DELETE triggering statements.
database_name	an optional qualifier for the subject table.
table_name	the name of the subject table to be associated with this trigger. This cannot be a global temporary table, a volatile table, or a view.
ORDER	a keyword allowing you to control trigger execution order within a request. When more than one trigger defined on the same subject table has the same trigger action time, trigger event, and FOR EACH clause, then the order value determines the execution sequence.
integer	the integer value assigned to ORDER for controlling the order of trigger execution within a request.

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td colspan="2">Sequential processing of REFERENCING clause objects</td></tr>
<tr><td colspan="2">The referenced objects of a REFERENCING clause can be specified in any order, as detailed in the table below.</td></tr>
<tr><td></td><td>
<table>
<tr><th>IF the object of a FOR EACH clause is a …</th><th>THEN the referenced objects of a REFERENCING clause can be ordered as follows …</th></tr>
<tr><td>ROW</td><td>• OLD only
• NEW only
• OLD then NEW
• NEW then OLD</td></tr>
<tr><td>STATEMENT</td><td>• OLD_TABLE only
• NEW_TABLE only
• OLD_TABLE then NEW_TABLE
• NEW_TABLE then OLD_TABLE</td></tr>
</table>
</td></tr>
<tr><td>REFERENCING</td><td>a keyword that permits you to create a situation in which you can indirectly access before and after subject table rows for making comparisons or for use in SQL_procedure_statement.
The REFERENCING clause is optional and there is no default.</td></tr>
<tr><td>OLD_TABLE</td><td>a keyword introducing an alias for the qualified results table before it is modified.
You can only use OLD_TABLE with a statement trigger (defined by a FOR EACH STATEMENT clause).</td></tr>
<tr><td>identifier</td><td>the correlation name for the premodification qualified subject table rows.</td></tr>
<tr><td>NEW_TABLE</td><td>a keyword introducing a correlation name for the qualified results table after it has been modified.
You can only use NEW_TABLE with a statement trigger (defined by a FOR EACH STATEMENT clause).</td></tr>
<tr><td>identifier</td><td>the correlation name for the postmodification qualified subject table rows.</td></tr>
<tr><td>OLD</td><td>a keyword introducing a correlation name for a column in a qualified row before it is modified.
You can only use OLD with a row trigger (defined by a FOR EACH ROW clause).
When the trigger event is an INSERT statement, then you cannot specify OLD.</td></tr>
</table>

Syntax Element …	Specifies …
identifier	the correlation name for a premodification qualified subject table row.
NEW	a keyword introducing a correlation name for a column in a qualified row after it has been modified. You can only use NEW with a row trigger (defined by a FOR EACH ROW clause). When the trigger event is a DELETE statement, then you cannot specify NEW.
identifier	the correlation name for a postmodification qualified subject table row.
FOR EACH ROW	keywords specifying that the trigger is to fire for each qualified row.
FOR EACH STATEMENT	keywords specifying that the trigger is to fire once per processed statement. FOR EACH STATEMENT is the default.
WHEN	keyword introducing a search condition clause. This is optional and has no default.
search_condition	logic that further refines the conditions for firing the trigger. The search condition evaluates to either TRUE or FALSE (NULL is treated as a FALSE result), TRUE meaning that at least one row qualifies. When true, the *SQL_procedure_statement* (triggered action) fires; else, it does not. Aggregates are allowed in the search condition logic only if they are specified in a subquery. Any expression that does not evaluate to TRUE or FALSE (such as a join expression) is not permitted as the *search_condition* for a WHEN clause.
BEGIN ATOMIC	keyword introducing multiple triggered action statements. If you begin the triggered action statements clause with BEGIN ATOMIC, then you must also terminate it with the END keyword.

Syntax Element …	Specifies …
SQL_procedure_statement	one or more of the following triggered action statements: • ABORT/ROLLBACK • DELETE • INSERT • INSERT … SELECT • UPDATE • A macro containing any combination (including none) of these valid statements can also be performed in a trigger. Each triggered action statement in an *SQL_procedure_statement* list must be terminated with a SEMICOLON character. Triggered action statements cannot act on the subject table because they cannot be recursive. Cascading triggers are bound by the same rule. When the triggered action statement is an INSERT … SELECT that selects aggregates, you must also specify a GROUP BY clause or an error is returned when you attempt to create the trigger.
END	keyword terminating the statement block introduced by the BEGIN ATOMIC keywords. If you use BEGIN ATOMIC, then you must also specify END to terminate the triggered action statements clause.

ANSI Compliance

CREATE TRIGGER is partly ANSI SQL-99-compliant.

The following trigger types are not ANSI-SQL-99-compliant and are scheduled to be removed in a future release:

- INSTEAD OF triggers
- BEFORE statement triggers
- BEFORE row triggers that contain data-changing statements

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

Authorization

To create a trigger, you need the following privileges:

- CREATE TRIGGER on the subject table or the database
- SELECT on any column referenced in a WHEN clause or a triggered SQL statement subquery
- Depending on the triggered SQL statement, INSERT, UPDATE, or DELETE on the triggered SQL statement target table.

To replace a trigger, you need the following privileges:

- DROP TRIGGER on the subject table or the database

  The exception is when you use REPLACE TRIGGER when no target trigger exists and you instead create a new trigger.
- SELECT on any column referenced in a WHEN clause or a triggered SQL statement subquery
- Depending on the triggered SQL statement, INSERT, UPDATE, or DELETE on the triggered SQL statement target table.

What REPLACE TRIGGER Does

In effect, the REPLACE TRIGGER statement operates as a DROP followed by a CREATE, except for the handling of access privileges. All access privileges that were granted directly on the original trigger are retained.

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 remains in place as it was prior to the performance of the REPLACE TRIGGER statement (it is not dropped). This is analogous to a ROLLBACK on the operation.

Triggered Events

Triggered events (or triggering actions) are the data modification language statements that cause a trigger to fire. In the syntax diagram, this component corresponds to the set of three DML statements following the trigger action time definition. The entry defines the trigger event that fires the trigger defined by the CREATE/REPLACE TRIGGER statement.

Triggering actions are external to the trigger process. Their importance is that their action, from within an application, attempts to modify a subject table that has one or more triggers defined. This action causes the trigger (or triggers) to fire, modifying the contents of other tables in the database.

The only statements that can act as a triggered event are the following:

- DELETE
- INSERT
- INSERT … SELECT
- MERGE
- UPDATE

The only statement that can trigger events on specific columns is UPDATE. The other statements trigger only on tables. Using UPDATE, you can specify columns on which an UPDATE causes a trigger to fire.

All these statement options are valid for both row and statement triggers.

Triggered Action Statement

A triggered action is the component of a trigger that performs a task. It can contain a condition, a statement or row delimiter, and certain SQL statements. In the syntax diagram, this component is defined to be the variable *SQL_procedure_statement*.

The only required portion of a triggered action is the triggered SQL statement, which can be either a single statement or a statement block demarcated by a BEGIN ATOMIC-END.

Statement triggers are limited to performing the following SQL statements:

- ABORT/ROLLBACK
- DELETE
- INSERT
- INSERT … SELECT
- MERGE
- UPDATE
- Any macro containing these statements

All these statement options are valid for statement triggers.

Row triggers are limited to performing the following SQL statements:

- Simple INSERT statements. A row trigger cannot perform INSERT … SELECT statements.
- ABORT and ROLLBACK statements.
- Macros that contain simple INSERT, ABORT, or ROLLBACK statements.

If a triggered action statement specification selects aggregates in an INSERT … SELECT without also specifying a GROUP BY clause, then the statement is not valid and an error is returned to the user. If the SELECT contains a GROUP BY clause, then aggregates are permitted.

Triggered actions cannot modify the subject table. This ensures that recursive modification of the table does not occur, eliminating any possibility of deadly loops developing.

Restriction on Triggers and Identity Columns

You cannot create or replace a trigger definition if its triggering action statement is an INSERT on a subject table that has an identity column and the triggered statement or WHEN clause of the FOR EACH clause of the trigger definition references that identity column.

Trigger Action Time

The trigger action time is defined by a keyword specifying when the trigger fires with respect to the triggering statement definition that follows it. In the syntax diagram, this is the set of three keywords following the ENABLED/DISABLED option.

The possible values for a trigger action time are as follows:

- BEFORE
- AFTER
- INSTEAD OF

IF the trigger action time is ...	THEN the trigger fires ...
BEFORE	before the triggered event modifies the subject table. When the trigger action completes its updates, the triggered event can modify the subject table.
AFTER	after the triggered event modifies the subject table.
INSTEAD OF	instead of the triggered event. The subject table is never modified.

The following trigger types are not ANSI-SQL-99-compliant and are scheduled to be removed in a future release:

- INSTEAD OF triggers
- BEFORE statement triggers
- BEFORE row triggers that contain data-changing statements

If you create a new trigger that uses these types, Teradata returns a warning message.

ORDER Value and Trigger Execution Order Within a Request

The value assigned with an ORDER keyword permits you to control trigger execution order within a request. If more than one trigger defined on the same table has the same trigger action time, trigger event, and FOR EACH clause, then this value determines the execution sequence.

The rules are as follows.

IF you ...	THEN ...
do not assign an order value	the value 32 767 is assigned by default. To change this to another value, you must recreate the trigger using the value you wish to assign to ORDER.
assign an order value	the number controls execution. You should leave a large numeric range and specify order values within this range. The maximum value for this integer is 32 767.

If two triggers have the same subject table, triggering event, trigger action time, order specification, and timestamp, then Teradata arbitrarily selects one to go first. Such a tie is an extremely unlikely event.

Making Comparisons Between Before and After Rows in 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 a subject table.

You cannot make direct comparisons between before and after rows of a subject table. Instead, you must make use of a REFERENCING clause and the keywords that assign names to the before (OLD or OLD_TABLE) and after (NEW or NEW_TABLE) rows.

Row and statement triggers force the REFERENCING clause to have two mutually exclusive states: a subject table reference can either specify a name for a new or old column value (row trigger) or for the new or old body of rows resulting from a trigger (statement trigger).

Row and Statement Triggers

There are two, mutually exclusive, types of trigger: row and statement triggers. You cannot combine row and statement operations within a single trigger.

Row triggers fire once per row changed by the triggering statement.

Statement triggers fire once per triggering statement.

You define these trigger types using the FOR EACH ROW or FOR EACH STATEMENT clause.

The REFERENCING clause can be used with both trigger types. This clause makes the old and new values visible to triggered statements and permits before and after triggered modification comparisons of data in the subject table.

A row trigger uses the OLD and NEW keywords to compare data, while a statement trigger uses the OLD_TABLE and NEW_TABLE keywords for the same purpose. Because a statement trigger can modify multiple rows in a subject table, the contents of OLD_TABLE and NEW_TABLE, if provided with a name (identified as *identifier* in the syntax diagram) can be joined with other tables.

A row trigger is limited to performing the following:

- Simple INSERT statements. A row trigger cannot perform INSERT … SELECT statements.
- ABORT and ROLLBACK statements.
- Macros that contain simple INSERT, ABORT, or ROLLBACK statements.

Rules for Using FOR EACH ROW

The rules for using a FOR EACH ROW clause are as follows:

- All subject table columns must be qualified with the old (OLD) or new (NEW) correlation name identifier.
- OLD and NEW can be specified once each and only once in a trigger definition.
- If both OLD and NEW are used in the same trigger definition, you can specify them in any order.
- OLD and NEW can only be used with a FOR EACH ROW clause.

  There are exceptions when OLD and NEW cannot be specified.

WHEN the trigger event is this statement …	THEN you cannot specify this keyword …
INSERT	OLD
DELETE	NEW

- The scope of the OLD and NEW names is the entire trigger definition. Neither name can be the same as that of the subject table, any name referenced in the OLD or NEW table correlation name list, or any table name referenced in the WHEN clause or triggered SQL statements.
- Comparisons between OLD and NEW names are done in the same way as any other comparison of object names in the database: the comparison is case blind and the shorter string is padded to the length of the longer string.
- You cannot SELECT from the old or new correlation names; instead, the old table and new table correlation names must be used.
- Triggered actions cannot reference a table that any of the triggered actions update. An error message results. Rewrite the statement as a FOR EACH STATEMENT trigger instead.

Rules for Using FOR EACH STATEMENT

The rules for using a FOR EACH STATEMENT clause are as follows:

- All subject table references must be qualified with the old (OLD_TABLE) or new (NEW_TABLE) values table correlation name identifier.
- OLD_TABLE and NEW_TABLE can be specified once each, and only once in a trigger definition.
- If both OLD_TABLE and NEW_TABLE are used in the same trigger definition, you can specify them in any order.
- OLD_TABLE and NEW_TABLE can only be used with a FOR EACH STATEMENT clause.

THIS table type ...	ACCESSES the whole collection of rows ...
OLD_TABLE	before any UPDATEs or DELETEs were applied to the subject table.
NEW_TABLE	after any UPDATEs or DELETEs were applied to the subject table, including default values, constraint checks, and so on.

- OLD_TABLE and NEW_TABLE see the set of rows qualified by the triggering statement. They do not see the entire subject table unless the triggering statement qualifies every row in the table. This means that in a million-row table where an UPDATE changed fifty rows, OLD_TABLE and NEW_TABLE see those fifty rows, not to the million.
- OLD_TABLE and NEW_TABLE cannot be updated; however, when specified in a query, the old and new table correlation names are treated like tables. As a result, a statement such as the following joins the “old” and “new” tables:

```
SELECT *
FROM old_table_alias, new_table_alias;
```

  There are exceptions when OLD_TABLE and NEW_TABLE cannot be specified.

WHEN the trigger event is this statement ...	THEN you cannot specify this keyword ...
INSERT	OLD_TABLE
DELETE	NEW_TABLE

- The scope of the OLD_TABLE and NEW_TABLE names is the entire trigger definition. Neither name can be the same as that of the subject table, any name referenced in the old or new correlation name list, or any table name referenced in the WHEN clause or triggered SQL statements.
- Comparisons between OLD_TABLE and NEW_TABLE names are done in the same way as any other comparison of object names in the database: the comparison is case blind and the shorter string is padded to the length of the longer string.

Cascading Triggers

Triggers can cascade (fire other triggers) until they exhaust system resources. There is no numeric limit to the number of triggers that can be involved in a cascade.

Restrictions and Limitations for Load Utilities

You cannot use FastLoad, MultiLoad, or Teradata Warehouse Builder to load data into base tables that have enabled triggers. If you attempt to load data into trigger-enabled base tables using either of these utilities, an error message returns and the load does not continue.

To load data into a trigger-enabled base table, you must disable all defined triggers before you can run FastLoad, MultiLoad, or Teradata Warehouse Builder.

Load utilities like TPump, which performs standard SQL inserts and updates, are supported for tables on which triggers are enabled.

Triggers, Partitioned Primary Indexes, and the ALTER TABLE Statement

You must disable some triggers before you can perform an ALTER TABLE statement that modifies or revalidates the primary index for a table.

IF you specify this clause in an ALTER TABLE statement that modifies or revaluates the primary index ...	THEN you must disable these triggers before you perform the statement ...
WITH DELETE	all delete triggers on *table_name*.
WITH INSERT	• all delete triggers on *table_name*. • all insert triggers on *save_table*.

You can re-enable the triggers after the ALTER TABLE statement finishes.

Triggers Supported Only for Base Tables

You can only define triggers for base tables. Triggers are supported for all base tables, whether they have a partitioned or nonpartitioned primary index.

You cannot define triggers for any of the following database objects.

- Global temporary tables
- Volatile tables
- Views

Triggers Not Supported for Positioned Updates and Deletes

Updatable cursors are not supported for triggers. If you attempt to perform positioned UPDATE or DELETE operations against a subject table with triggers defined, either an error or a failure is generated.

You must disable all triggers defined on a subject table prior to positioning cursors for UPDATEs or DELETEs on it.

Triggers Are Not Supported With Hash or Join Indexes

You cannot define triggers and hash or join indexes on the same table.

Non-Boolean Search Conditions Not Supported in WHEN Clause

You cannot specify any expression other than a Boolean expression (evaluates to TRUE or FALSE) for the left hand side of a search condition in a WHEN clause.

Table references or aggregates *cannot* appear on the left hand side of the search condition of a WHEN clause. This specifically excludes join and table expressions and aggregates. Aggregates and table references *are* allowed on the right hand side of the search condition (the subquery) of a WHEN clause.

For example, the following WHEN clause is illegal because t1 is a table reference, and the result of the condition does not evaluate as either TRUE or FALSE.

```
.
.
.
WHEN (t1.a <= (SELECT b
               FROM t2, t3
               WHERE t2.c = t3.c))
.
.
.
```

The following WHEN clause *is* legal because the aggregate appears on the right-hand side of the search condition and the left-hand side of the inequality condition is just a constant.

```
.
.
.
WHEN (10 <= (SELECT SUM(b)
             FROM t2, t3
             WHERE t2.c = t3.c))
.
.
.
```

See "Example 5: Valid Use of Subquery in WHEN Clause" on page 1-294, "Example 6: Non-Valid Use of Subquery in WHEN Clause" on page 1-295, and "Example 7: Non-Valid Use of Subquery in WHEN Clause" on page 1-295 for additional examples of correct and incorrect WHEN clause usage.

Triggers and Tactical Queries

Teradata triggers are parallelized. That is, they are implemented as multistatement requests along with the update statement that caused them to fire. Being part of an multistatement request means that triggers and the statement that fires them become a single optimizing 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.

To understand the impact performing the trigger has in the database, EXPLAIN the statement that causes the trigger to fire. In the following example, a trigger has been specified to perform each time a new supplier is added. To see the impact of the trigger on the SQL that inserts a supplier, EXPLAIN the INSERT statement.

```
CREATE TRIGGER supp_insert AFTER INSERT ON supplier
REFERENCING NEW AS n
FOR EACH ROW
 (INSERT INTO supplog 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')
```

```
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.supplog.      <-- The trigger
  2) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

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.

Example 1: Ensure That Parent Table Updates Propagate to Its Child

Suppose you want to ensure that all UPDATEs and DELETEs to a parent table are propagated to a child table. This example defines two triggers on the parent table to make this happen.

These are the table definitions.

```
CREATE TABLE Parent_Tab
(
 PrimeKey INTEGER,
 column_2 INTEGER,
 column_3 INTEGER
);

CREATE TABLE Child_Tab
(
 PrimeKey INTEGER,
 ForKey INTEGER,
 column_3 INTEGER
);
```

Now define the triggers on Parent_Tab, the subject table.

```
CREATE TRIGGER UpdateForKey
 BEFORE UPDATE OF (PrimeKey)
  ON Parent_Tab
  REFERENCING OLD_TABLE AS OldTable NEW_TABLE AS NewTable
 FOR EACH STATEMENT
 (
  UPDATE Child_Tab
  SET ForKey=NewTable.PrimeKey
  WHERE Child_Tab.ForKey=OldTable.PrimeKey;);

CREATE TRIGGER DelForKey
 BEFORE DELETE ON Parent_Tab
  REFERENCING OLD_TABLE AS OldTable
 FOR EACH STATEMENT
 (
 UPDATE Child_Tab
 SET ForKey=NULL
 WHERE ForKey=OldTable.PrimeKey;);
```

Example 2: Audit Log for Large Pay Raises

Triggers are particularly useful for audits of all kinds. This example inserts a log record whenever an employee gets a raise larger than ten percent.

These are the table definitions.

```
CREATE TABLE Employee
(
Name CHARACTER(30),
Deptid INTEGER,
Salary DECIMAL(10,2),
Comments CHARACTER(30)
);

CREATE TABLE SalaryLog
(
UserName CHARACTER(30),
EmpName CHARACTER(30),
OldSalary DECIMAL(10,2),
NewSalary DECIMAL(10,2)
);
```

Now define the trigger on the Employee table.

```
CREATE TRIGGER RaiseTrig
 AFTER UPDATE OF (Salary) ON Employee
  REFERENCING OLD AS OldRow NEW AS NewRow
 FOR EACH ROW
  WHEN ((NewRow.Salary-
   OldRow.Salary)/OldRow.Salary>.10)
  (
  INSERT INTO SalaryLog
  VALUES (USER, NewRow.Name, OldRow.Salary,
  NewRow.Salary);
  );
```

When the following request is processed, three INSERTs are made to the SalaryLog table.

```
UPDATE Employee
SET Salary = Salary*2, Comments = 'Employee of the Year'
WHERE Name = 'John Smith';

UPDATE EMPLOYEE
SET SALARY = SALARY*2, Comments = 'Employee of the Decade'
WHERE Name = 'Min Chan';

UPDATE EMPLOYEE
SET SALARY = SALARY*2, Comments = 'Employee of the Century'
WHERE Name = 'Lev Ulyanov';
```

Example 3: Prevention of Massive Updates During Business Hours

Suppose you want to exclude massive update processing during business hours in order to free machine time for online users. The following statement trigger operates on the entire set of rows changed by the triggering statement to ensure that large updates are staged during the day.

These are the table definitions.

```
CREATE TABLE ProdTab
(
column_1 INTEGER,
column_2 INTEGER,
column_3 INTEGER
);

CREATE TABLE ProdTabStage
(
 UserName CHARACTER(20),
 Oper CHARACTER(10),
 column_1 INTEGER,
 column_2 INTEGER,
 column_3 INTEGER
);
```

This statement creates a trigger to stage large updates during the work day.

```
CREATE TRIGGER StageUpd
  INSTEAD OF UPDATE ON ProdTab
  REFERENCING OLD_TABLE AS OldTab NEW_TABLE AS NewTab
 FOR EACH STATEMENT
  WHEN (TIME > 80000 AND TIME < 170000 AND 100 <
        (SELECT COUNT(column_1)
         FROM OldTab))
(
 INSERT ProdTabStage
  SELECT USER, 'UPDATED', column_1, column_2, column_3
  FROM NewTab;
);
```

This statement creates a trigger to stage large INSERTs during the work day.

```
CREATE TRIGGER StageIns
 INSTEAD OF INSERT ON ProdTab
  REFERENCING NEW_TABLE AS NewTab
 FOR EACH STATEMENT
  WHEN (TIME > 80000 AND TIME < 170000 AND 100 <
        (SELECT COUNT(column_1)
         FROM NewTab))
 (
  INSERT ProdTabStage
  SELECT USER, 'INSERTED', column_1, column_2, column_3
  FROM NewTab;
 );
```

This statement creates a trigger to stage large DELETEs during the work day.

```
CREATE TRIGGER StageDel
 INSTEAD OF DELETE ON ProdTab
  REFERENCING OLD_TABLE AS OldTab
 FOR EACH STATEMENT
  WHEN (TIME > 80000 AND TIME < 170000 AND 100 <
        (SELECT COUNT(column_1)
         FROM OldTab))
 (
  INSERT ProdTabStage
  SELECT USER, 'DELETED', column_1, column_2, column_3
   FROM OldTab; );
```

Example 4: Cascaded Triggers

This example demonstrates how one triggered action statement can cause another trigger to fire.

These are the table definitions.

```
CREATE TABLE Tab1
(
a INTEGER,
b INTEGER,
c INTEGER
);

CREATE TABLE Tab2
(
d INTEGER,
e INTEGER,
f INTEGER
);
```

```
CREATE TABLE Tab3
(
g INTEGER,
h INTEGER,
i INTEGER
);
```

These are the trigger definitions.

```
CREATE TRIGGER Trig1
 AFTER INSERT ON Tab1
  REFERENCING NEW AS NewRow
 FOR EACH ROW
 (
  INSERT INTO Tab2
  VALUES (NewRow.a + 10, NewRow.b + 10, NewRow.c);
 );

CREATE TRIGGER Trig2
 AFTER INSERT ON Tab2
  REFERENCING NEW AS NewRow
 FOR EACH ROW
 (
  INSERT INTO Tab3
  VALUES (NewRow.d + 100, NewRow.e + 100, NewRow.f);
 );
```

Now, suppose the following INSERT is submitted.

```
INSERT INTO Tab1 VALUES (1,2,3);
```

This triggering event fires a trigger to insert into Tab2. This operation is equivalent to the following INSERT statement.

```
INSERT INTO Tab2 VALUES (11,12,3);
```

This triggering event fires a trigger to insert into Tab3. This operation is equivalent to the following INSERT statement.

```
INSERT INTO Tab3 VALUES (111,112,3);
```

Example 5: Valid Use of Subquery in WHEN Clause

The following partial CREATE TRIGGER statement is legal because the correlation name previous references a single row (because this is a row trigger) and can be evaluated as TRUE or FALSE. The subquery is on the right hand side of the inequality condition.

```
CREATE TRIGGER...   BEFORE...   REFERENCING OLD AS previous
FOR EACH ROW   WHEN (previous.a <= (SELECT SUM(b) from t2, t3
WHERE t2.c = t3.c)) ...
```

Example 6: Non-Valid Use of Subquery in WHEN Clause

The following partial CREATE TRIGGER statement is illegal because the aggregate appears on the left hand side of the search condition.

```
CREATE TRIGGER ...    REFERENCING OLD AS previous    FOR EACH
ROW    WHEN (COUNT(previous.a) = 1)    ...
```

Example 7: Non-Valid Use of Subquery in WHEN Clause

The following partial CREATE TRIGGER statement is illegal because the correlation name previous references a table (a set of rows) because this is a statement trigger, and a set of rows cannot evaluate to TRUE or FALSE.

```
CREATE TRIGGER ...   BEFORE ...   REFERENCING OLD_TAB AS previous
FOR EACH STATEMENT    WHEN (previous.a IN (SELECT b FROM t2,
t3 WHERE t2.c = t3.c))    ...
```

CREATE USER

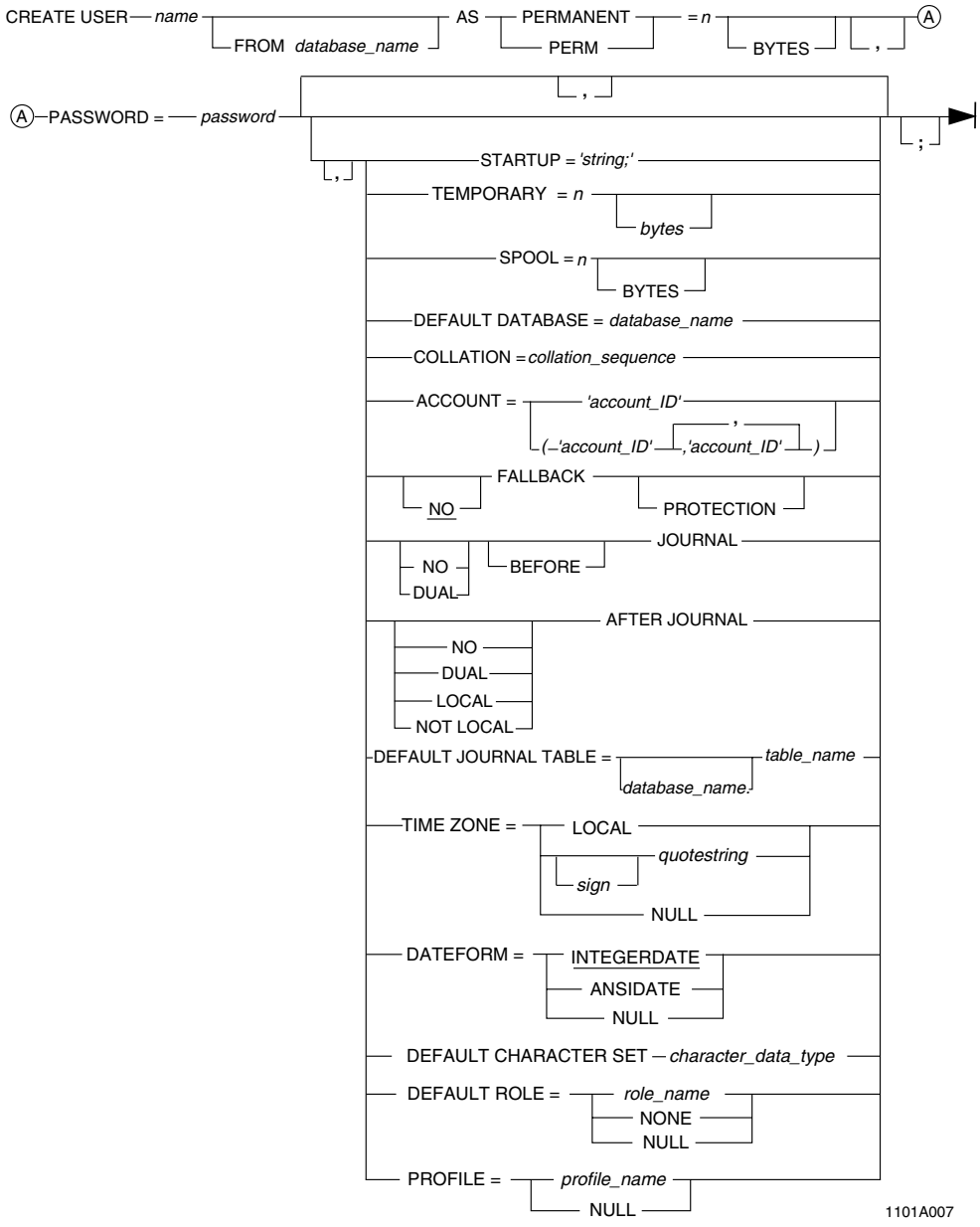
Purpose

Defines a new database user.

Invocation

Executable.

Syntax



where:

Syntax Element	Description
name	Specifies the name of the new user.
FROM	Introduces an optional clause that specifies the owner *database_name*.
database_name	Specifies the name of the immediate owner user or database. The default is the user name associated with the current session.
AS	Introduces the first user options. Subsequent options can be entered in any order.
PERMANENT = *n* BYTES	Specifies the number of bytes to be reserved for permanent storage of the new user database. The space is taken from unallocated space in the database of the immediate owner. The value *n* can be entered as an integer, decimal, or floating point. This option must be specified. There is no default. You cannot specify options until you specify the PERMANENT and PASSWORD options.
PASSWORD	Keyword introducing the clause that specifies *password* or NULL. This option must be specified. There is no default. You cannot specify options until you specify the PERMANENT and PASSWORD options.
password	Specifies the password associated with the new user. A password is considered to be a Teradata SQL word and can contain up to 30 characters. See "Names" in the chapter "SQL Lexicon" in *Teradata RDBMS SQL Reference, Volume 1*, for a description of Teradata SQL words.
STARTUP	Keyword introducing the clause declaring the startup string.
'string;'	Specifies one or more SQL statements, separated by SEMICOLON characters, that are performed to establish the session environment when the user logs on. Strings can be up to 255 characters, must be terminated by a SEMICOLON character, and must be enclosed by APOSTROPHE characters. The default is no startup string. A startup string can perform a macro, however, the USING modifier is not supported in a startup string. If a string includes a DDL statement, no other statement is allowed in the string. Strings are only performed when you log on through BTEQ.
TEMPORARY	Permits you to define how much space (in bytes) is to be allocated for creating temporary tables by this user. Note that temporary space is reserved prior to spool space for any user defined with this characteristic. Disk usage for a materialized temporary table is charged to the temporary space allocation of the user who referenced the table. If no temporary space is defined for a user, then the space allocated for any temporary tables referenced by that user is set to the maximum temporary space allocated for the immediate owner.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>SPOOL</td><td>Keyword introducing the bytes allocated to the spool file.</td></tr>
<tr><td>n</td><td>Specifies the number of bytes (n) to be allocated for spool files. The default is the largest value that is not greater then the owner spool space, and that is a multiple of the number of AMPs on the system.

n must not exceed the size of the owner spool space.

n can be entered as an integer, a decimal value, or as a floating point value.</td></tr>
<tr><td>BYTES</td><td>Optional unit assigned to n bytes allocated to the spool file. n refers to bytes, whether or not the BYTES keyword is specified.</td></tr>
<tr><td>DEFAULT DATABASE</td><td>Keyword introducing the database name, database_name.</td></tr>
<tr><td>database_name</td><td>Specifies the name of the default database established each time a user logs onto Teradata.

The user name associated with the CREATE USER statement is the default if the DEFAULT DATABASE clause is not specified.</td></tr>
<tr><td>COLLATION</td><td>Keyword introducing the clause that sets a collation default for the user.</td></tr>
<tr><td>collation_sequence</td><td>the name of the collation to be used as the default for this user.

<table>
<tr><th>This keyword ...</th><th>Defines this collation ...</th></tr>
<tr><td>ASCII</td><td>HOST

The collation is ASCII for configurations other than an IBM channel-attached host.

In this case, ASCII refers to the Teradata extension to the ASCII standard.</td></tr>
<tr><td>EBCDIC</td><td>HOST

The collation is EBCDIC for an IBM channel-attached host.</td></tr>
<tr><td>MULTINATIONAL</td><td>One of the European (diacritical) or Japanese language sort sequences (see “MULTINATIONAL Collation” on page 1-305).</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>collation_sequence
(continued)</td><td>
<table>
<tr><th>This keyword …</th><th>Defines this collation …</th></tr>
<tr><td>HOST</td><td>that of the logon client system.
This is the default.</td></tr>
<tr><td>CHARSET_COLL</td><td>HOST
The collation is binary in the order of the current client character set.
Strings are compared character-by-character and padded with the pad character for the appropriate character data type where necessary.
When not CASE SPECIFIC, lowercase characters are mapped to their uppercase counterparts.</td></tr>
<tr><td>JIS_COLL</td><td>Japanese Industrial Standards (JIS).
The collation is as follows.
1 Characters and symbols from the JIS X 0201 standard (in JIX X 0201 order)
2 Ideographs, characters, and symbols from the JIS X 0208 standard (in JIS X 0208 order)
3 Ideographs, characters, and symbols from the JIS 0212 standard (in JIS X 0212 order)
4 IBM Kanji ideographs not present in JIS X 0208 and JIS X 0212 (in KanjiEBCDIC order)
5 User-defined ideographs (in U+ order)
6 Any remaining characters in UNICODE (in U+ order)
Treatment of KANJISJIS, GRAPHIC, and LATIN character types is as if the data were first converted to UNICODE and then the appropriate JIS_COLL ordering is applied.</td></tr>
</table>
</td></tr>
<tr><td>ACCOUNT</td><td>Keyword introducing the account ID identifiers.</td></tr>
</table>

Syntax Element	Description

account_id	APOSTROPHE character-quoted identifier or a list of identifiers of the accounts that are to be charged for the space used by this user. The first (or only) identifier becomes the default account for the user.
	
	If *account_id* is specified in the logon string, it must have been specified in the ACCOUNT option of the last CREATE USER or MODIFY USER statement for the user.
	
	If *account_id* is not specified, it defaults to the account identifier of the immediate owner database.
	
	Each *account_id* can be up to 30 characters and must be enclosed by APOSTROPHE characters. A list of *account_ids* must be separated by COMMA characters and enclosed by LEFT PARENTHESIS and RIGHT PARENTHESIS characters.
	
	You can associate a Priority Scheduler performance group with each account ID you specify. This assignment specifies the relative service priority for the account based on the Priority Scheduler parameters specified for it by the system administrator.
	
	Each performance group for an account is named using the syntax `$group_name$`. *group_name* is limited to 16 single-byte characters (or 16 bytes for multibyte character sets). If you specify a performance group name when you create a user, the name is not verified at that time. Validation occurs only at logon.
	
	When you log on, the performance group name you specify is validated against existing performance group names. If no match is found, Teradata assigns one to you by default. The default is one of the following, depending on the situation.
	• First account string defined for the user.
	• Performance group M if the user has no account string with a defined performance group.
	
	The performance group names H, L, M, and R do not require the trailing $ character to maintain backward compatibility with earlier Teradata releases and can be typed as $H, $L, $M, and $R, respectively. For example, both of the following account specifications are valid.
	
	`'$H$acctPUB'`
	
	`'$HacctPUB'`
	
	The following interactive mode BTEQ logon commands are valid examples of how you can specify a performance group.
	
	`.LOGON elnino/GMK '$L1$&D&TacctPUB'`
	
	`.LOGON elnino/KGM '$L&D&TacctPUB'`
	
	Performance group names, their relative levels of service, and other related parameters are defined either by default or as individually customized variables by your system administrator. See "Priority Scheduler Facility" in *Teradata RDBMS Utilities, Volume 2* for more information about performance groups and other Priority Scheduler parameters.

Syntax Element	Description
[NO] FALLBACK [NO] FALLBACK PROTECTION	Specifies whether to create and store a duplicate copy of each table created in the new database. The default value is NO FALLBACK. This setting can be overridden for a particular table when the table is created (see "CREATE TABLE" on page 1-182). The FALLBACK keyword used alone implies PROTECTION.
[NO] JOURNAL DUAL JOURNAL BEFORE JOURNAL	Specifies the number of before change images to be maintained by default for each data table created in the new database. The JOURNAL keyword without NO or DUAL implies single copy journaling. If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection. The JOURNAL keyword without BEFORE implies both types (BEFORE and AFTER) of images.
[NO] AFTER JOURNAL DUAL AFTER JOURNAL LOCAL AFTER JOURNAL [NOT] LOCAL AFTER JOURNAL	Specifies the type of image to be maintained by default for data tables created in the new database. The JOURNAL keyword without NO or DUAL implies single copy journaling. If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection. NOT LOCAL and LOCAL specify whether single after-image journal rows for non-fallback data tables are written on the same virtual AMP (LOCAL) as the changed data rows, or on another virtual AMP in the cluster (NOT LOCAL). The JOURNAL keyword without AFTER implies both types (BEFORE and AFTER) of images. The default is no journaling If only AFTER JOURNAL is specified, then a before change image is not maintained. If both types are specified, the two specifications must not conflict. This setting can be overridden for a particular data table when the table is created (see "CREATE TABLE" on page 1-182).
DEFAULT JOURNAL TABLE = *database_name.table_name*	Identifies the default table that is to receive the journal images of data tables created in the new database. *table_name* must be defined if journaling is requested, but it need not reside in the new database. *table_name* is automatically created in the new database if *database_name* is not specified, or the new database is specified. If a different database is specified, then that database must exist and *table_name* must have been defined as its default journal table.
TIME ZONE	Defines the default time zone displacement for the user.
LOCAL	Sets the value for TIME ZONE to that defined for the system default. 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 use MODIFY USER.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>quotestring</td><td>Signed value that sets a non-system-default interval offset for converting the user TIME ZONE.

The format is hh:mm.</td></tr>
<tr><td>NULL</td><td>No default TIME ZONE exists for the user.</td></tr>
<tr><td>DATEFORM =</td><td>the default format for importing and exporting DATE values for the user.

For further information, see “DATE Formats” in the chapter “DateTime and Interval Data Types” in Teradata RDBMS SQL Reference, Volume 3.

Valid options are listed in the following table:

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>INTEGERDATE</td><td>Sets the DATEFORM option to import and export DATE values as encoded integers.

INTEGERDATE results in a default DATE format in field mode of 'YY/MM/DD' for date columns created and for date constants in character form. The 'YY/MM/DD' default DATE format can be changed by using the tdlocaledef utility. For details, see Teradata RDBMS Utilities.

INTEGERDATE is the DATEFORM default.</td></tr>
<tr><td>ANSIDATE</td><td>Sets the DATEFORM option to import and export DATE values as CHAR(10).

Results in a 'YYYY-MM-DD' date format for date columns created and for date constants in character form.</td></tr>
<tr><td>NULL</td><td>Sets DATEFORM null.

When DATEFORM is null, the default format for a date in field mode, and the default format for importing and exporting DATE values, is the same as the system default defined by the DATEFORM flag in the DBSControl record.</td></tr>
</table></td></tr>
<tr><td>DEFAULT CHARACTER SET</td><td>Sets the user default for character data type.

If no user default character type is defined, then one is assigned automatically according to the following rules.

<table>
<tr><th>WHEN no user default character type is defined for this type of site …</th><th>The definition defaults to this character_data_type …</th></tr>
<tr><td>U.S. and European</td><td>LATIN</td></tr>
<tr><td>Japanese</td><td>UNICODE</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>character_data_type</td><td>Keyword that defines the default character data type for the user.
<table><tr><th>This keyword …</th><th>Defines this character data type …</th></tr><tr><td>LATIN</td><td>fixed 8-bit Latin.</td></tr><tr><td>UNICODE</td><td>fixed 16-bit Unicode.</td></tr><tr><td>GRAPHIC</td><td>fixed 16-bit Unicode as defined by IBM Corporation for DB2.</td></tr><tr><td>KANJISJIS</td><td>mixed single and multibyte characters defined by KanjiSJIS.</td></tr><tr><td>KANJI1</td><td>mixed single and multibyte characters defined by KanjiEBCDIC, KanjiShift-JIS, or KanjiEUC, depending on the current character set for the session.</td></tr></table></td></tr>
<tr><td>DEFAULT ROLE</td><td>Defines the default role for the user.</td></tr>
<tr><td>role_name</td><td>The name of a role to assign as the default role for the user.</td></tr>
<tr><td>NONE
NULL</td><td>No default role exists for the user.</td></tr>
<tr><td>PROFILE</td><td>Assigns a profile to the user.</td></tr>
<tr><td>profile_name</td><td>The name of a profile to assign to the user.</td></tr>
<tr><td>NULL</td><td>No profile exists for the user.</td></tr>
</table>

ANSI Compliance

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

CREATE USER is functionally equivalent to the ANSI SQL-99 CREATE SCHEMA statement.

Authorization

To create a user, you must have CREATE USER privilege on the immediate owner database. The creator receives all privileges on the newly created user.

- CHECKPOINT
- CREATE DATABASE
- CREATE MACRO
- CREATE TABLE
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW
- DELETE
- DROP DATABASE
- DROP MACRO
- DROP PROCEDURE
- DROP TABLE
- DROP TRIGGER
- DROP USER
- DROP VIEW
- DUMP
- EXECUTE
- INSERT
- REVOKE
- RESTORE
- UPDATE

Privileges given to the creator and owner are WITH GRANT OPTION.

The newly created user also receives all relevant privileges on itself.

- CREATE MACRO
- CHECKPOINT
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DELETE
- DROP MACRO
- DROP PROCEDURE
- DROP TABLE
- DROP TABLE
- DROP TRIGGER
- DROP VIEW
- DUMP
- EXECUTE
- INSERT
- REVOKE
- RESTORE
- UPDATE

New users do *not* receive WITH GRANT OPTION privileges by default.

Newly created users are automatically granted the privileges needed to create and maintain objects in their space, but must be explicitly granted the privileges needed to CREATE USER/MODIFY USER or CREATE DATABASE (see also "GRANT (SQL Form)" on page 2-22).

The *account_ID* can include an optional priority-level prefix. User priorities are used to maintain efficient operation of Teradata while providing equitable service to multiple client systems and users. Any user who has the CREATE/MODIFY USER privilege can assign a priority; therefore, the Teradata administrator should monitor priority assignments closely.

Maximum Number of Databases and Users

The combined total number of databases and users for any one system can be up to 4.2 billion.

See also "CREATE DATABASE" on page 1-80.

Collation

The COLLATION option defines the name for a collation that determines the ordering of data characters during comparison operations and when sorting data in response to a SELECT query that includes an ORDER BY or WITH...BY clause.

If you do not provide a COLLATION name, then the default is HOST.

During any session, you can override the COLLATION attribute by executing a SET SESSION COLLATION statement.

MULTINATIONAL Collation

The MULTINATIONAL keyword defines the name of the default collation sequence for the user to be one of the International sort orders (see "ORDER BY Clause" in the chapter "The SELECT Statement," in *Teradata RDBMS SQL Reference, Volume 6*.

The MULTINATIONAL parameter determines that the International sort order is to be used. This parameter returns an error unless the hashing algorithm is set to recognize diacritical characters.

On International/European sites MULTINATIONAL collation is two level.

On Japanese language sites, collation is one level and is always available.

Account IDs Can Be Specified During Logon

You can specify an *account_ID* at logon time by including it in the logon string. The *account_ID* specified at logon must be defined for the user.

IF no account ID is specified ...	THEN the default ID used is ...
in the logon string	the first account ID found in the most recent CREATE USER or MODIFY user statement for the user.
by a CREATE USER or MODIFY USER statement	that of the immediate owner of the database for the user.

CPU and I/O resources expended by a session are charged to the account used at logon time.

Reporting Space Used by User

All space (permanent, temporary, and spool) used is reported in the standard Teradata views under either the account ID specified at logon or the default account ID if none is specified at logon.

PERM, SPOOL, and TEMP Space

If necessary, the defined PERM, SPOOL, or TEMP space is changed to the highest multiple of the number of AMPs on the system less than or equal to the requested space.

If no SPOOL space is defined, then the space assigned is the largest value that meets both of the following criteria:

- A multiple of the number of AMPs on the system
- Not greater than the spool space of the owner

If no TEMP space is defined, then the space assigned is the largest value that meets both of the following criteria:

- A multiple of the number of AMPs on the system
- Not greater than the temporary space of the owner

USING Modifier in STARTUP String

The USING modifier is not supported in a STARTUP string. If it is specified, no error is indicated when the CREATE USER statement is processed, and the USING clause is stored. However, when the string is invoked, a message is returned that a “Data or IndicData parcel” is not found or is not allowed.

A user/database can contain only one journal table.

Local Journaling

The LOCAL single AFTER image journal is supported analogously to single BEFORE image journal:

- The access privileges required to create or drop LOCAL single AFTER image journal are the same as for the analogous operation on single BEFORE image journal.
- LOCAL single AFTER image journaling is restricted to non-fallback data tables.
- Archive/Recovery rules for LOCAL single AFTER image journal and 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.

Creating a User With a Null Password

You cannot create a user definition with a null password. The following procedure explains how you can permit an existing user to log on with a null password.

Step	Action

1	Modify user rhh using the following CREATE USER statement.
	`CREATE USER rhh AS`
	`PERM = 10000000,`
	`PASSWORD = secret;`
	This step is not necessary if the user definition already exists.

2	Grant user rhh the following logon privileges using the GRANT LOGON statement.
	`GRANT LOGON ON ALL`
	`TO rhh`
	`WITH NULL PASSWORD;`

3	Install an appropriate security user exit on TDP for the user. See the chapter "TDP Security Features" in *Teradata TDP Reference* for instructions on how to do this.

Creating a User for Single Sign-On

Single sign-on is available only in the Windows environment. This feature permits a user to log onto a computer one time and, for the duration of that logon, to be able to access a Teradata database without providing a user name, password, or account name. You do this by granting the user logon privileges with a null password.

For details about the single sign-on feature, see *Teradata RDBMS Utilities*.

The following procedure creates a Teradata user who can log onto Teradata through a gateway that does not have the Append Domain Name option set using the Gtwcontrol utility. This user is already defined to Windows as user rhh.

Step	Action

1	Create user rhh using the following CREATE USER statement.
	`CREATE USER rhh AS`
	`PERM = 10000000,`
	`PASSWORD = secret;`

2	Grant user rhh the following logon privileges using the GRANT LOGON statement.
	`GRANT LOGON ON ALL`
	`TO rhh`
	`WITH NULL PASSWORD;`

The following procedure creates a Teradata user who can log onto Teradata through a gateway that has Append Domain Name set. This user is already defined to Windows as user rhh and her account is in the esw2kdev domain.

Step	Action

1	Create user rhh using the following CREATE USER statement.
	`CREATE USER "rhh@esw2kdev" AS`
	`PERM = 10000000,`
	`PASSWORD = secret;`

2	Grant user rhh the following logon privileges using the GRANT LOGON statement.
	`GRANT LOGON ON ALL`
	`TO "rhh@esw2kdev"`
	`WITH NULL PASSWORD;`

Specifying a Default Role for a User

Roles define access privileges on database objects. Specifying a default role for a user gives the user access to all the objects that the role has privileges to. If the default role has members that are roles, the user also has access to all the objects that the nested roles have privileges to.

To submit a CREATE USER statement that specifies a default role, you must be one of the following:

- A user who is a member of the specified role WITH ADMIN OPTION
- A user whose current role is a member of the specified role WITH ADMIN OPTION
- A user whose current role has a member that is also a member of the specified role WITH ADMIN OPTION

The newly-created user automatically becomes a member of the default role.

The default role for a user automatically becomes the current role for the session when the user first logs on. The default role for a user automatically becomes the active role for the session when the user first logs on.

Only one role can be current in a session. More than role can be active, however. The active roles for a user include the current role and all of its members that are roles. Privileges validation uses the active roles for a user.

FOR more information on ...	SEE ...
creating roles	"CREATE ROLE" on page 1-180.
adding members to roles	"GRANT (Role Form)" on page 2-19.
changing the current role for a session	"SET ROLE" on page 1-382.

Assigning a Profile to a User

Profiles define system parameters and user-level password security attributes. Assigning a profile to a group of users makes sure that all group members operate with a common set of parameters.

To submit a CREATE USER statement that assigns a profile, you must have the DROP PROFILE privilege.

When a user who has both profile and user settings logs on, Teradata sets the current settings to the profile settings instead of the user settings.

After logging on, you can use the SET SESSION statement to override the default database or account settings for a session. To modify spool and temp space settings for a session, use the MODIFY PROFILE statement.

FOR more information on ...	SEE ...
creating profiles	"CREATE PROFILE" on page 1-173.
changing system parameters for a profile	"MODIFY PROFILE" on page 1-359.
obtaining the DROP PROFILE privilege	"GRANT (SQL Form)" on page 2-22.

Example

Assume that Teradata has just been installed. The system administrator has logged on as user DBC, and has submitted a CREATE USER statement to create the administrative user called Sysadmin. As user Sysadmin, the administrator submits CREATE DATABASE statements to create such required databases as Finance, Executive, Engineering, Manufacturing, and Marketing.

The administrator adds a new user to the finance database via the CREATE USER statement, as follows:

```
CREATE USER                   marks
   FROM                       finance
   AS
   PERMANENT =                1000000
   ,PASSWORD =                finan1
   ,SPOOL =                   1200000
   ,FALLBACK PROTECTION
   ,DUAL AFTER JOURNAL
   ,DEFAULT JOURNAL TABLE =   finance.journals
   ,DEFAULT DATABASE =        finance
   ,STARTUP =                'EXEC setpf;'
   ,ACCOUNT =                '$mfinance','$hfinance';
```

After user Marks is created, Sysadmin submits the following statement to grant Finance privileges with the WITH GRANT OPTION to Marks:

```
GRANT ALL ON finance
TO marks
WITH GRANT OPTION;
```

CREATE VIEW
REPLACE VIEW

Purpose

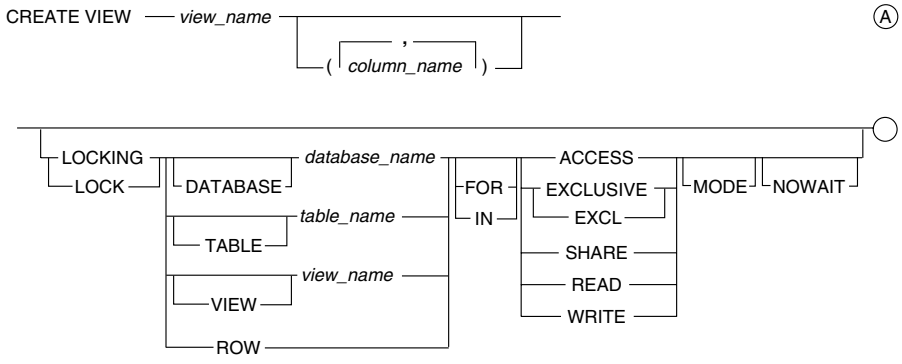
CREATE VIEW defines a view on a set of tables.

REPLACE VIEW redefines an existing view or, if the specified view does not exist, creates a new view with the specified name.

Invocation

Executable.

Syntax



1101S077

where:

Syntax Element ...	Specifies ...
view_name	the name of the new view. If *view_name* is not fully qualified, the default database is used.
column_name	the name of a view column. If more than one column is specified, list the their names in the order in which each column is to be displayed for the view.
AS	an introduction to the view definition.
LOCKING Clause	
LOCKING	the type of lock to be placed on a database, table, view, or row. This setting will override any default usage lock placed on that object by Teradata (see "Locks" on page 1-319 and "LOCKING Modifier" in Chapter 3: "SQL Data Manipulation Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 6*).
DATABASE TABLE VIEW	the name of a database, table, or view that is to be locked
ROW	a row or rows to be locked in accordance with the defining SQL statement.
database_name	the name of the database (or user) to be locked.
table_name	the name of the table to be locked. You can create views that reference both global temporary tables and volatile tables.
view_name	the name of the view to be locked.
FOR IN	the type of lock to be placed.
ACCESS READ SHARE WRITE EXCLUSIVE	the type of lock to be placed. The SHARE keyword can be used as a synonym for READ.
MODE	an optional keyword.
NOWAIT	that if the indicated lock cannot be obtained, the statement should be aborted. Used for situations where it is not desirable to have a statement wait for resources, possibly also tying up resources in the process of waiting.

Syntax Element …	Specifies …
SELECT Clause	
SELECT	that the *expression* to be associated with the *column_name* list are to select from one or more existing tables or views.
DISTINCT	that only one row is to be returned from any set of duplicates that might result from a given *expression* list. Two rows are considered duplicates only if each value in one is equal to the corresponding value in the other.
ALL	that all rows, including duplicates, are to be returned in the results of the expression list. This is the default value.
expression	an expression. Aggregate and arithmetic operators can be used with expressions.
*	that all columns of all tables referenced in the FROM clause be returned. When qualified by *table_name*, specifies that all columns of only *table_name* are to be returned. View columns are explicitly enumerated when views are defined; therefore, if a table is changed after a view is defined, those changes will not appear if the SELECT * construct is used.
table_name	the name of a table or view. The *table_name*.* in the SELECT list can define the table from which rows are to be returned when two or more tables are referenced in the FROM clause.
correlation_name	a temporary name for the table that is referenced by *table_name*. The alias *correlation_name* is used to name expressions and must be used during a self-join operation on the table. Correlation names are also referred to as range variables.
AS	an introduction to *correlation_name*.
FROM Clause	
FROM	the names of one or more tables, views, or derived tables from which *expression* is to be derived. See "FROM Clause" in the chapter "The SELECT Statement" in *Teradata RDBMS SQL Reference, Volume 6* for information about the FROM clause.
joined_table	the name of a joined table.

Syntax Element …	Specifies …
INNER	a join in which qualifying rows from one table are combined with qualifying rows from another table according to some join condition. Types of inner joins include exclusion, merge, nested, product, and row ID. This is the default join type.
OUTER	a join in which qualifying rows from one table that do not have matches in the other table, are included in the join result. The rows from the first table are extended with nulls.
LEFT OUTER	the table that was listed first in the FROM clause. In a LEFT OUTER JOIN, the rows from the left table that are not returned in the result of the inner join of the two tables are returned in the outer join result, and extended with nulls.
RIGHT OUTER	the table that was listed second in the FROM clause. In a RIGHT OUTER JOIN, the rows from the right table that are not returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.
FULL OUTER	a join that returns rows from both tables. In a FULL OUTER JOIN, rows from both tables that have not been returned in the result of the inner join are returned in the outer join result, and extended with nulls.
JOIN	an introduction to the name of the second table to participate in the join.
joined_table	the name of the joined table.
ON *search_condition*	one or more conditional expressions that must be satisfied by the result rows. An ON condition clause is required if the FROM clause specifies outer join syntax.
CROSS JOIN	A CROSS JOIN is an unconstrained, or Cartesian join; it returns all rows from all tables specified in the FROM clause.
single_table	the name of table participating in join.

Derived Tables

The derived table allows the FROM list to specify a spool file composed of selected data from underlying tables. The derived table acts like a viewed table.

subquery	a SELECT statement that defines the derived table.
AS	an optional introductory clause to derived table name.
derived_table_name	the name of derived table.

Syntax Element …	Specifies …
column_name	the column name. Specify only the column name here; do not use fully qualified forms such as `databasename.tablename.columnname`, or `tablename.columnname`.
	WHERE Clause
WHERE	an introduction to the search condition in the SELECT statement.
search_condition	a conditional search expression that must be satisfied by the row(s) returned by the statement.
GROUP BY	an introduction to a reference to one or more expressions in the select expression list.
column_name	specifies the names of columns used in the GROUP BY clause in the SELECT statement.
column_position	specifies the numeric position of the column(s) used in the GROUP BY clause.
expression	the expressions used in the GROUP BY clause.
	HAVING Clause
HAVING	an introduction to the conditional clause in the SELECT statement.
condition	one or more conditional expressions that must be satisfied by the result rows. Aggregate operators can be used with HAVING. HAVING *condition* selects rows from a single group defined in the select expression list that has only aggregate results, or it selects rows from the group or groups defined in a GROUP BY clause.
WITH CHECK OPTION	Restricts the rows that can be updated in the table by an INSERT or UPDATE statement. See "WITH CHECK OPTION Clause in Views" on page 1-318.

ANSI Compliance

CREATE VIEW is ANSI SQL-99-compliant.

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

Authorization: CREATE VIEW

If a user other than the owner utilizes a view, then all of the relevant privileges needed by the owner of the view to access underlying tables and views must be held by the owner of the view WITH GRANT OPTION.

To create a view, the user must have the CREATE VIEW privilege. The creator also receives the DROP VIEW, INSERT, UPDATE, DELETE, SELECT privileges on the newly created view, WITH GRANT OPTION.

To update a table via a view, you must have the update privilege on the view, and the view owner must have the update privilege on the table.

To delete rows using a view, you must have the delete privilege on the view and the view owner must have the delete privilege on the table.

If updating or deleting using a view, and a search condition exists, you must have SELECT ACCESS privileges on the view, and the view owner must have SELECT ACCESS on the objects searched.

Authorization: REPLACE VIEW

You must have the DROP VIEW privilege to perform REPLACE VIEW.

You must have the CREATE VIEW privilege if the specified view does not exist.

Limit On Request Size

The maximum size of the fully expanded text for a view is 2 Mbyte.

Because expansions of source text in view definitions are made to fully qualify object names and to normalize expressions, it is possible for a view to be defined but to be unusable because of stack overflows in the Syntaxer at execution. In this case, Teradata returns an error.

WHERE and HAVING Clauses

The WHERE clause specifies the conditional expression by which rows for the view are to be selected.

The HAVING clause selects the groups defined by a GROUP BY clause that satisfy a conditional expression.

Joins can only be defined in a HAVING clause of a view when the HAVING clause is part of a subquery.

Do not use `SELECT *` 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.

Rules for Creating, Replacing, and Using Views

To avoid unexpected or incorrect results when querying a view, consider the following rules when creating the view:

- An operation on a view that references multiple tables causes the tables 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).
- A view can be used as a modifier; that is, to insert or update data in the underlying table. A modifier view can reference only one table, and no data descriptions or aggregates can be used in the expressions that reference the table columns.
- Any user with the privilege to use a view has the privilege to override a lock specified for a view. Users of the Performance Monitor/Application Programming Interface who have the MONSESSION privilege can view all locks in the system and can determine which other user locks are blocking other users.
- 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 and volatile tables.
- Changing the DateForm 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 table or view. However, expressions in a CREATE VIEW statement that require date validation against the DateForm or default Date format can generate an error message if the DateForm 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.
- Teradata 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 a message is returned to the requestor in the following cases:
  - You perform a HELP VIEW or SHOW VIEW statement on the view.
  - You perform a DML statement that accesses tables through the view.

Support for Global Temporary and Volatile Tables

You can reference both global temporary tables and volatile tables in a view.

When you access a view that involves a global temporary or volatile table, then all references to that table are mapped to the corresponding global temporary or volatile table within the current session.

When an application performs an INSERT statement that inserts rows to a 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 statement performs.

If the INSERT 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 an error is returned to you.

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, 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 statement 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.

WHEN WITH CHECK OPTION is ...	Any insert or update made to the table through the view ...
used	only inserts or updates rows that satisfy the WHERE clause.
not used	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.

Updatable Views and WITH CHECK OPTION

The following rules apply to updatable views and WITH CHECK OPTION.

- If WITH CHECK OPTION is specified, the view should be updatable. In addition, any insert or update to the table through the view will not result in the creation of a row for which the WHERE clause is false.
- If WITH CHECK OPTION is not included in an updatable view, then any WHERE clause contained in the query defining the view is ignored for any insert or update action through the view.

- Nested Views: A user can define nested views that only reference a single base table, which might allow the views to be updatable. In this case, the presence of a WITH CHECK OPTION in a view definition causes the WHERE clause for that view, and WHERE clauses of underlying views, to be used in the constraint on INSERT or UPDATE.

See "Example 1" on page 1-321 and "Example 6" on page 1-323.

Operators

When defining a view that is not a modifier, one or more aggregate operators (SUM, AVG, MAX, MIN, and COUNT) can be used in the SELECT expression list or in the conditional expression of a HAVING clause. They cannot be used in the conditional expression of a WHERE clause.

SELECT *expression* Can Include Data Definitions

When defining a view that is not a modifier, any expression in the SELECT expression list can include a data definition (Data Type, FORMAT, TITLE, etc.). Data definitions determine the form of the view display.

For example, a TITLE phrase in the CREATE statement of the underlying table or view will not be displayed unless the TITLE phrase is also included in the CREATE VIEW statement.

The data definition for a view column can differ from that defined for the associated column of the underlying table or view. However, not all data definitions are relevant to view expressions.

For example, range constraints and DEFAULT declarations cannot be used in the expression list. The WHERE clause, or the HAVING clause if GROUP BY is specified, should be used to define range constraints in a view.

Locks

When a view is defined with a LOCKING modifier, the specified lock is placed on the underlying table set each time the view is referenced in a 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 ad hoc users and users who will modify the data, concurrent access. Also see "LOCKING Modifier" in Chapter 3: "SQL Data Manipulation Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 6*.

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 a user modifying the data, attempts to modify the data. The READ lock placed by the ad hoc user will prevent the user modifying the data, 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, the ad hoc user can not prevent the user modifying the data from completing 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.

View Definitions and Character Data Types

A view can contain explicit [VAR]CHARACTER(n) [CHARACTER SET …] clauses. If the character data type (for example, CHARACTER SET LATIN) is not specified, the view definition expands the [VAR]CHARACTER(n) clause.

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 character data type of the expression if the expression is of type CHARACTER.
- Otherwise, the character data type is set to the user default character type of the creator/modifier of the view (CREATE USER default character type).

  The CREATE USER default character type of the view is used because otherwise the user default character type would depend on the user, and thus two users with different user default character type could get different results from the view.

  For example, if the CREATE USER default character type is Kanji1, then the view is expanded as shown in the immediately following example.

  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 in the immediately following example.

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);
```

What REPLACE VIEW Does

The syntax for REPLACE VIEW is identical to that of the CREATE VIEW statement except that the keyword REPLACE is substituted for the keyword CREATE.

REPLACE VIEW operates as a DROP followed by a CREATE, except for the handling of access privileges.

All access privileges that were granted directly on the original view are retained.

If the specified view or macro does not exist, the REPLACE VIEW statement creates the view or macro. In this case, the REPLACE statement has the same effect as a CREATE VIEW statement.

If an error occurs during the replacement of the view, the existing view remains in place as it was prior to the performance of the REPLACE VIEW statement (it is not dropped). This is analogous to a ROLLBACK on the operation.

Example 1

The following statement creates a view of the Employee table so that it provides access only to the names and job titles of the employees in Department 300:

```
CREATE VIEW Dept300 (Name, JobTitle) AS
 SELECT Name, JobTitle
 FROM Employee
 WHERE DeptNo = 300
WITH CHECK OPTION;
```

The WITH CHECK OPTION prevents using this view to INSERT a row into Employee through the view, or to update any row of Employee for which DeptNo <> 300.

Example 2

The following statement creates a view of the Department table. Each column in the view is defined with a title. Therefore, the view data is displayed with column titles that differ from those defined for the Department table.

```
CREATE VIEW DEPT (DeptNo, DeptName, Loc, MgrNo) AS
 SELECT   DeptNo(TITLE 'Department Number'),
          DeptName(TITLE 'Department Name'),
          Loc (TITLE 'Department Location'),
          MgrNo(TITLE 'Manager Number')
 FROM Department;
```

Example 3

The following statement creates a view that allows a personnel executive to keep track of employees who have more experience on the job than their supervisors. This statement defines a self-join of the Employee table. The temporary names "Workers" and "Managers", created in the FROM clause, see the two temporary tables participating in the self-join.

```
CREATE VIEW EmpInfo (WorkerName,WorkerYrsExp,Department,
            ManagerName,ManagerYrsExp) AS
 SELECT Workers.Name, Workers.YrsExp, Workers.DeptNo,
        Managers.Name, Managers.YrsExp
 FROM Employee Workers, Employee Managers
 WHERE Workers.DeptNo = Managers.DeptNo
 AND Managers.JobTitle IN ('Manager', 'Vice Pres')
 AND Workers.YrsExp > Managers.YrsExp;
```

Example 4

The following statement illustrates the use of aggregates in a view definition. The result rows are grouped by department number and include only those rows with an average salary of $35,000 or higher.

```
CREATE VIEW DeptSal (DeptNo, MinSal, MaxSal, AvgSal) AS
 SELECT DeptNo, MIN(Salary), MAX(Salary), AVG(Salary)
 FROM Employee
 GROUP BY DeptNo
 HAVING AVG(Salary) >= 35000;
```

Now perform the following SELECT statement on this view.

```
SELECT * FROM DeptSal;
```

The query returns the following response set.

```
DeptNo          MinSal          MaxSal          AvgSal
------      ----------      ---------      ----------
   600       28,600.00       45,000.00       36,650.00
   300       23,000.00       65,000.00       47,666.67
   700       30,000.00       45,000.00       37,666.67
   500       22,000.00       56,000.00       38,285.71
```

Example 5

The following SQL request returns the response set that follows.

```
SELECT DeptNo, MinSal, MinSal+10000, AvgSal
FROM DeptSal
WHERE AvgSal > (MinSal + 10000);
```

```
DeptNo       MinSal          (MinSal+10000)          AvgSal
------      ----------    -----------------      ----------
   500       22,000.00             32000.00       38,285.71
   300       23,000.00             33000.00       47,666.67
```

Example 6

This example shows how use of a view name in an UPDATE, INSERT, or DELETE statement allows you to add, change, or remove data in the table (or tables) upon which the view is based. For example, updating data via a view changes data in the underlying table. Inserting or deleting rows via a view adds or removes rows from the underlying table.

Consider the following Staff_Info view, which gives a personnel clerk retrieval access to employee numbers, names, job titles, department numbers, sex, and dates of birth for all employees except vice presidents and managers:

```
CREATE VIEW staff_info (number, name, position, department,
                        sex, dob) AS
 SELECT employee.empno, name, jobtitle, deptno, sex, dob
 FROM employee
 WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
WITH CHECK OPTION;
```

If the owner of Staff_Info has the insert privilege on the Employee table, and if the clerk has the insert privilege on Staff_Info, then the clerk can use this view to add new rows to Employee. For example, performing the following INSERT statement inserts a row in the underlying Employee table that contains the specified information.

```
INSERT INTO staff_info (number, name, position, department,
sex, dob)
 VALUES (10024, 'Crowell N', 'Secretary', 200, 'F', 'Jun 03
1960');
```

The constraint on Staff_Info illustrated by the following WHERE clause applies to any insert using this view that includes the WITH CHECK OPTION phrase.

```
…
WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
…
```

Therefore, the preceding INSERT statement would fail if the Position entered for Crowell was Vice Pres or Manager.

If this view were defined to not include the WITH CHECK OPTION, and a user had UPDATE privilege, that user could update a job title to 'Vice Pres' or 'Manager'. The user would be unable to access the changed row through the view.

Entering the following statement changes the department number (from 200 to 300) entered for Crowell in the preceding INSERT statement.

```
UPDATE staff_info
 SET department = 300
 WHERE number = 10024;
```

Performing the following DELETE statement removes the row for employee Crowell from the staff_info table.

```
DELETE FROM staff_info
 WHERE number = 10024;
```

A view is a useful medium for allowing other users access to table data. However, as the preceding examples suggest, granting another user insert, update, and delete privileges on a view means relinquishing some control over your data. Carefully consider granting such privileges.

The default is *not* to constrain updated or inserted values unless the view definition explicitly includes WITH CHECK OPTION.

Example 7: REPLACE VIEW

To change the department name column in the employee_info view to a department number column, type:

```
REPLACE VIEW employee_info (Number, Name, Position,
                            Department)
 AS SELECT employee.empno, name, jobtitle, deptno
    FROM emp_info
    WHERE jobtitle NOT IN ('vice pres', 'manager');
```

You must have the drop privilege on a view to replace it.

If you enter a REPLACE VIEW statement for a view that does not exist, Teradata creates the view following the specifications in the REPLACE statement.

DATABASE

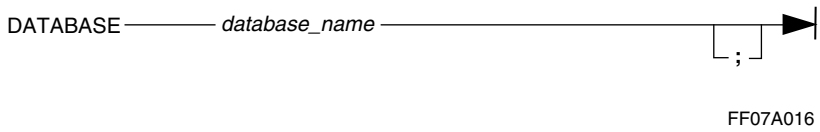
Purpose

Establishes a new default database for the current session for SQL statements that do not require fully-qualified table, view, or macro names.

Invocation

Executable.

Syntax



FF07A016

where:

Syntax Element ...	Specifies ...
database_name	the name of the default database.

ANSI Compliance

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

Authorization

No privileges are required to use DATABASE. However privileges will be checked subsequently when an object is accessed.

Logging Sessions On and Creating a Default Database

When sessions are logged on, an initial default database is set. Normally, this is the database of the user.

You can also use the CREATE PROFILE (see "CREATE PROFILE" on page 1-173) and MODIFY PROFILE (see "MODIFY PROFILE" on page 1-359) 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. The DATABASE statement can be included in the startup string for logging on. See also "CREATE TRIGGER REPLACE TRIGGER" on page 1-276.

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 more than one database, unqualified names can apply to only one of those databases.

When encountering an unqualified table name, Teradata looks for a table 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 "ambiguous table name" error message results 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). It is therefore recommended that object references in a macro data definition statement be fully qualified in the macro body (see "CREATE MACRO REPLACE MACRO" on page 1-142).

Example

The following statement establishes Personnel as the default database for the current session:

```
DATABASE Personnel;
```

DELETE DATABASE
DELETE USER

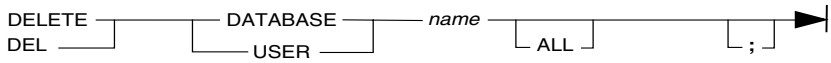
Purpose

Deletes all data tables, views, triggers, stored procedures, and macros from a database or user.

Invocation

Executable.

Syntax



FF07D026

where:

Syntax Element …	Specifies …
name	the name of the database or user from which all tables, views, triggers, stored procedures, and macros are to be deleted.
ALL	that all objects, including the materialized temporary tables within the target database, are to be dropped. If you do not specify ALL and there are materialized global temporary tables in the database, the request aborts.

ANSI Compliance

DELETE DATABASE and DELETE USER are Teradata extensions to the ANSI SQL-99 standard.

Authorization

You must have the DROP privilege on the referenced database/user.

DELETE USER can delete a database, and DELETE DATABASE can delete a user.

Access Privileges Are Removed With DELETE

The DELETE DATABASE/USER statement also removes all access privileges to the objects that were deleted. Because the Dump and Restore utility does not automatically recover these access privileges, they must be reestablished following the restore operation.

Locks

When a DELETE DATABASE or DELETE USER statement is performed, an EXCLUSIVE lock is placed on the database or user from which objects are being deleted.

Referenced Objects

You cannot delete a database in which a hash or join index is defined without first dropping the index.

You cannot delete a database that contains a table on which a hash or join index is defined without first dropping the index.

DELETE DATABASE/USER Does Not Delete Journal Tables

The DELETE DATABASE/USER statement does not delete a journal table. You must enter a MODIFY DATABASE/MODIFY USER statement to remove a journal table.

DELETE DATABASE/USER and Temporary and Volatile Tables

Rules for temporary and volatile tables are as follows:

<table>
<tr><th>IF a table is this type ...</th><th>THEN the following things occur when you delete a database or user ...</th></tr>
<tr><td>Global Temporary</td><td>
<table>
<tr><th>IF ...</th><th>THEN ...</th></tr>
<tr><td>you specify the ALL keyword</td><td>all objects, including the materialized temporary tables within the target database are dropped.</td></tr>
<tr><td>you do not specify the ALL keyword</td><td>the statement aborts if there are any materialized temporary tables; else all objects in the target database are dropped.</td></tr>
</table>
</td></tr>
<tr><td>Volatile</td><td>All objects in the target database/user are dropped except volatile tables.</td></tr>
</table>

After DELETE Structure Remains Intact

After all the data tables, views, triggers, stored procedures, and macros have been deleted from a database/user, the structure remains intact (as though its CREATE statement had just been performed) and can be used in a subsequent restore operation. The DELETE DATABASE/USER statement is used when a database or user has been only partially restored and is not in a usable state.

Orphaned Triggers

If a trigger has been defined in another database against a table in the database being deleted, then Teradata deletes the subject table for that trigger, but not the trigger itself.

Such an orphaned trigger cannot be used again even if the subject table is recreated; it must be dropped in its home database.

Example

The following statement can be used to delete all tables, views, triggers, stored procedures, and macros from the UsedCars database:

```
DELETE DATABASE UsedCars;
```

DROP DATABASE

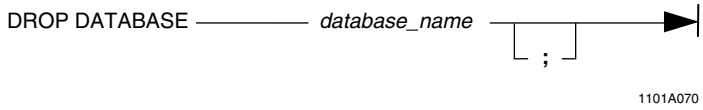
Purpose

Drops the definition for an empty database from the data dictionary. All objects associated with that database must have already been dropped.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
database_name	the name of the database that is to be dropped.

ANSI Compliance

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

DROP DATABASE is functionally equivalent to the ANSI SQL-99 statement DROP SCHEMA.

Authorization

You must have the DROP DATABASE privilege on the database. The database that is being dropped cannot own other databases or users, and must be at the bottom of the database hierarchy.

Journal Tables

If a database contains a journal table in its disk space, then the database cannot be dropped until the journal table is removed from the system. A MODIFY DATABASE statement must be used to drop the journal table.

DROP Functions

The drop operation performs the following functions:

- Sets an exclusive lock on the database.
- Adds the disk space that the drop makes available to that of the immediate owner database or user.

After a database is dropped, it cannot be recovered by using the Dump and Restore utility unless the database is restored.

Referenced Objects

You cannot drop a database in which a hash or join index is defined without first dropping the index.

You cannot drop a database that contains a table on which a hash or join index is defined without first dropping the index.

Example

The following statement drops database Accounting:

```
DROP DATABASE accounting;
```

DROP HASH INDEX

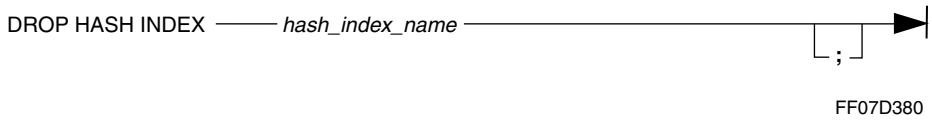
Purpose

Drops a hash index definition from the data dictionary.

Invocation

Executable.

Syntax



FF07D380

where:

Syntax element …	Specifies …
hash_index_name	the name of the hash index to be dropped.

ANSI Compliance

DROP HASH INDEX is an extension to the ANSI SQL-99 standard.

Authorization

You must have the DROP TABLE privilege on the hash index to be dropped.

Example

This statement drops the definition for the hash index named OrdHIdx_1 from the data dictionary.

```
DROP HASH INDEX OrdHIdx_1;
```

DROP INDEX

Purpose

Drops a secondary index on a table.

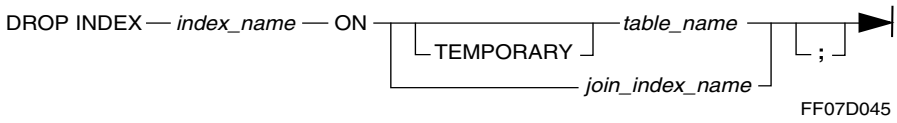
Invocation

Executable.

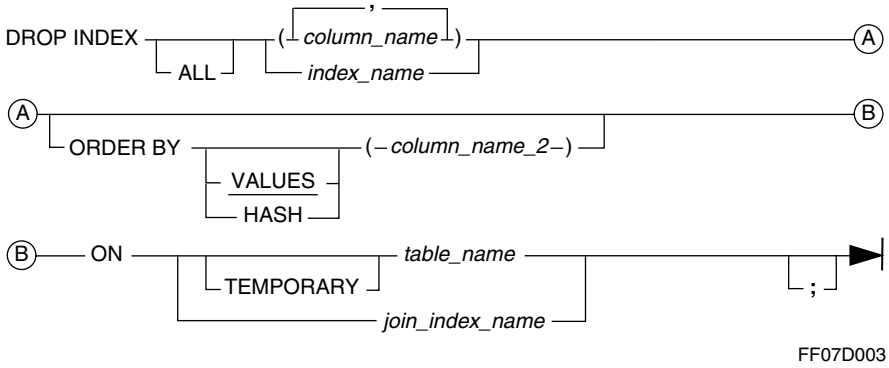
Syntax

There are two forms of the DROP INDEX statement.

DROP *index_name* Syntax



DROP *index_definition* Syntax



where:

Syntax Element …	Specifies …
ALL	that ALL was used to create the index.
column_name	a list of columns defined for the index. Limited to 64 columns.
index_name	the name of the index to be dropped.

Syntax Element …	Specifies …
ORDER BY	that index rows are ordered on each AMP by a single value- or hash-ordered column. If ORDER BY is specified, then VALUES, HASH, *column_name_2*, or a combination of VALUES or HASH with *column_name_2* must be specified.
VALUES	value-ordering for the ORDER BY column. If neither VALUES nor HASH is specified, VALUES is used as the default.
HASH	hash-ordering for the ORDER BY column.
column_name_2	a single column from the column list defined by *column_name_1* that specifies the sort order to be used for either HASH or VALUES. The size of the column is limited to four or fewer bytes. Supported numeric date types for column_name_2 are the following: • DATE • INTEGER • SMALLINT • BYTEINT • DECIMAL If you omit *column_name_2*, the first column in the *column_name* list is chosen by default.
TEMPORARY	that you are dropping an index on a materialized temporary table. You cannot specify this keyword when dropping an index on a permanent table. If you do not specify TEMPORARY, and the table from which the index is to be dropped is a global temporary table, then the statement is applied to the base global temporary table. There must not be any materialized instances of the named table or you cannot drop the index on the base temporary table.
table_name	table on which the secondary index is to be dropped.
join_index_name	the name of the join index from which this index is to be dropped.

ANSI Compliance

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

Authorization

You must have the INDEX or DROP TABLE privilege on the table.

To drop an index from a join index, you must have one of the following privileges:

- DROP TABLE on the join index
- DROP TABLE on one of the base tables
- INDEX on the join index
- INDEX on one of the base tables

Why Drop an Index?

A secondary index is normally dropped after usage patterns for a table have changed and the index is no longer used.

Another reason to DROP a UNIQUE SECONDARY INDEX is so that a MultiLoad can be performed on the table; after the MultiLoad completes, you can recreate the index.

How a DROP Functions

The drop operation performs the following functions.

Stage	Process
1	Sets an exclusive lock on the table that contains the index.
2	Frees the disk space that was used by the index subtable.

A named index is dropped either by specifying its name or its definition (but not both).

Dropping Indexes on Global Temporary and Volatile Tables

You can drop indexes on a materialized temporary table if you use the keyword TEMPORARY. If you specify TEMPORARY and the table is permanent, then Teradata returns an error.

You can neither create nor drop indexes for a volatile table.

UNIQUE Constraints

If an index has been created in the form of an unnamed UNIQUE constraint, it can only be removed by a DROP INDEX statement referencing the column or columns defined for the constraint.

Examples

The following examples illustrate the use of DROP INDEX:

Example 1

The following statement drops a secondary index that was defined on the Name column of the Employee table:

```
DROP INDEX (Name) ON Employee;
```

Example 2: Dropping Named Indexes

This statement drops the existing named index *index_1* from table *table_5*.

```
DROP INDEX index_1 ON table_5;
```

Example 3

This statement drops the existing index on columns *field_1* and *field_2* from table *table_6*.

```
DROP INDEX (field_1, field_2) ON table_6;
```

Example 4: Non-Valid

The following example demonstrates a non-valid use of dropping an index.

```
DROP INDEX index_1 (field_1, field_2) ON table_8;
```

DROP JOIN INDEX

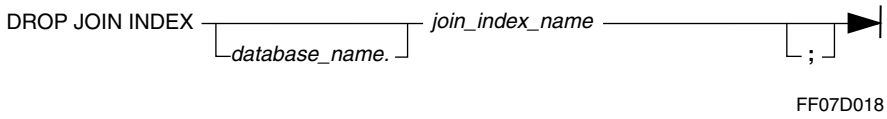
Purpose

Drops the definition for a join index from the data dictionary.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
database_name	the name of the database that contains the join index.
join_index_name	the name of the join index to be dropped.

ANSI Compliance

DROP JOIN INDEX is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have either of the following sets of privileges to drop a join index:

- DROP TABLE on the join index
- INDEX or DROP TABLE on at least one of the base tables

DROP MACRO
DROP PROCEDURE
DROP TABLE
DROP TRIGGER
DROP VIEW

Purpose

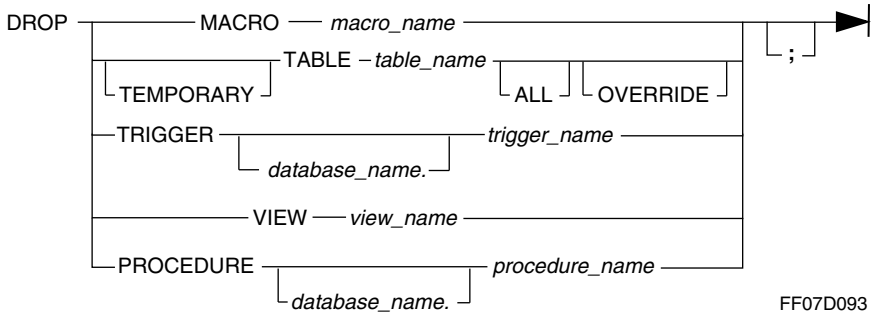
Drops the following database objects from a database, depending on the keyword specified.

Keyword Specified	Database Object Dropped
MACRO	macro definition.
PROCEDURE	stored procedure definition.
TABLE	base table definition and all table rows.
TEMPORARY TABLE	global temporary table definition.
TRIGGER	trigger definition from its subject table.
VIEW	view definition.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
macro_name	the name of the macro to be dropped.
TEMPORARY	that the table to be dropped is the local materialized instance. If you do not specify TEMPORARY and you are dropping a global temporary table, then the base global temporary definition table is dropped. If any materialized instances of the base table exist anywhere in the system, you cannot drop the table and an error is returned. Not valid with the ALL keyword. Not valid when specified for a permanent table.
table_name	the name of the table or temporary table to be dropped. To drop a base temporary table, you must have DROP TABLE privilege. To drop a local materialized temporary table, you need no privileges. If the table to be dropped has triggers associated with it, you must first drop those triggers, else the statement aborts.
ALL	that the base temporary table definition and all its materialized instances are to be dropped when they are not being accessed by any other transactions. If you do not specify ALL and if the target table is materialized in any other session, the statement aborts. Not valid with the TEMPORARY keyword. Not valid when specified for a permanent table.
OVERRIDE	to drop the table even if its table header is missing or if its version is out of sync with its dictionary definitions in tables such as DBC.TVM or DBC.TVFields.
database_name	the name of a database in which the trigger, view, or procedure to be dropped exists.
trigger_name	the name of the trigger to drop.
view_name	the name of the view to drop.
procedure_name	the name of the stored procedure to drop.

ANSI Compliance

DROP PROCEDURE, DROP TABLE, DROP TRIGGER and DROP VIEW are ANSI SQL-99-compliant.

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

Authorization

You must have the DROP privilege on the referenced table, view, trigger, stored procedure, or macro except for volatile tables, which do not require any privileges.

DROP Functions

The drop operation performs the following functions.

Stage	Process
1	Sets an exclusive lock on the table, trigger, view, stored procedure, or macro.
2	Frees the disk space used by the dropped table, any secondary index subtables, and any fallback copy for use by the owner database.
3	Removes any explicit access privileges on the object.

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 hash or join index is defined. First drop the index, then drop the table definition.

OVERRIDE Option

Use this option only when you have knowingly restored a table and its dictionary definition incorrectly.

Teradata follows numerous safeguards to ensure against data corruption, and a discrepancy between the version record for a table and its dictionary definition is a sure sign of a data integrity problem that requires immediate attention. Because of this, you cannot drop a table if it has returned a 2641 error unless you specify the OVERRIDE option.

Example

The following statement drops a table named Profile.

```
DROP TABLE Profile;
```

DROP PROFILE

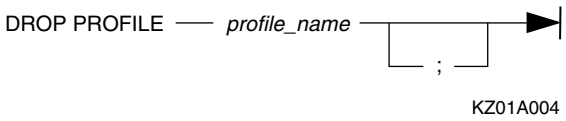
Purpose

DROP PROFILE drops the specified profile.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
profile_name	the name of the profile to drop.

ANSI Compliance

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

Authorization

The following users can drop a profile:

- User DBC
- A user who has the DROP PROFILE privilege

The profile creator does not have the implicit privilege to drop a profile. A profile creator who does not have the DROP PROFILE privilege cannot drop the profile.

Effects of a Profile Drop

When a profile is dropped, users who have the profile assigned to them continue to have that profile assigned to them; Teradata does *not* reset the profile for the affected users to NULL. Affected users receive no warnings or errors the next time they log on.

The effects of re-creating a profile with the same name as the dropped profile are not immediate. The parameter settings in the re-created profile take effect the next time that users who are assigned the profile log on.

Effects of a Profile Drop on User Parameters

The rule that determines when changes to user parameters in a MODIFY USER statement take effect also applies to a DROP PROFILE statement.

DROP PROFILE has the following effects on sessions logged on with the profile:

- Spool and temporary space settings immediately change to the settings defined for the affected users.
- Account and database settings change to the settings defined for the affected users the next time the users log on or explicitly change the settings.

  However, changes to the list of valid accounts take effect immediately. Users can only explicitly change to an account in the list of accounts available to them.

Effects of a Profile Drop on Password Attributes

DROP PROFILE does not immediately affect the password attributes of affected users. Rather, Teradata uses the password attribute settings defined for the system the next time affected users log on.

Example

The following statement drops the profile called *finance*:

```
DROP PROFILE finance;
```

For More Information

FOR more information on …	SEE …
assigning profiles to users	• "CREATE USER" on page 1-296. • "MODIFY USER" on page 1-364.
creating profiles	"CREATE PROFILE" on page 1-173.
obtaining the DROP PROFILE privilege	"GRANT (SQL Form)" on page 2-22.
system-level password attributes	"Controlling Access to Teradata RDBMS" in *Teradata RDBMS Security Administration*.

DROP ROLE

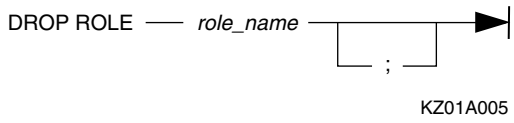
Purpose

DROP ROLE drops the specified role.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
role_name	the name of the role to drop.

ANSI Compliance

DROP ROLE is ANSI SQL-99-compliant.

Authorization

The following users can drop a role:

- User DBC
- A user who has the DROP ROLE privilege
- A user who is a member of the specified role WITH ADMIN OPTION
- A user whose active role[16] is a member of the specified role WITH ADMIN OPTION

The role creator does not have the implicit privilege to drop a role. A role creator who does not have the DROP ROLE privilege and who loses WITH ADMIN OPTION on a role cannot drop the role.

16 This also applies to current roles because an active role also includes the current role or one of its nested roles.

Effects of a Role Drop

A DROP ROLE statement removes the privileges on database objects that were granted to the dropped role.

The effect of a role drop is immediate. Users who are logged on with the dropped role as an active role lose the access privileges that were granted to the role.

A DROP ROLE statement only eliminates the specified role. If you no longer need roles that were members of a dropped role, you must drop them individually.

Default role settings for all users with the dropped role as their default role do *not* reset to NULL. Affected users receive no warnings or errors the next time they log on. Teradata does not use the obsolete default role for privileges validation.

Example

The following statement drops the role called *temp_admin*:

```
DROP ROLE temp_admin;
```

For More Information

FOR more information on ...	SEE ...
creating roles	"CREATE ROLE" on page 1-180.
granting privileges on database objects to roles	"GRANT (SQL Form)" on page 2-22.
adding members to roles	"GRANT (Role Form)" on page 2-19.
changing the current role for a session	"SET ROLE" on page 1-382.

DROP USER

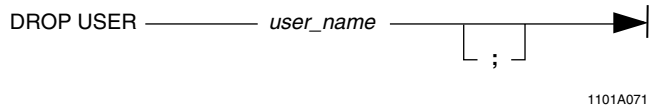
Purpose

Drops the definition for an empty user from the data dictionary. All objects associated with that user must have already been dropped.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
user_name	the name of the user that is to be dropped.

ANSI Compliance

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

DROP USER is functionally equivalent to the ANSI SQL-99 statement DROP SCHEMA.

Authorization

You must have the DROP USER privilege on the user. The user that is being dropped cannot own other databases or users, and must be at the bottom of the database hierarchy.

Journal Tables

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.

DROP Functions

The drop operation performs the following functions:

- Sets an exclusive lock on the user.
- Adds the disk 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 the user is restored.

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.

Privileges

Individual privileges granted directly to a dropped user and privileges obtained through role assignment are deleted from dictionary tables.

Dropping database objects in the user space prior to the DROP USER statement removes any privileges on the objects from roles and other users or databases.

Privileges granted by the dropped user on objects outside the user space remain in the system.

Roles and Profiles

Roles and profiles created by a dropped user remain in the system.

Example

The following statement drops user Jones:

```
DROP USER Jones;
```

END LOGGING

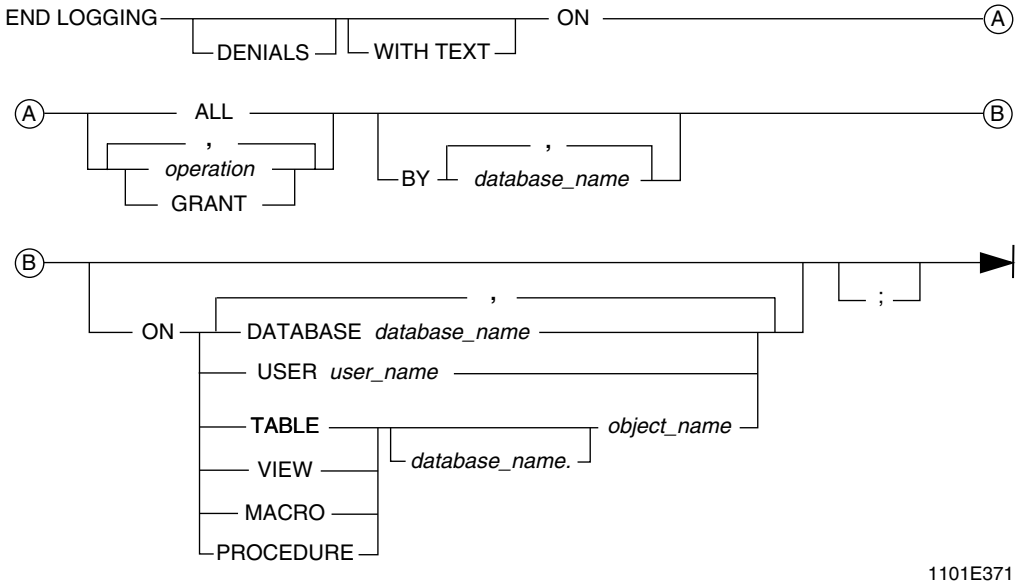
Purpose

Stops the auditing of SQL requests that attempt to access data that was started with a BEGIN LOGGING statement.

Invocation

Executable.

Syntax



where:

Syntax Element ...	Specifies ...
DENIALS	that entries should be made only when statement execution fails because the user did not have the privilege set necessary to perform the statement. DENIALS is applied to only those actions listed in the BEGIN LOGGING statement that contains it. For example, two BEGIN LOGGING statements can specify the same object, user, and action, but different frequency and DENIALS options. This allows the user to log all denials, but only the first successful use of a privilege. If this option is not specified, a log entry is potentially made if the privilege check either fails or succeeds.

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>WITH TEXT</td><td>that the text of the statement which caused the log entry is to be saved in the log.

If two BEGIN LOGGING statements specify the same user, object, and action, and one of the statements specifies the DENIALS option, then WITH TEXT can be specified on either one of the statements, on both statements, or on neither statement.

<table>
<tr><th>IF WITH TEXT is specified …</th><th>THEN the text is …</th></tr>
<tr><td>only on the statement that also specifies DENIALS</td><td>saved only when the entry is created as a result of a denial.</td></tr>
<tr><td>either without DENIALS or on both statements</td><td>always saved.</td></tr>
<tr><td>on END LOGGING</td><td>not saved for the specified actions or the DENIALS specification, but the logging frequency is not affected.</td></tr>
</table>
</td></tr>
<tr><td>ALL
operation
GRANT</td><td>the frequency with which log entries should be made.

ALL specifies that a log entry will potentially be made when any of the actions listed in the table following this one is attempted against the specified object. ALL does not include the CREATE ROLE, DROP ROLE, CREATE PROFILE, or DROP PROFILE actions, because they do not apply to a specific object.

Note that the column name and ID are displayed for each logged column when you log privileges to UPDATE and REFERENCES.

If logging has already begun for ALL actions on an object, a subsequent statement to begin or end logging for an action will change logging activity only for the specified action.

One or more action names from the following list can be used to specify the type of access to be logged. These are the same as the functions for which an access privilege can be granted.</td></tr>
</table>

Syntax Element …	Specifies …
ALL *operation* GRANT (continued)	• CHECKPOINT • CREATE DATABASE • CREATE MACRO • CREATE PROCEDURE • CREATE PROFILE • CREATE ROLE • CREATE TABLE • CREATE TRIGGER • CREATE USER • CREATE VIEW • DELETE • DROP DATABASE • DROP MACRO • DROP PROCEDURE • DROP PROFILE • DROP ROLE • DROP TABLE • DROP TRIGGER • DROP USER • DROP VIEW • DUMP • EXECUTE • EXECUTE PROCEDURE • GRANT • INDEX • INSERT • REFERENCES • RESTORE • ROLLBACK DATABASE • ROLLFORWARD DATABASE • SELECT • UPDATE
BY *database_name*	the community for which log entries should be made. If *user_name* is specified, it must be a user currently defined in Teradata; that is, a name for which space has been created and under which a session can be established. Absence of the BY *user_name* option implies *all* users (those already defined to Teradata as well as any defined in the future while this logging directive is in effect). Logging begun for specific user names cannot be ended by omitting the BY *user_name* option. If both BY *user_name* and ON *keyword object_name* are absent, then the specified action is logged for all users and objects throughout Teradata.
ON	one or more entities for which log entries are to be made. Absence of the `ON keyword object_name` option implies logging of all entities the user attempts to access. If neither `ON keyword object_name` nor `BY user_name` are specified, then the specified action is logged for *all* users and objects throughout Teradata. Role and profile access privileges do not apply to specific objects. Do not use the ON keyword object_name option if you specify a CREATE PROFILE, CREATE ROLE, DROP PROFILE, or DROP ROLE action.

Syntax Element …	Specifies …
DATABASE *database_name* USER *database_name.user_name* TABLE *database_name.table_name* VIEW *database_name.view_name* MACRO *database_name.macro_name* PROCEDURE *database_name.procedure_name*	the name of the object to be logged. The keyword DATABASE, MACRO, PROCEDURE, TABLE, USER, or VIEW must precede each object name. The object name is the name of a database, macro, stored procedure, table, user, or view. The name of a macro, stored procedure, table, or view must be qualified by the name of the database or user under which that object is defined.

ANSI Compliance

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

Authorization

None required.

Permissions

Before you can perform END LOGGING, the following two actions must be taken:

- Perform the DIPACC DBCSQL script, which is provided on the software release tape.

  This script creates the DBC.AccLogRule security macro.
- Reset Teradata to initialize the logging software.

Otherwise, access logging is not permitted.

Blank Logs Are Deleted From DBC.AccLogRuleTbl

The END LOGGING statement erases only the frequency or text flags for the specified actions and user or object. However, if erasing a frequency leaves all logging blank for a particular user, database, and table, then the row is deleted from the AccLogRuleTbl table.

Logging Must Be Enabled To Perform END LOGGING

Performing the END LOGGING statement returns an error if BEGIN LOGGING is not currently in effect for the community for which you specify logging is to be ended.

Rules for Using DDL Statements

END LOGGING statements must be submitted following the rules for DDL statement usage.

- A DDL statement cannot be combined with other statements as part of a multistatement request.
- A DDL statement can only be included in an explicit transaction (one or more requests bracketed by BEGIN/END TRANSACTION statements) if it is one of the following:
  - the only statement in the transaction, or
  - structured as a single-statement request that appears as the last request in the transaction.

When an END LOGGING statement is submitted, Teradata checks that the executing user has EXECUTE privilege on the special system macro associated with that statement. No checks are made for privileges on the user or objects identified in the LOGGING statement.

If the statement cannot be performed, an error message is returned to the requestor. Possible errors could be a non-valid action name, user name, or object name.

Access Logging Must Be Disabled To Run INITIATE INDEX ANALYSIS

You must disable access logging to perform an INITIATE INDEX ANALYSIS or RESTART INDEX ANALYSIS statement.

Once the index analysis completes, you can enable access logging once again.

Related Information

See "BEGIN LOGGING" on page 1-67.

MODIFY DATABASE

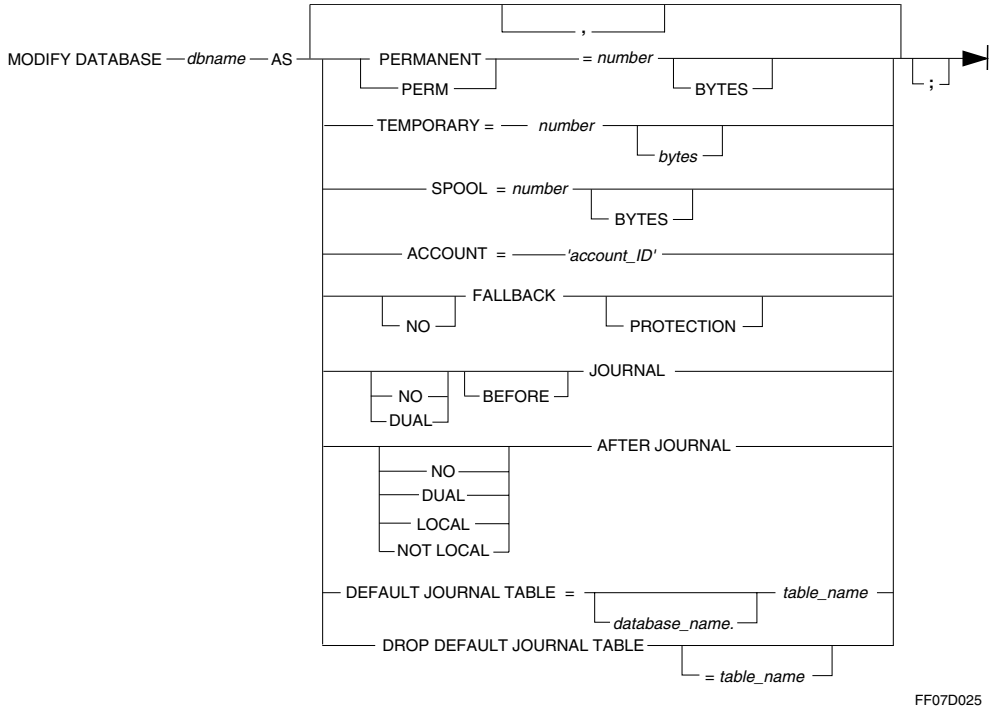
Purpose

Changes the options that were specified when a database was created.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
database_name	the name of the database to be modified.
AS	an introduction to the list of options for modifying the database.
PERMANENT	a keyword introducing the allocated fixed space.
number	a new value (n) for fixed space allocation, in bytes. *number* can be entered as an integer, a decimal value, or as a floating point value.
BYTES	an optional unit assigned to the fixed space value.

Syntax Element …	Specifies …
TEMPORARY = *number*	Permits you to define how much space (in bytes) is to be allocated by default for creating temporary tables by users within this database. Note that temporary space is reserved prior to spool space for any user defined with this characteristic. Disk usage for a materialized temporary table is charged to the temporary space allocation of the user who referenced the table. If no default temporary space is defined for a database, then the space allocated for any temporary tables created in that database is set to the maximum temporary space allocated for its immediate owner.
SPOOL	a keyword introducing the allocated spool space.
number	a new value (n) for the maximum space allocation for spool files, in bytes. *number* can be entered as an integer, a decimal value, or as a floating point value.
BYTES	an optional unit assigned to the spool space.
ACCOUNT	a keyword introducing the account identifier.
account_ID	a new identifier for the account to be charged for the space used by this database. The *account_ID can* be up to 30 characters (provided they are not Japanese characters, in which case *account_ID* can be up to 30 bytes, using any of the supported character sets), and must be enclosed by APOSTROPHE characters.
[NO] FALLBACK PROTECTION	a new default for duplicate copy protection of each data table subsequently created in the database. The current fallback setting for existing data tables remains unchanged. The FALLBACK keyword used alone implies PROTECTION.

Syntax Element …	Specifies …
NO DUAL BEFORE JOURNAL	a new default for the number of before change images to be maintained for data tables subsequently created in the database. The NO keyword terminates any current journaling default. If the JOURNAL keyword is specified without NO or DUAL, single copy journaling is implied. If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection. Existing images are not affected until the corresponding table is updated. This option can appear twice in the same statement: once to specify a BEFORE or AFTER image, and again to specify the alternate type. If only one type is specified, then the current default is modified only for that type. If both BEFORE and AFTER are specified then the two must not conflict. The JOURNAL keyword without AFTER or BEFORE indicates that both types of images are to be maintained. In this case, the current default for either or both types is modified accordingly. For example, if only AFTER JOURNAL is specified, the current default for before change images remains in effect.
NO DUAL LOCAL NOT LOCAL AFTER JOURNAL	the type of image to be maintained for the table; any existing images are not affected until the table is updated. NO, DUAL, LOCAL, or NOT LOCAL: the NO and DUAL options specify the number of after-change images to be maintained for the table. NOT LOCAL and LOCAL specify whether single after-image journal rows for non-fallback data tables are written on the same virtual AMP (LOCAL) as the changed data rows, or on another virtual AMP in the cluster (NOT LOCAL).

Syntax Element …	Specifies …
DEFAULT JOURNAL TABLE= *database_name.table_name* DROP DEFAULT JOURNAL TABLE= *table_name*	a redefinition for the current journal table or removes its status as the default for the database being modified. The DROP keyword removes the default status of the journal table currently defined as the default for the database being modified. If the journal table resides in the database being modified, DROP also deletes the table from the system. An error message is returned if the DROP request would delete a journal table that is being used by active data tables. The *table_name* parameter is required if this clause is specified without the DROP keyword. If a database is not specified, then the database being modified is assumed. If the database being modified does not have a journal table, *table_name* is created. If a different database is specified, then it must already exist and *table_name* must have been defined as its default journal table. Specifying this option does not change the status of existing data tables in the modified database.

ANSI Compliance

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

Authorization

You must have the DROP DATABASE privilege on the referenced database.

MODIFY DATABASE cannot be used to modify users. Instead, use MODIFY USER. See "MODIFY USER" on page 1-364 for more information.

When a MODIFY DATABASE statement is performed, an exclusive lock is placed on the database that is being modified.

If necessary, the defined PERM, SPOOL, or TEMPORARY space is changed to the next higher value that is a multiple of the number of AMPs on the system.

One or more options can be changed in the same MODIFY statement. A changed option applies to all subsequent operations that occur for the database.

If an existing option is not changed by a MODIFY statement, that option remains the same.

Specifying Multiple Options

Several options can be included in a single MODIFY DATABASE statement. No order is imposed, but an error is reported if options are duplicated or conflicting options are included.

Changing the Default Journal Table Residing in the Modified Database

Two modify statements are needed to change a default journal table that resides in the database being modified. The first statement is used to DROP the present journal table from the system. An error message is returned if the journal table is being used by active data tables.

After the table is dropped, a second statement is required to define a new default journal table.

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.

Two or More Databases/Users Can Share Journal Table

Two or more databases/users can share a journal table. However, if the sharing databases/users retain as their default the name of a journal table that has been dropped from the system, subsequent CREATE TABLE statements that use that default will return an error message.

Local Journaling

The LOCAL single AFTER image journal is supported analogously to single BEFORE image journal:

- Access privileges required to create or drop LOCAL single AFTER image journal are the same as for the analogous operation on single BEFORE image journal. The DROP DATABASE privilege is required.
- LOCAL single AFTER image journaling is restricted to non-fallback data tables.
- Archive/Recovery rules for LOCAL single AFTER image journal and 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.

MultiLoad, Teradata Warehouse Builder, FastLoad and Archive/Recovery are impacted by the use of LOCAL journaling.

Locks and Concurrency

Without the SPOOL, TEMP, or PERM options, MODIFY DATABASE places an ACCESS lock on the database being altered and thus runs concurrently with all DML statements. The SPOOL, TEMP, and PERM options require an EXCLUSIVE lock.

MODIFY DATABASE also places a row-hash WRITE lock on the database for DBC.DBase. The SPOOL, TEMP, and PERM options then place a second lock on the immediate parent for DBC.DBase. For PERM, the second lock is a row-hash WRITE lock. For SPOOL or TEMP, the second lock is a row-hash READ lock.

The SPOOL, TEMP, and PERM options also place a row-hash WRITE lock on DBC.DataBaseSpace. The PERM option then places a second row-hash WRITE lock on the immediate parent for DBC.DataBaseSpace.

Examples

The following examples illustrate the use of MODIFY DATABASE:

Example 1

The following statement can be used to change the permanent space allocation for the Personnel database to 6,000,000 bytes:

```
MODIFY DATABASE Personnel AS
 PERMANENT = 6000000 BYTES;
```

Example 2

Two MODIFY statements are needed to change the default journal table if it resides in the database being modified.

For example, suppose the journal table FinCopy resides in the Personnel database. To change the default journal table from FinCopy to Jrnl1, the present default journal table must be dropped by entering the statement below. This statement removes FinCopy as the default and also drops it from the system.

If any existing tables use FinCopy as their journal table, the statement returns an error message.

```
MODIFY DATABASE Personnel AS
 DROP DEFAULT JOURNAL TABLE;
```

Example 3

Now that the present default journal has been dropped, this statement can be used to define a new default journal table:

```
MODIFY DATABASE Personnel AS
 DEFAULT JOURNAL TABLE = Jrnl1;
```

Example 4

If the current journal table does *not* reside in the database being modified, then you could use the following statement to change the default journal table from FinCopy to Jrnl1:

```
MODIFY DATABASE Personnel AS
 DEFAULT JOURNAL TABLE = Jrnl1;
```

Example 5

Change fallback protection and space allocation options on the Finance database as follows.

```
MODIFY DATABASE finance AS
 PERM =                         75000000,
 SPOOL =                       150000000,
 NOFALLBACK,
 DROP DEFAULT JOURNAL TABLE = finance.journals;
```

MODIFY PROFILE

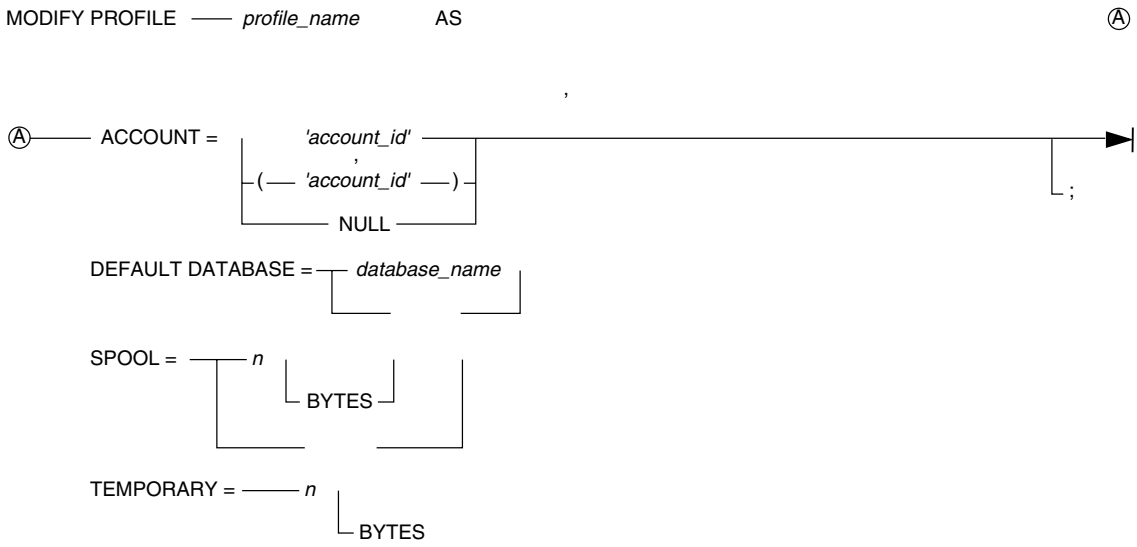
Purpose

MODIFY PROFILE changes the system parameters for a profile.

Invocation

Executable.

Syntax



KZ01a002

where:

Syntax Element …	Specifies …
profile_name	the name of the profile to modify.
ACCOUNT = *account_id*	a replacement for one or more accounts to charge for the space used by users with this profile. Each *account_ID* can be up to 30 characters and must be enclosed in APOSTROPHE characters. A list of *account_IDs* must be separated by COMMA characters and enclosed by LEFT PARENTHESIS and RIGHT PARENTHESIS characters. The first (or only) identifier becomes the default account for the user. If no account is defined in the profile assigned to a user, or if the account is set to NULL, Teradata uses the setting defined for the user. Accounts in a profile assigned to a user override any other accounts the user might have.
DEFAULT DATABASE = *database_name*	a replacement for the default database established each time a user with this profile logs onto Teradata. *database_name* need not be an existing database. Teradata returns an error when a user with this profile tries to create or reference an object within a nonexistent database. If no default database is defined in the profile assigned to a user, or if the default database is set to NULL, Teradata uses the setting defined for the user.
SPOOL = *n* BYTES	a replacement for the number of bytes to allocate to the spool file. *n* cannot exceed the spool space parameter in the profile of the modifier or, if none is specified, the user setting for the profile modifier. *n* can be an integer, a decimal value, or a floating point value. *n* refers to bytes, whether or not the optional BYTES keyword is specified. If no spool space is defined in the profile assigned to a user, or if the spool space is set to NULL, Teradata uses the setting defined for the user.

Syntax Element …	Specifies …
TEMPORARY = *n* BYTES	a replacement for how much space to allocate for creating temporary tables. *n* cannot exceed the temporary space parameter in the profile of the creator. If no temporary space is defined for that profile, then the temporary space defined for the user setting of the creator is the criterion. *n* can be an integer, a decimal value, or a floating point value. *n* refers to bytes, whether or not the optional BYTES keyword is specified. If no temporary space is defined in the profile assigned to a user, or if the temporary space is set to NULL, Teradata uses the setting defined for the user. Temporary space is reserved prior to spool space for any user defined with this characteristic. Disk usage for a materialized temporary table is charged to the temporary space allocation of the user who referenced the table.
PASSWORD [ATTRIBUTES]	a keyword allowing you to modify password security attributes. The default value is NULL for password attributes that are not defined in the profile for a user, which causes Teradata to use the system setting. Password security attributes defined in a profile take effect the next time a user logs on.
EXPIRE = *n*	the number of days to elapse before the password expires. A value of 0 for *n* indicates the password never expires.
MINCHAR = *n*	the minimum number of characters in a password string. The valid range for *n* is 1 to 30.
MAXCHAR = *n*	the maximum number of characters in a password string. The valid range for *n* is 1 to 30.
DIGITS = *c*	whether digits can appear after the first character in a password string. Valid values for *c* are Y, y, N, or n.

Syntax Element …	Specifies …
SPECCHAR = *c*	whether special characters can appear in a password string. Valid values for *c* are Y, y, N, or n.
MAXLOGONATTEMPTS = *n*	the number of incorrect logon attempts allowed before locking the user. A value of 0 for *n* indicates never to lock the user.
LOCKEDUSEREXPIRE = *n*	the number of minutes to elapse before unlocking a locked user. A value of 0 for *n* unlocks the user immediately. A value of -1 for *n* locks the user indefinitely.
REUSE = *n*	the number of days to elapse before a password can be reused. A value of 0 for *n* allows the password to be reused immediately.

ANSI Compliance

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

Authorization

To use MODIFY PROFILE, you must have the DROP PROFILE privilege.

New users do not implicitly have the DROP PROFILE privilege. User DBC or a user who has the DROP PROFILE privilege WITH GRANT OPTION can grant this privilege to another user.

Effects of Modifying User Parameters in Profiles

The rule that determines when changes to user parameters in a MODIFY USER statement take effect also applies to parameter changes in a MODIFY PROFILE statement.

MODIFY PROFILE has the following effects on sessions logged on with the profile:

- Changes to spool and temporary space take effect immediately.
- The current account and database do not change until the next time users log on or explicitly change them.

  However, changes to the list of valid accounts take effect immediately. Users can only explicitly change to an account in the list of accounts available to the user.

Effects of Modifying Password Settings in Profiles

Changes to password attribute settings in a profile do not require a system restart to take effect. The new settings take effect the next time users who are assigned the profile log on.

Changing Spool and Temporary Settings in Profiles

The spool and temporary settings in a MODIFY PROFILE statement cannot exceed the settings of the profile modifier at the time of modification.

Subsequent reductions to the spool or temporary space of the profile modifier do not reduce the spool or temporary settings in a previously-modified profile.

Example

Assume user *anderson* has a spool setting of 500 000 bytes and user *brewster* has a spool setting of 800 000 bytes.

User *anderson* can create a *finance* profile with a spool setting of 500 000 bytes:

```
CREATE PROFILE finance AS
  SPOOL = 500000;
```

User *anderson* can assign this profile to user *brewster*, effectively reducing the spool limit from 800 000 to 500 000 bytes:

```
MODIFY USER brewster AS
  PROFILE = finance;
```

If the owner of *anderson* later increases the spool allotment to 1 000 000, *anderson* can also increase the *finance* profile spool setting up to 1 000 000 bytes. All users assigned that profile, including *brewster*, automatically get 1 000 000 bytes of spool space.

```
MODIFY PROFILE finance AS
  SPOOL = 1000000;
```

If the owner of *anderson* later decreases the spool allotment back to 500 000 bytes, the *finance* profile and the allotment for *brewster* are not affected.

For More Information

FOR more information on ...	SEE ...
assigning profiles to users	• "CREATE USER" on page 1-296. • "MODIFY USER" on page 1-364.
obtaining the DROP PROFILE privilege	"GRANT (SQL Form)" on page 2-22.
system-level password attributes	"Controlling Access to Teradata RDBMS" in *Teradata RDBMS Security Administration*.

MODIFY USER

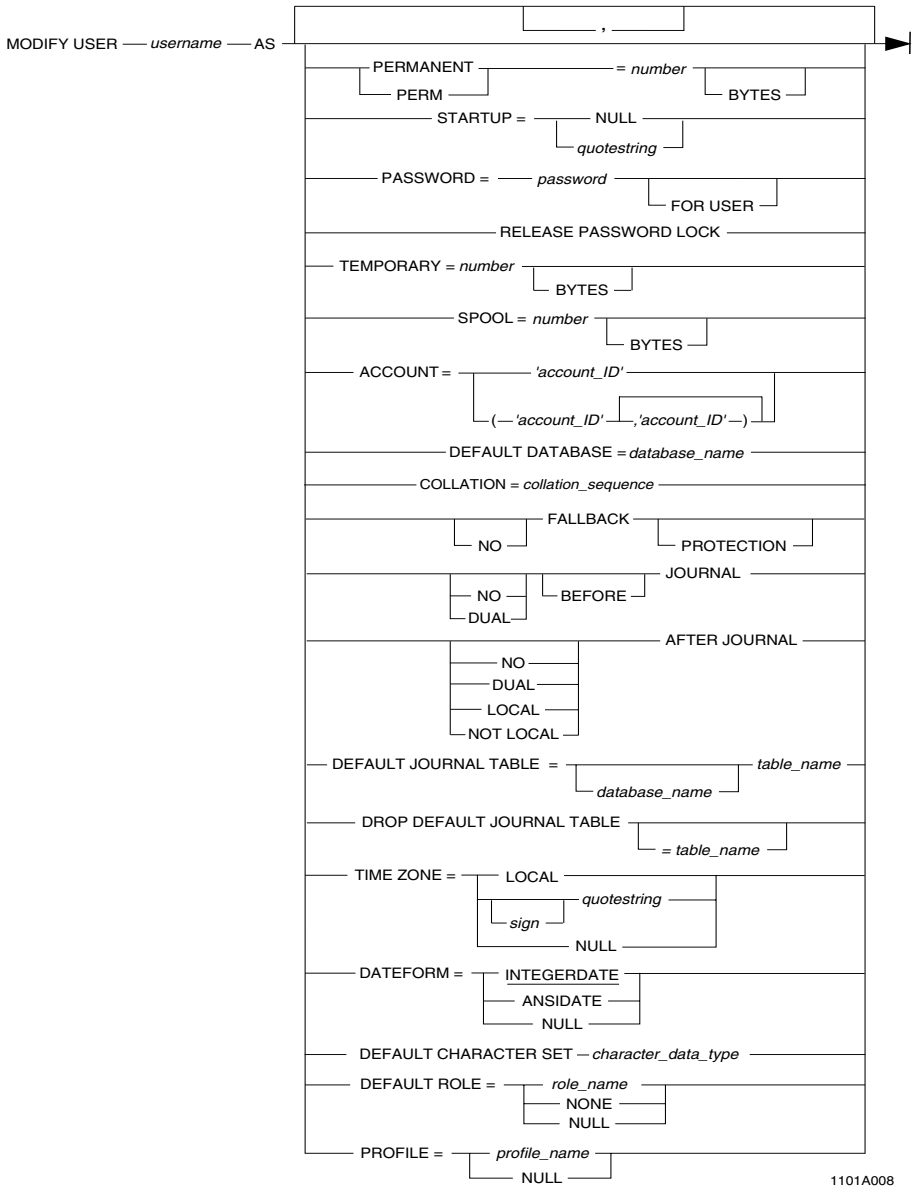
Purpose

Changes the options that were specified when a user was created.

Invocation

Executable.

Syntax



where:

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>user_name</td><td>the name of the user to be modified.</td></tr>
<tr><td>PERMANENT = number

BYTES</td><td>a new value, in bytes, for fixed space allocation.

number can be entered as an integer, decimal, or floating point value.

BYTES is a noise keyword that redundantly specifies the unit for permanent file space allocation.</td></tr>
<tr><td>STARTUP =</td><td>a keyword introducing the startup options.

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>NULL</td><td>any existing startup string for the user is to be deleted.</td></tr>
<tr><td>quotestring</td><td>a replacement for the user startup string, which consists of one or more SQL statements that are performed when the user logs onto Teradata.

A startup quotestring can be up to 255 characters, must be terminated by a SEMICOLON character, and enclosed by APOSTROPHE characters.

A startup Kanji quotestring can be up to 255 bytes, must be terminated by a SEMICOLON character, and be enclosed by APOSTROPHE characters.

The USING modifier is not supported, and if a DDL statement is used, no other statement is allowed in the string.</td></tr>
</table>
</td></tr>
<tr><td>PASSWORD =</td><td>a new password for the user. A password is an SQL word and can contain up to 30 characters.

If the name is Japanese, see the formula in the chapter “SQL Lexicon” in “Name Validation for Japanese Language Support” in Teradata RDBMS SQL Reference, Volume 1.</td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>FOR USER</td><td>to remove the existing password immediately and set the password expiration date to zero, depending on the value of the Expire option in the profile assigned to the user

If user is not assigned a profile or if the Expire option in the assigned profile is not set, then the expiration date is set to the value of the ExpirePassword column specified in DBC.SysSecDefaults as shown in the following table.

<table>
<tr><th>IF the Expire value in the user profile or the value for ExpirePassword in DBC.SysSecDefaults is this value …</th><th>THEN the current password …</th></tr>
<tr><td>0</td><td>expires immediately and the new temporary password becomes valid indefinitely.

This is a potential security breach and you should ensure that it does not happen.</td></tr>
<tr><td>> 0</td><td>expires immediately and the new temporary password becomes valid.

The value for ExpirePassword is reset to 0.

The new temporary password expires at first logon and a new, permanent password must be created by the user at that time.</td></tr>
</table>

The password attribute settings in a user profile override the corresponding system-level password settings.

For more information about DBC.SysSecDefaults and ExpirePassword, see Teradata RDBMS Security Administration and Teradata RDBMS Data Dictionary. For more information about profiles, see “CREATE PROFILE” on page 1-173.</td></tr>
<tr><td>RELEASE PASSWORD LOCK</td><td>keywords to release a user lock.

A user row in DBC.DBase becomes locked if the number of successive erroneous logon attempts reaches the site-specified threshold.

The user row remains locked to logon attempts until the site-defined time has passed, or until a MODIFY USER statement is submitted to release the lock on the user.</td></tr>
<tr><td>TEMPORARY = number

BYTES</td><td>how much space (in bytes) is to be allocated for creating temporary tables by this user. Temporary space is reserved prior to spool space for any user defined with this characteristic.

Disk usage for a materialized temporary table is charged to the temporary space allocation of the user who referenced the table.

BYTE is a noise keyword that redundantly specifies the unit of space allocation.</td></tr>
<tr><td>SPOOL = number

BYTES</td><td>a new value for the maximum space allocation for spool files, in bytes. number can be entered as an integer, decimal, or floating point value.

BYTES is a noise keyword that redundantly specifies the unit for spool file space allocation.</td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>ACCOUNT =
account_id</td><td>a replacement for one or more account identifiers (quotestring) associated with the user. The quotestring can be up to 30 characters and must be delimited by APOSTROPHE characters.

If account_ID is Japanese, see the formula discussed in “Name Validation for Japanese Language Support” in Chapter 3: “SQL Lexicon” of Teradata RDBMS SQL Reference, Volume 1.

Japanese character account_IDs can be up to 30 bytes long.

A list of account_IDs must be separated by COMMA characters and enclosed by LEFT PARENTHESIS and RIGHT PARENTHESIS characters.

The first (or only) account_id in the list becomes the new default account_id for the user.</td></tr>
<tr><td>DEFAULT DATABASE =
database_name</td><td>a new name for the default database that is established each time the user logs on.</td></tr>
<tr><td>COLLATION =
collation_sequence</td><td>the collation option that redefines the user default collation.

See the chapter “International Character Support” in Teradata RDBMS SQL Reference, Volume 1 for more information about collation options.

The valid collation sequences are defined in the following table.

<table>
<tr><th>This keyword …</th><th>Defines this collation …</th></tr>
<tr><td>ASCII</td><td>HOST

The collation is ASCII for configurations other than an IBM channel-attached host. ASCII here refers to the Teradata extension to the ASCII standard.</td></tr>
<tr><td>EBCDIC</td><td>HOST

The collation is EBCDIC for an IBM channel-attached host.</td></tr>
<tr><td>MULTINATIONAL</td><td>One of the European (diacritical) or Japanese language (Kanji) sort sequences (see “Collation” on page 1-375).</td></tr>
<tr><td>HOST</td><td>that of the logon client system.

This is the default.</td></tr>
<tr><td>CHARSET_COLL</td><td>HOST

The collation is binary in the order of the current client character set.

Strings are compared character-by-character and padded with the pad character for the character data type where necessary.

When not CASE SPECIFIC, lowercase characters are mapped to their uppercase counterparts.</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>COLLATION =
collation_sequence

(continued)</td><td>
<table>
<tr><th>This keyword …</th><th>Defines this collation …</th></tr>
<tr><td>JIS_COLL</td><td>Japanese Industrial Standards (JIS).

The collation sequence is as follows.

1 Characters and symbols from the JIS X 0201 standard (in JIX X 0201 order)

2 Ideographs, characters, and symbols from the JIS X 0208 standard (in JIS X 0208 order)

3 Ideographs, characters, and symbols from the JIS 0212 standard (in JIS X 0212 order)

4 IBM Kanji ideographs not present in JIS X 0208 and JIS X 0212 (in KanjiEBCDIC order)

5 User-defined ideographs (in U+ order)

6 Any remaining characters in UNICODE (in U+ order)

Treatment of KANJISJIS, GRAPHIC, and LATIN character types is as if the data were first converted to UNICODE and then the appropriate JIS_COLL ordering is applied.</td></tr>
</table>
</td></tr>
<tr><td>NO
FALLBACK
PROTECTION</td><td>a new default for duplicate copy protection of each data table subsequently created in the database. The current fallback setting for existing data tables remains unchanged.

The FALLBACK keyword used alone implies PROTECTION.</td></tr>
<tr><td>NO
DUAL
BEFORE
JOURNAL</td><td>a new default for the number of before change images to be maintained for data tables subsequently created in the database.

The NO keyword terminates any current journaling default.

If the JOURNAL keyword is specified without NO or DUAL, single copy journaling is implied. If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection.

Existing images are not affected until the corresponding table is updated.

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

If only one type is specified, then the current default is modified only for that type.

If both BEFORE and AFTER are specified then the two must not conflict.

The JOURNAL keyword without AFTER or BEFORE indicates that both types of images are to be maintained. In this case, the current default for either or both types is modified accordingly.

For example, if only AFTER JOURNAL is specified, the current default for before change images remains in effect.</td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>NO
DUAL
LOCAL
NOT LOCAL
AFTER JOURNAL</td><td>a change to the type of image to be maintained for the table; any existing images are not affected until the table is updated.

NO, DUAL, LOCAL, or NOT LOCAL: the NO and DUAL options specify the number of after-change images to be maintained for the table.

NOT LOCAL and LOCAL specify whether single after-image journal rows for non-fallback data tables are written on the same virtual AMP (LOCAL) as the changed data rows, or on another virtual AMP in the cluster (NOT LOCAL).</td></tr>
<tr><td>DEFAULT
JOURNAL TABLE=
database_name.
table_name</td><td>a redefinition of the current journal table or removes its status as the default for the database being modified.</td></tr>
<tr><td>DROP DEFAULT
JOURNAL TABLE =
table_name</td><td>removal of the default status of the journal table currently defined as the default for the database being modified.

If the journal table resides in the database being modified, DROP also deletes the table from the system. An error message is returned if the DROP request would delete a journal table that is being used by active data tables.

The table_name parameter is required if this clause is specified without the DROP keyword.

If a database is not specified, then the database being modified is assumed.

If the database being modified does not have a journal table, table_name is created.

If a different database is specified, then it must already exist and table_name must have been defined as its default journal table.

Specifying this option does not change the status of existing data tables in the modified database.</td></tr>
<tr><td>TIME ZONE =</td><td>the default time zone displacement for the user.

Valid options are listed in the following table.

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>LOCAL</td><td>Changes the value for TIME ZONE to that defined for the system default.

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 use MODIFY USER.</td></tr>
<tr><td>± quotestring</td><td>Signed value that sets a non-system-default interval offset for converting the user TIME ZONE.

The format is hh:mm.</td></tr>
<tr><td>NULL</td><td>No default TIME ZONE is defined for the user.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>DATEFORM =</td><td>the default format for importing and exporting DATE values for the user.

For further information, see “DATE Formats” in the chapter “DateTime and Interval Data Types” in Teradata RDBMS SQL Reference, Volume 3.

Valid options are listed in the following table.

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>INTEGERDATE</td><td>Sets the DATEFORM option to import and export DATE values as encoded integers.

INTEGERDATE results in a default DATE format in field mode of 'YY/MM/DD' for date columns created and for date constants in character form. You can use the tdlocaledef utility to change the 'YY/MM/DD' default DATE format. For details, see Teradata RDBMS Utilities.

INTEGERDATE is the DATEFORM default.</td></tr>
<tr><td>ANSIDATE</td><td>Sets the DATEFORM option to import and export DATE values as CHAR(10).

Results in a 'YYYY-MM-DD' date format for date columns created and for date constants in character form.</td></tr>
<tr><td>NULL</td><td>Sets DATEFORM null.

When DATEFORM is null, the default format for a date in field mode, and the default format for importing and exporting DATE values, is the same as the system default defined by the DATEFORM flag in the DBSControl record.</td></tr>
</table></td></tr>
<tr><td>DEFAULT CHARACTER SET =
character_data_type</td><td>the user default for character data type.

All changes affect only those columns created after the user definition has been modified. Columns created previously by this user retain their original type.

If no user default character type is defined, then one is assigned automatically according to the following rules.

<table>
<tr><th>WHEN no user default character type is defined for this type of site …</th><th>The definition defaults to this character_data_type …</th></tr>
<tr><td>U.S. and European (all users)</td><td>LATIN</td></tr>
<tr><td>Japanese (all users except user DBC)</td><td>UNICODE</td></tr>
<tr><td>Japanese (user DBC only)</td><td>KANJI1</td></tr>
</table>
Valid options are listed in the following table.</td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>DEFAULT CHARACTER SET = character_data_type
(continued)</td><td>
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>LATIN</td><td>fixed 8-bit Latin.</td></tr>
<tr><td>UNICODE</td><td>fixed 16-bit Unicode.</td></tr>
<tr><td>GRAPHIC</td><td>fixed 16-bit Unicode as defined by IBM Corporation for DB2.</td></tr>
<tr><td>KANJISJIS</td><td>mixed single and multibyte characters defined by KanjiSJIS.</td></tr>
<tr><td>KANJI1</td><td>mixed single and multibyte characters defined by KanjiEBCDIC, KanjiShift-JIS, or KanjiEUC, depending on the current character set for the session.</td></tr>
</table>
</td></tr>
<tr><td>DEFAULT ROLE =</td><td>the default role for the user.
Valid options appear in the following table:
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>role_name</td><td>Changes the value for the default role assigned to the user to role_name.</td></tr>
<tr><td>NULL</td><td rowspan="2">Do not assign a default role to the user.</td></tr>
<tr><td>NONE</td></tr>
</table>
</td></tr>
<tr><td>PROFILE =</td><td>the profile for the user.
Valid options appear in the following table:
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>profile_name</td><td>Changes the value for the profile assigned to the user to profile_name.</td></tr>
<tr><td>NULL</td><td>Do not assign a profile to the user.</td></tr>
</table>
</td></tr>
</table>

ANSI Compliance

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

Authorization

You must have the DROP USER privilege on the referenced user.

A user must have DROP USER privileges on himself in order to change PERMANENT, SPOOL, ACCOUNT, and PROFILE parameters.

MODIFY USER cannot be used to modify a database.

If the user who is entering the MODIFY USER statement is the same as the user who is referenced in the statement, then changing BEFORE JOURNAL, AFTER JOURNAL, DEFAULT JOURNAL TABLE, FALLBACK, PASSWORD, STARTUP, COLLATION, and DEFAULT DATABASE options does not require any access privilege.

To enter a MODIFY USER statement that assigns a profile, you must have the DROP PROFILE privilege.

Option Checking

Several options can be included in a single MODIFY USER statement. No order is imposed, but an error is reported if options are duplicated or conflicting options are included.

MODIFY USER Rules

The following rules apply to the use of MODIFY USER.

- Any or all options can be changed using the same MODIFY statement. A changed option applies to all subsequent operations that occur for the user.
- If an existing option is not changed by a MODIFY USER statement, that option remains the same.

PERM, SPOOL, or TEMPORARY Space is Changed Accordingly

If necessary, the defined PERM, SPOOL, or TEMPORARY space is changed to the next higher value that is a multiple of the number of AMPs on the system.

Modifying a User to Have a Null Password

You cannot modify a user definition to have a null password. The following procedure explains how you can permit an existing user to log on with a null password.

Step	Action

1	Grant user rhh the following logon privileges using the GRANT LOGON statement.
	`GRANT LOGON ON ALL`
	`TO rhh`
	`WITH NULL PASSWORD;`

2	Install an appropriate security user exit on TDP for the user. See the chapter "TDP Security Features" in *Teradata TDP Reference* for instructions on how to do this.

Modifying a User for Single Sign-On

Single sign-on is available only in the Windows environment. This feature permits a user to log onto a computer one time and, for the duration of that logon, to be able to access a Teradata database without providing a user name, password, or account name. You do this by granting the user logon privileges with a null password.

See *Teradata RDBMS Utilities* for details about the single sign-on feature.

The following procedure modifies a user to be able to log onto Teradata through a gateway that does not have the Append Domain Name option set using the Gtwcontrol utility. This user is already defined to Windows as user rhh.

Step	Action

1	Grant user rhh the following logon privileges using the GRANT LOGON statement.
	`GRANT LOGON ON ALL`
	`TO rhh`
	`WITH NULL PASSWORD;`

The following procedure modifies a Teradata user who can log onto Teradata through a gateway that has Append Domain Name set. This user is already defined to Windows as user rhh and her account is in the esw2kdev domain.

Step	Action

1	Grant user rhh the following logon privileges using the GRANT LOGON statement.
	`GRANT LOGON ON ALL`
	`TO "rhh@esw2kdev"`
	`WITH NULL PASSWORD;`

Procedure for Creating a Temporary Password

The following procedure expires the existing password and creates a temporary password to replace it. The user must then create a new, secure password when they log on for the first time.

Step	Action

1	SELECT ExpirePassword from the DBC.SecurityDefaults view.
	`SELECT ExpirePassword`
	`FROM DBC.SecurityDefaults;`

2	Examine the reported value for ExpirePassword.
	
	IF the value is … / **THEN …**
	0 → Change it to a value > 0.
	`UPDATE DBC.SecurityDefaults`
	`SET ExpirePassword = 2;`
	Go to Step 3.
	> 0 → Go to Step 3.

3	Perform MODIFY USER with the FOR USER option.
	`MODIFY USER JWilkins AS`
	`PASSWORD = mysecret`
	`FOR USER;`
	The existing password expires and is replaced by ‘mysecret.’
	The value for ExpirePassword is reset to 0.
	The temporary password expires immediately when JWilkins logs on for the first time, and she must create a new, permanent password at that time.
	If ExpirePassword = 0 when you perform MODIFY USER, the statement completes but the temporary password that was created does not expire and can be used indefinitely. This is a potential security breach and you should ensure that it does not happen.

You can also use a profile to create a temporary password.

Step	Action

1	Create a profile and set the Expire option to a value greater than 0.
	`CREATE PROFILE temp_password AS`
	`PASSWORD = (EXPIRE = 2);`

2	Assign the profile to the user.
	`MODIFY USER JWilkins AS`
	`PROFILE = temp_password,`
	`PASSWORD = mysecret FOR USER;`

DEFAULT DATABASE

If a DEFAULT DATABASE clause is specified, the default database is changed for subsequent user sessions. You can perform a DATABASE statement o change the default database for the current session.

The default database for a user cannot be set to NULL. To remove the current DEFAULT DATABASE, you can set it to the user name associated with the MODIFY USER statement. This is the default setting if a DEFAULT DATABASE clause has not been specified.

Two MODIFY Statements Needed to Change Default Journal Table

Two MODIFY statements are needed to change a default journal table that resides in the user space being modified. The first statement drops the present journal table from the system. After the table is dropped, a second statement is required to define a new default journal table.

An error message is returned if the journal table is being used by active data tables.

Dropped Journal Tables Cannot be Restored

A dropped journal table cannot be restored. Archives containing data tables that reference the journal are no longer valid, and cannot be used.

Two or More Databases/Users Can Share a Journal Table

Two or more databases/users can share a journal table. However, if the sharing databases/users retain as their default the name of a journal table that has been dropped from the system, subsequent CREATE TABLE statements that use the corresponding default will return an error message.

Collation

The COLLATION option determines the ordering of data characters during comparison operations and when sorting data in response to a SELECT query that includes an ORDER BY or WITH...BY clause.

The MULTINATIONAL parameter defines the name of the default collation sequence for the user as one of the International sort orders (see “ORDER BY Clause” in the chapter “The SELECT Statement” in *Teradata RDBMS SQL Reference, Volume 6*).

During any session, you can override the COLLATION attribute by executing a SET SESSION COLLATION statement.

Local Journaling

The LOCAL single AFTER image journal is supported analogously to single BEFORE image journal:

- Access privileges required to create or drop LOCAL single AFTER image journal are the same as for the analogous operation on single BEFORE image journal. The DROP DATABASE privilege is required.
- LOCAL single AFTER image journaling is restricted to non-fallback data tables.
- Archive/Recovery rules for LOCAL single AFTER image journal and 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.

MultiLoad, Teradata Warehouse Builder, FastLoad and Archive/Recovery are all impacted by the use of LOCAL journaling.

Locks and Concurrency

Without the SPOOL, TEMP, or PERM options, MODIFY USER places an ACCESS lock on the database being altered and thus runs concurrently with all DML statements. The SPOOL, TEMP, and PERM options require an EXCLUSIVE lock.

MODIFY USER also places a row-hash WRITE lock on the database for DBC.DBase. The SPOOL, TEMP, and PERM options then place a second lock on the immediate parent for DBC.DBase. For PERM, the second lock is a row-hash WRITE lock. For SPOOL or TEMP, the second lock is a row-hash READ lock.

The SPOOL, TEMP, and PERM options also place a row-hash WRITE lock on DBC.DataBaseSpace. The PERM option then places a second row-hash WRITE lock on the immediate parent for DBC.DataBaseSpace.

Modifying a Default Role

Roles define access privileges on database objects. Specifying a default role for a user gives the user access to all the objects that the role has privileges to. If the default role has members that are roles, the user also has access to all the objects that the nested roles have privileges to.

The default role for a user automatically becomes the current role for the session when the user first logs on. Modifying the default role for a user who is logged on does not affect the current role for a session until the next time the user logs on.

The user must be a member of the new default role.

The user remains a member of the previously-assigned default role, if any, and can use the SET ROLE statement during a session to change the current role to the previously-assigned default role.

Only one role can be *current* in a session. More than role can be *active*, however. The active roles for a user include the current role and all of its members that are roles. Privileges validation uses the active roles for a user.

FOR more information on ...	SEE ...
creating roles	"CREATE ROLE" on page 1-180.
adding members to roles	"GRANT (Role Form)" on page 2-19.
changing the current role for a session	"SET ROLE" on page 1-382.

Modifying a Profile

Profiles define system parameters and user-level password security attributes. Assigning a profile to a group of users makes sure that all group members operate with a common set of parameters.

When a user who has both profile and user settings logs on, Teradata sets the current settings to the profile settings instead of the user settings. After logging on, users can use the SET SESSION statement to override the default database or account settings for a session.

Using MODIFY USER to modify the profile for a user has the following effects on sessions logged on with the profile:

- Password attribute settings take effect the next time the user logs on.
- Spool and temporary space settings in the new profile take effect immediately.
- The current account and database settings in the new profile do not take effect until the next time the user logs on, or if the user uses the SET SESSION statement to explicitly change them.

  However, the list of valid accounts in the new profile immediately affects the accounts available to a user who wants to change the current account for a session.

FOR more information on ...	SEE ...
creating profiles	"CREATE PROFILE" on page 1-173.
modifying profiles	"MODIFY PROFILE" on page 1-359.
obtaining the DROP PROFILE privilege	"GRANT (SQL Form)" on page 2-22.

Examples

The following examples illustrate the use of MODIFY USER.

Example 1

The following statement can be used to change the permanent space allocation for user Peterson:

```
MODIFY USER Peterson AS
 PERMANENT = 6000000 BYTES;
```

Example 2

The following statement can be used to change user the default database and password for Chin:

```
MODIFY USER Chin AS
 DEFAULT DATABASE = Personnel,
 PASSWORD = Nitram;
```

Example 3

Two MODIFY statements are needed to change the default journal table if it resides in the user being modified.

For example, suppose the journal table FinCopy resides in user Jones. To change the default journal table from FinCopy to Jrnl1, the present default journal table must be dropped and then a new journal table must be created.

The first statement removes FinCopy as the default and drops it from the system. If there are any existing tables that use it as their journal table, the statement returns an error.

```
MODIFY USER Jones AS
 DROP DEFAULT JOURNAL TABLE;
```

When the current journal table has been dropped, the second statement creates a new default journal table:

```
MODIFY USER Jones AS
 DEFAULT JOURNAL TABLE = Jrnl1;
```

Example 4

If the current journal table does not reside under the user being modified, the following statement could be used to change the default journal table from FinCopy to Jrnl1:

```
MODIFY USER Peterson AS
 DEFAULT JOURNAL TABLE = Jrnl1;
```

Example 5

You do not need access privileges to change the PASSWORD, STARTUP, FALLBACK, or DEFAULT DATABASE definitions in space you own. Otherwise, the user submitting a MODIFY statement must have the DROP privilege on the user or database being changed.

By way of example, although Marks is not the creator of his own space and does not own himself, he can submit a MODIFY USER statement to change his password, modify his startup string, or redirect his default database. Marks can resize Finance, as well.

Change password and other options on user Marks:

```
MODIFY USER marks AS
 PASSWORD =         design,
 SPOOL =            1500000,
 STARTUP =          'EXEC paystat;',
 DEFAULT DATABASE =  payroll;
```

Example 6

The following statement creates a profile called *human_resources* that defines spool space and default database system parameters:

```
CREATE PROFILE human_resources AS
  SPOOL =            1500000,
  DEFAULT DATABASE = personnel;
```

The following statement assigns the *human_resources* profile to user Ortega:

```
MODIFY USER ortega AS
  PROFILE = human_resources ;
```

RENAME MACRO
RENAME PROCEDURE
RENAME TABLE
RENAME TRIGGER
RENAME VIEW

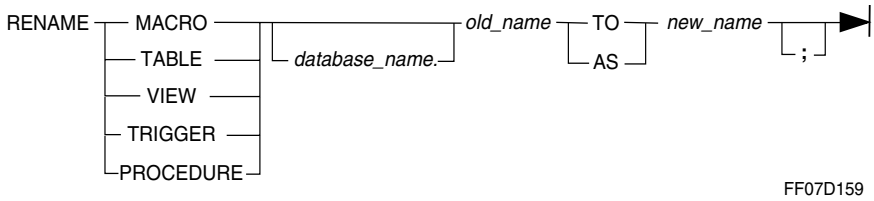
Purpose

Renames an existing macro, stored procedure, table, trigger, or view.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
database_name	the optional name of the database in which the object to be renamed resides. You cannot use this statement to change the database qualifier for the object.
old_name	the existing name for the macro, stored procedure, table, trigger, or view.
TO AS	keywords introducing the new name for the macro, stored procedure, table, trigger, or view.
new_name	the new name for the macro, stored procedure, table, trigger, or view.

ANSI Compliance

RENAME MACRO, RENAME PROCEDURE, RENAME TABLE, and RENAME VIEW are Teradata extensions to the ANSI SQL-99 standard.

RENAME TRIGGER is ANSI SQL-99-compliant.

Authorization

You must have DROP privileges on the macro, stored procedure, table, trigger, or view to be renamed, and the appropriate CREATE privileges on the containing database.

Renaming Global Temporary Tables

You can rename a global temporary table only when there are no materialized instances of that table anywhere in the system.

Qualifying an Object From Another Database

To rename a macro, stored procedure, table, trigger, or view that is not contained in the same database as your current default database setting, you must qualify both the old and new names with the name of the database in which the object exists.

Referenced Objects

You cannot rename a table on which a hash or join index is defined. First drop the index, then rename the table.

What Gets Changed

When a table is renamed, only the table name is changed. All statistics and privileges belonging to the table remain with it under the new name.

When a macro, stored procedure, table, trigger, or view is renamed, an exclusive lock is placed on the item being renamed.

Example 1

The following statement renames the Employee table:

```
RENAME TABLE Employee TO Emp;
```

Example 2

The following statement renames the Employee table if the current default database setting for the user is something other than Personnel:

```
RENAME TABLE Personnel.Employee TO Personnel.Emp;
```

SET ROLE

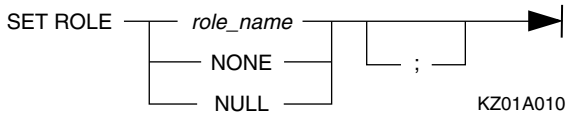
Purpose

SET ROLE sets the current role for a session.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
role_name	the name of the role to set as the current role for a session.
NONE NULL	to disable the current role for a session.

ANSI Compliance

SET ROLE is ANSI SQL-99-compliant, except for the NULL option, which is a Teradata extension.

Authorization

No privilege is needed to disable the current role for a session.

To set a role as the current role for a session, you must be a member of the role.

Rules for Using SET ROLE

The SET ROLE statement is not supported within a stored procedure definition.

Example 1: Changing the Active Role

The following statement changes the current role for a session to *administration*:

```
SET ROLE administration;
```

Example 2: Disabling the Current Role

The following statements disable the current role for a session:

```
SET ROLE NONE;
SET ROLE NULL;
```

For More Information

FOR more information on ...	SEE ...
creating roles	"CREATE ROLE" on page 1-180.
granting privileges on database objects to roles	"GRANT (SQL Form)" on page 2-22.
adding members to roles	"GRANT (Role Form)" on page 2-19.
assigning default roles to users	• "CREATE USER" on page 1-296. • "MODIFY USER" on page 1-364.

SET SESSION

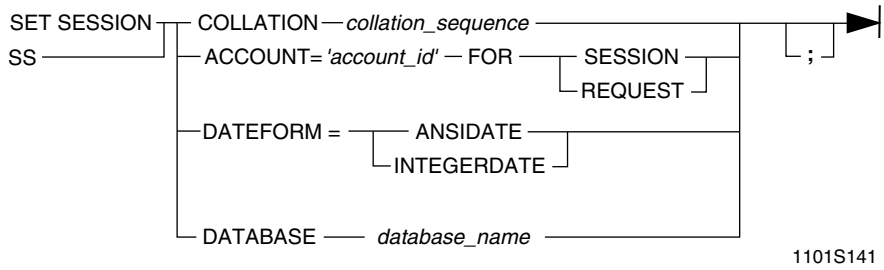
Purpose

Overrides the collation sequence, account priority, or DateForm currently in effect for the session.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
COLLATION	that the collation sequence in effect for the session is to be changed.
collation_sequence	the name of the collation sequence to be established for the session.

This collation sequence name keyword …	Specifies that comparison and sort operations are to use this collation type …
ASCII	ASCII
EBCDIC	EBCDIC
MULTINATIONAL	European (diacritical) character or Kanji character.
CHARSET_COLL	binary ordering based on the current client character set.
JIS_COLL	logical ordering of characters based on the Japanese Industrial Standards collation.
HOST	EBCDIC for IBM channel-attached clients. ASCII for all other clients. This is the default collation sequence.

Syntax Element …	Specifies …
ACCOUNT	that the current account priorities are to be changed.
account_id	the account and its new performance group code prefix for the current session or request. This must be one of the accounts defined originally for the logged-on user. You cannot use SQL to change the account performance group for any other user. To change account performance groups for other users, you must use the parallel PMPC statement SET SESSION ACCOUNT. You can only downgrade priorities. See *Teradata RDBMS Utilities* and *Teradata RDBMS System Administration* for discussions of performance groups and priorities.
SESSION	that the performance group is to be changed for the remainder of the session.
REQUEST	that the performance group code prefix is to be changed for the next request only (next being defined as the first request performed after the current SET SESSION ACCOUNT performs). After that, the previous performance group declaration resumes. The specified performance group is kept in volatile memory, not in DBC.Sessiontbl, so a subsequent transparent system crash and recovery effectively voids this statement. To be sure the request performs correctly, set SET CRASH to NOWAIT_TELL or verify the result.
DATEFORM	the default format for a date in field mode, and the default format for importing and exporting DATE values, for the session.
ANSIDATE	the DATEFORM option to import and export DATE values as CHAR(10). ANSIDATE results in a 'YYYY-MM-DD' date format in field mode for date columns created and for date constants in character form.
INTEGERDATE	the DATEFORM option to import and export DATE values as encoded integers. When the session DATEFORM is INTEGERDATE, the default DATE format in field mode is 'YY/MM/DD' for date columns created and date constants in character form. You can use the tdlocaledef utility to change the 'YY/MM/DD' default DATE format. For details, see *Teradata RDBMS Utilities*.
DATABASE	that the current default database is to be changed.
database_name	the name of the new default database for the remainder of the current session.

ANSI Compliance

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

Authorization

None.

Related Information

For additional information about the SET SESSION options, including individual syntax diagrams and examples, see the following statements:

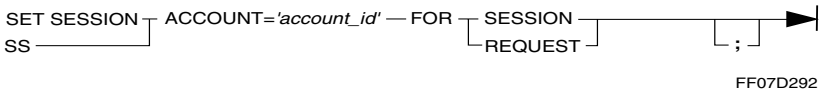
- "SET SESSION ACCOUNT" on page 1-387
- "SET SESSION COLLATION" on page 1-391
- "SET SESSION DATABASE" on page 1-397
- "SET SESSION DATEFORM" on page 1-398

SET SESSION ACCOUNT

Purpose

Dynamically changes your account priorities for the duration of a session or for one SQL request only.

Syntax



where:

Syntax Element …	Specifies …
account_id	the account and its new performance group code prefix for the current session or request. This must be one of the accounts defined originally for the logged-on user. You cannot use SQL to change the account performance group for any other user. To change account performance groups for other users, use the PM request SET SESSION ACCOUNT (see *Teradata RDBMS PM/API Reference*). You can only downgrade priorities. See *Teradata RDBMS System Administration* for a discussion of performance groups and priorities.
SESSION	that the performance group is to be changed for the remainder of the session.
REQUEST	that the performance group code prefix is to be changed for the next request only (*next* being defined as the first statement performed after the current SET SESSION ACCOUNT performs). After that, the previous performance group declaration resumes.

Usage Notes

Changing your account priorities dynamically is sometimes referred to as self-nicing.

SET SESSION ACCOUNT permits you to change your own account priority (or priorities) either for the next SQL query you run or for all queries run for the remainder of the current session.

WHEN *account_id* is changed for this …	THEN the performance group change is effective for this duration …
Session	the current session. Performance group assignments revert to their previously defined values when the session ends.
Request	the next request performed after SET SESSION ACCOUNT performs.

Examples

The example set for SET SESSION ACCOUNT use the following *account_ids* as logon definitions for the performance groups of the three accounts Sales, Development, and Marketing.

- $HSales
- $MMarketing
- $LDevelopment

The following table illustrates the defined, permitted, and forbidden performance groups (account priorities) for the accounts used in the examples.

Account	Defined Priority	Priority Definition	Permitted Priority Changes	Forbidden Priority Changes
Sales	$H	High	<= $H	>$H
Marketing	$M	Medium	< $H	>$M
Development	$L	Low	None	>$L

Example 1

You want to change the performance group for the Sales account for the remainder of the current session as follows.

Change the priority for account Sales from $H (high priority) to $L (low priority).

```
SET SESSION ACCOUNT = '$LSales'FOR SESSION;
```

The account priority change is valid because the originally defined performance group for the Sales account, $H, is not exceeded.

Example 2

You want to change the performance group for account Sales from $H (high priority) to $M (medium priority) for the duration of the current session.

```
SET SESSION ACCOUNT = '$MSales'FOR SESSION;
```

This account priority change is also valid because $M does not exceed the originally defined performance group for the Sales account, which is $H.

Example 3

You could also change the performance group for the Marketing account for the duration of the current session from $M to $L, for example:

```
SET SESSION ACCOUNT = '$LMarketing' FOR SESSION;
```

This account priority change is valid because the originally defined priority for the Marketing account, $M, is not exceeded.

Suppose you log on to a machine where the same original set of three accounts are defined. This time you want to change the performance group for the Development account from $L to $H.

This request fails because the new priority exceeds that originally defined by the performance group for the Development account, $L.

Similarly, if you were to attempt to change the performance group from $L to $M, the change again fails because the new priority still exceeds that originally defined for Development.

These changes are not valid because the new priorities exceed those originally defined by the performance groups for the named accounts.

Example 4

Suppose another user has a business-critical query that must be performed immediately, but the account it originates from, Marketing, has a lower priority ($M) than your logged on account, Sales, which has a priority of $H. You change your performance group code prefix to $L (low priority) in order to lower your own priority and bump the business-critical query up higher in the execution queue.

Assume that the logged on accounts and priorities are these:

- $HSales
- $MMarketing

You can change the priority from $H to $L for the Sales account as follows.

```
SET SESSION ACCOUNT = '$LSales' FOR SESSION;
```

This action permits the business-critical query to run sooner than it otherwise would have. The priority change is only effective for the duration of the current session—it does not persist after the session ends.

Example 5

You have a relatively low priority, but resource-intensive query that should not steal CPU cycles from more important jobs. The logged on performance group code prefix is $H for Sales, the account from which you must run the query.

To change the priority for your resource-intensive query, perform the following statement:

```
SET SESSION ACCOUNT = '$LSales' FOR REQUEST;
```

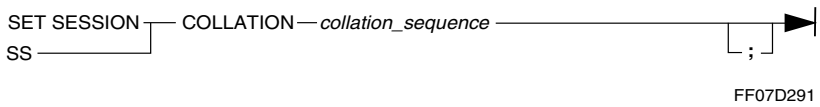
Then run your query immediately following the SET SESSION ACCOUNT request. Your priority returns to its normal defined value and your remaining queries take the priority assigned for the account they run from as soon as the query completes.

SET SESSION COLLATION

Purpose

Changes the collation sequence for the current session.

Syntax



where:

Syntax Element …	Specifies …
collation_sequence	the name of the collation sequence to be established for the session.

This collation sequence name keyword …	Specifies that comparison and sort operations are to use this collation type …
ASCII	ASCII
EBCDIC	EBCDIC
MULTINATIONAL	European (diacritical) character or Kanji character.
CHARSET_COLL	binary ordering based on the current client character set.
JIS_COLL	logical ordering of characters based on the Japanese Industrial Standards collation.
HOST	EBCDIC for IBM channel-attached clients. ASCII for all other clients. This is the default collation sequence.

SET SESSION COLLATION Is Not Valid for 2PC Session Collation

SET SESSION COLLATION is not valid in 2PC sessions.

The collation for a session determines the ordering of data characters during comparison operations and when sorting data in response to a SELECT request that includes a WITH...BY or ORDER BY clause.

Defining Collation

COLLATION can be defined as an attribute of the user (see also "CREATE USER" on page 1-296 and "MODIFY USER" on page 1-364).

If the name is not specified, then the collation for the session defaults to that of the logon client system.

The SET SESSION COLLATION statement defines collation after a session is started. Use the HELP SESSION statement to determine the default collation in effect for the current session.

MULTINATIONAL

The MULTINATIONAL option is useful only on a European Feature Site or a Japanese language site.

MULTINATIONAL sets the collation for the session to an international sort sequence that is compatible with either a diacritical character set or a Japanese character set, as described under the ORDER BY clause of the SELECT statement.

Each character in this collation is assigned two integer values. The first indicates the equivalence class of the character and the second indicates the ordering of the character within that class. For example, the letters LATIN CAPITAL LETTER A (Latin A) and LATIN CAPITAL LETTER A WITH GRAVE (Latin À) might belong to the same class, but are ordered differently.

The general ordering follows UNICODE ordering: Latin A precedes Latin B, and so on.

Single byte MULTINATIONAL collations, such as the Scandinavian languages and other user-installed collations, are extended to include the UNICODE repertoire. The remainder of the UNICODE character repertoire collates the same as the default MULTINATIONAL collation.

When you apply MULTINATIONAL to KANJISJIS, GRAPHIC, or LATIN character data types, the result is as if the data were first converted to Unicode and the appropriate MULTINATIONAL ordering applied.

Do not specify MULTINATIONAL collation for the KANJI1 character set. MULTINATIONAL collates KANJI1 data with only a single level comparison rather than the proper two-level comparison.

Do not specify MULTINATIONAL collation for the Katakana EBCDIC, KanjiEBCDIC5035_I, or KanjiEBCDIC5026_0I character data types either because the definitions of MULTINATIONAL collation designed for KANJI1 data also apply to non-KANJI1 data, and the results are unpredictable. Instead, use CHARSET_COLL for sessions with these character sets.

Each international sort sequence is defined by the database administrator (and no check is made to ensure that collation is compatible with the character set of the current session) but multibyte character collation sequences cannot be changed.

CHARSET_COLL

The CHARSET_COLL collation performs binary ordering based on the current client character set. Strings are compared byte-by-byte.

When one comparison string is shorter than the other, it is padded with the pad character defined for the character data type before the comparison is made.

When comparisons are not case specific, the following behavior occurs.

Stage	Process
1	Lowercase letters are mapped to their uppercase counterparts.
2	The strings are compared. **IF the strings are …** / **THEN the …** identical / equality relation holds. not identical / first pair of bytes that is not equal determines the collating sequence.

When string comparisons involve one or more characters outside the current client character set, then the following behavior occurs.

Stage	Process
1	The strings are compared.
2	Characters are checked to determine if they are all within the repertoire of the current client character set. **IF the characters compared have this relationship to the client character set …** / **THEN …** both are in / the binary ordering of the two characters in the client form-of-use becomes the ordering of the two strings. one is not in / the error character for the character set is used as the collation point for that character. neither is in / the binary ordering of the characters, either case blind or case specific, as appropriate) in the Unicode form-of-use becomes the ordering of the two strings.

CHARSET_COLL string sorts by character data type behave as follows.

FOR this character data type ...	CHARSET_COLL collation orders characters as follows ...
Kanji1	**1** Single byte characters based on the current character set. **2** Multibyte characters based on their internal value. **3** KANJI1 character data types can contain mixed single byte/multibyte character sets. **4** Single byte characters in KANJI1 are translated into the form-of-use. **5** Multibyte characters in KANJI1 are not translated.
KanjiEBCDIC KanjiShift-JIS	As a binary sort on the client would be ordered.
KanjiEUC	As a Kanji Phase I ASCII collation. The difference with a binary sort on the client is that JIS X 0208 characters sort before, rather than after, JIS X 0212 characters.

You can specify CHARSET_COLL as the default user collation in CREATE USER (see "CREATE USER" on page 1-296) or in MODIFY USER (see "MODIFY USER" on page 1-364) if the user definition already exists and you wish to change it.

SET SESSION COLLATION CHARSET_COLL overrides any such user default definitions for the duration of the session in which it is invoked.

JIS_COLL

Do not specify JIS_COLL collation for the KANJI1 character set.

To Sort and Compare, Characters Are Converted to Uppercase

Teradata converts data characters to their uppercase values for comparison and sorting operations unless the CASESPECIFIC option is included in the SQL request, or was defined at creation time for the column being queried.

Character Data Sorted in Ascending Order by Default

Character data is sorted in ascending order unless the DESC (descending) option is included in the SQL request. The results of CASESPECIFIC on sorted results is discussed under the topic "ORDER BY Clause" in Chapter 1 – "The Select Statement" in *Teradata RDBMS SQL Reference, Volume 6*.

Example 1

A session can be set to ASCII collation as follows:

```
SET SESSION COLLATION ASCII;
```

Example 2

Assuming that JobRept contains alphanumeric data, a SELECT that requests a sorted return, as follows:

```
SELECT EmpNo, JobId
FROM JobRept
ORDER BY JobId;
```

produces, in ascending order:

```
EmpNo       JobId
------    --------
10201      1001-AP2
10201      1032-AR3
10201      1031-AR2
10004      ENG-0002
10016      ENG-0002
10004      ENG-0003
10003      OE1-0001
10001      PAY-0002
```

Example 3

If the session is then set to EBCDIC collation:

```
SET SESSION COLLATION EBCDIC;
```

the same request returns:

```
EmpNo       JobId
------    --------
10004      ENG-000
10016      ENG-0002
10004      ENG-0003
10003      OE1-0001
10001      PAY-0002
10201      1001-AP2
10201      1032-AR3
10201      1031-AR2
```

Example 4

If the descending option is added, as follows:

```
SELECT EmpNo, JobId
FROM JobRept
ORDER BY JobId DESC;
```

the returned response set is the following.

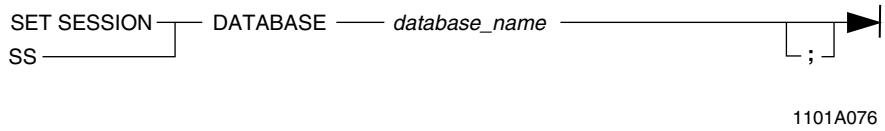
```
EmpNo       JobId
------    --------
10201      1031-AR2
10201      1032-AR3
10201      1001-AP2
10001      PAY-0002
10003      OE1-0001
10004      ENG-0003
10004      ENG-0002
10016      ENG-0002
```

SET SESSION DATABASE

Purpose

Changes the default database for the session.

Syntax



where:

Syntax Element …	Sets the default DATE format for the session to …
database_name	the name of the new default database for the session.

Session Database and Called Procedures

Stored procedures retain the default database that was in effect when they were created. They do not reflect the default database in effect for the session in which they are called unless it is the same as the database in effect when the procedure was created.

Example

Assume your default database is personnel. The following example shows how to change the default database to accounting for the current session.

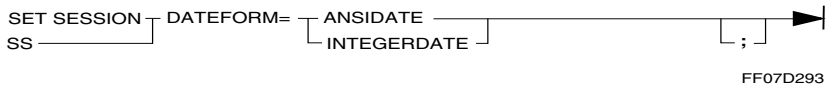
```
SET SESSION DATABASE = accounting;
```

SET SESSION DATEFORM

Purpose

Changes the default DATE format in field mode, and the default format for importing and exporting DATE values for the session.

Syntax



where:

Syntax Element ...	Sets the default DATE format for the session to ...
ANSIDATE	CHARACTER(10) for importing and exporting DATE values. ANSIDATE results in a 'YYYY-MM-DD' date format for date columns created and for date constants in character form.
INTEGERDATE	encoded integers for importing and exporting DATE values. When the session DATEFORM is INTEGERDATE, the default DATE format in field mode is 'YY/MM/DD' for date columns created and date constants in character form. You can use the tdlocaledef utility to change the 'YY/MM/DD' default DATE format. For details, see *Teradata RDBMS Utilities*.

Session Dateform and Called Procedures

Stored procedures retain the dateform (date and time formats) that was in effect when they were created. They do not reflect the dateform in effect for the session in which they are called unless it is the same as the dateform in effect when the procedure was created.

Examples

The following examples show how to change the DATEFORM for a session.

Example 1

You need to change the DATEFORM setting for the current session from ANSI mode to Teradata mode so you can run a legacy application that does not support ANSI dates.

```
SET SESSION DATEFORM = INTEGERDATE;
```

Example 2

After you finish running the job in the previous example, you need to change the DATEFORM setting back to ANSIDATE.

```
SET SESSION DATEFORM = ANSIDATE;
```

SET TIME ZONE

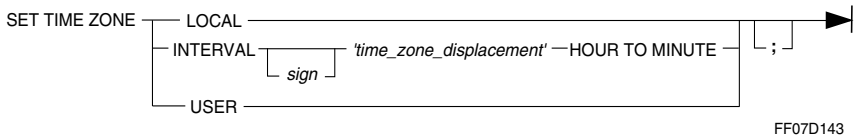
Purpose

Changes the default time zone displacement value for a session.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
LOCAL	change the time zone offset for the session to the system default.
INTERVAL HOUR TO MINUTE	change the time zone offset (based on the 24-hour UTC clock) for the session to an explicit (and optionally signed) time value. The value of the offset is defined by *time_zone_displacement*.
sign	a negative time displacement, represented by a minus (-). The default sign for the quotestring is positive.
'*time_zone_displacement*'	the interval offset required to convert the local or system-defined time to UTC (Coordinated Universal Time). The format is hh:mm.
USER	change the time zone offset for the session back to the defined user default from whatever it had been changed to. If no default is defined for the user, then the value for the time zone offset is changed to the system default.

ANSI Compliance

SET TIME ZONE is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Setting the Permanent Default Time Zone

You set the system default time zone using the DBSControl utility. See *Teradata RDBMS Utilities* for information on using DBSControl to set the default time zone for your site.

UTC Definition

UTC represents the Coordinated Universal Time, formerly referred to as Greenwich Mean Time.

Formally, the value for a local time with respect to UTC is defined as follows.

```
UTC  =  local time - 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.

For example, ignoring regional complications arising from Daylight Savings and related issues, the value for *time_zone_displacement* for Los Angeles would be -8 hours, while the value for New York would be -5 hours.

Rules

The range of time zone displacements permissible is -12:59 through +13.00 inclusive.

The form for time zone displacement values is always INTERVAL HOUR TO MINUTE.

Chapter 2:

SQL Data Control Language Statement Syntax

This chapter documents the Teradata SQL data control language statements. DCL statements grant and revoke access to database objects and change ownership of those objects from one user or database to another.

Experienced SQL users can also see simplified DCL statement descriptions in the chapter "SQL Data Control Language Statement Syntax" in *Teradata RDBMS SQL/Data Dictionary Quick Reference*.

Teradata data definition language (DDL) statements are described in this volume in Chapter 1: "SQL Data Definition Language Statement Syntax."

Teradata data manipulation language (DML) statements are described in *Teradata RDBMS SQL Reference, Volume 6.*

This chapter describes the following topics and statements:

- "Access Privileges" on page 2-2
- "Owners, Creators, and Users" on page 2-4
- "Privilege Types" on page 2-5
- "GIVE" on page 2-11
- "GRANT (Monitor Form)" on page 2-15
- "GRANT (Role Form)" on page 2-19
- "GRANT (SQL Form)" on page 2-22
- "GRANT LOGON" on page 2-43
- "REVOKE (MONITOR Form)" on page 2-49
- "REVOKE (Role Form)" on page 2-54
- "REVOKE (SQL Form)" on page 2-57
- "REVOKE LOGON" on page 2-70

Access Privileges

Introduction

DCL statements grant, transfer, and revoke the ability of users, databases, and roles to perform the following actions:

- Monitor system-wide performance
- Perform particular statements and access particular database objects
- Hold membership in a role
- Log onto Teradata.

These permissions are called access privileges. Access privileges are sometimes referred to as access rights or authorizations.

The following table lists the sets of DCL statements and describes their basic functions:

This DCL statement type …	Performs this function with access privileges …
GRANT	assigns them to specified users, databases, and roles.
• GRANT (Monitor Form)	grants system-wide performance monitoring privileges.
• GRANT (Role Form)	grants role membership to a user or role.
• GRANT (SQL Form)	grants selected or all privileges to a user or database on a specified database object.
• GRANT LOGON	grants privilege to log onto Teradata to a user and changes current system logon defaults.
REVOKE	rescinds them from specified users, databases, and roles.
• REVOKE (Monitor Form)	revokes system-wide performance monitoring privileges.
• REVOKE (Role Form)	revokes membership in a role from a user or role.
• REVOKE (SQL Form)	revokes selected or all privileges from a user or database on a specified database object.
• REVOKE LOGON	revokes the Teradata logon privilege from a user.
GIVE	transfers them from one user or database to another user or database. GIVE does not transfer privileges for performance monitoring, roles, or logons.

Storage of Access Privileges

With the exception of implicit privileges, Teradata access privileges are stored in the DBC.AccessRights table in the data dictionary.

Implicit privileges are not stored.

Users and Databases

The difference between users and databases in Teradata has important implications for matters related to access privileges, but neither the differences nor their implications are easy to understand. This is particularly true with respect to understanding fully the consequences of implicitly granted access privileges.

Formally speaking, the difference between a Teradata user and a Teradata database is that a user has a password and a database does not[1] (see *Teradata RDBMS Database Administration* for details). You might infer from this that databases are passive objects, while users are active objects. That is only true in the sense that databases cannot perform SQL statements. With respect to the semantics of access privileges, databases are not entirely passive (see "Implicit Privileges" on page 2-6 for an example of this).

1 Users can also have default attributes such as time zone, date form, character set, role, and profile, while databases cannot.

Owners, Creators, and Users

Introduction

With respect to access privileges on database objects, Teradata users are categorized as one or more of the following types:

- Owner
- Creator
- User

Depending on the context, a user can be one, several, or all of these user types simultaneously.

User Type	Definition
Owner	Databases and users owns their own space and all space that is derived from it. Ownership extends downward throughout the user/database space hierarchy. For example, user DBC owns all database objects in a Teradata system because all user space derives from DBC. As a result, a database object can have multiple users. A database or user from whose space a database object is created is called the immediate owner of that object. Owners can grant or revoke privileges to themselves on any objects they own. Owners implicitly hold the WITH GRANT OPTION on the objects they own. See the scenario documented in "Access Privileges and the Accessibility of Data By Means of a View, Macro, or Stored Procedure" on page 2-9 for an important consequence of this. See *Teradata RDBMS Database Administration* for a detailed description of ownership in Teradata.
Creator	A user who performs the DDL to create a database object is called the creator of that object. Creators are automatically granted all privileges on the database objects they create. A database cannot be a creator.
User	A user who is neither the owner nor the creator of the database objects with which he works is called a user. User access privileges on database objects must be granted explicitly. A database cannot be a user.

It is important to understand the differences among these user types because the accrual of privileges is different for each of them.

Privilege Types

Introduction

Teradata supports four general types of access privileges categorized by the way they are granted:

- Implicit
- Automatic
- Explicit
- Inherited

The following table summarizes the differences among these privilege types:

Privilege Type	Description
Implicit	The set of privileges granted to the owner of a database object. The privilege to grant themselves any privileges or any database, user, or database object they own is implicitly granted to an owning user or database by default. Implicit privileges are not stored in DBC.AccessRights. The privilege to grant and re-grant privileges on any owned object cannot be revoked directly, but can be transferred to another user or database by means of the GRANT statement. Implicit privileges are usually not sufficient for owners to access objects they own directly. They must also have the privileges, whether granted automatically or explicitly, to access those objects. See "Implicit Privileges" on page 2-6 for details. Implicit privileges are sometimes called ownership privileges.
Automatic	The set of privileges automatically granted on a database object to the creator of that object. These access privileges also accrue automatically to objects created by the user holding them. Automatic privileges are inserted into DBC.AccessRights by the CREATE statement that defines the database object. They can be revoked by any user with the authority to do so.
Explicit	The set of privileges assigned by a user higher in the privilege hierarchy to a lesser privileged user. Owners can also grant privileges to and revoke privileges from themselves on any objects they own. Explicit privileges are inserted into DBC.AccessRights by a GRANT statement that explicitly specifies a set of privileges on a database object. Explicit privileges can be revoked by any user with the authority to do so.

Privilege Type	Description
Inherited	Users assigned to a role inherit all the privileges granted to that role. Inherited privileges are a subclass of explicit privileges because a set of privileges is granted explicitly to each role after it is created (see "GRANT (SQL Form)" on page 2-22 for an explanation of how privileges are assigned to roles). Because they are actually explicit privileges, inherited privileges are stored in DBC.AccessRights and can be revoked by any user with the power to remove a user from membership in the role from which the privileges are inherited.

Implicit Privileges

Implicit privileges are sometimes called ownership privileges. They are privileges on an object that accrue to a database or user because they own that object, either directly or indirectly.

Unlike automatic and explicit privileges, implicit privileges are not recorded in the DBC.AccessRights table.

Also unlike automatic, explicit, and inherited privileges, implicit privileges cannot be revoked.

Even though owners have implicit privileges on their objects, they must have explicitly granted rights on most objects to access them directly. For example, an owner of a table must have the explicitly or automatically granted SELECT privilege on that table to be able to select rows from it.

Owners can grant themselves these rights on objects they own. For example, suppose I own a table named StatusThroughYesterday that I want to make read-only. Because I own StatusThroughYesterday, I can revoke UPDATE and DELETE privileges on it from all users, including myself. When I need to update StatusThroughYesterday at the end of the current day, I can grant the UPDATE and DELETE privileges back to myself, make the necessary updates, and then revoke those privileges once again.

Automatic Privileges

Automatic privileges are privileges on an object that accrue automatically to a database or user automatically when the object is created.

Automatic privileges are recorded in the DBC.AccessRights table.

These privileges can be revoked at any time by any user with the authority to do so.

Explicit Privileges

Explicit privileges are privileges that are granted to a user or role on an object using the SQL GRANT statement (see "GRANT (SQL Form)" on page 2-22).

Explicit privileges are recorded in the DBC.AccessRights table.

These privileges can be revoked at any time by any user with the authority to do so.

Explicit and automatic privileges are sometimes grouped together as explicit privileges because Teradata does not discriminate between them once they have been granted. The DBC.AccessRights rows for privileges do not record how they were granted.

Inherited Privileges

Inherited privileges are privileges that are passively granted to users by means of the role in which they are a member.

Note that both implicit and automatic privileges can be inherited. Users also inherit privileges explicitly when they are created from a user or database defined with `GRANT privilege ON object TO ALL`. As used here, the term inherited refers to the access privileges users assume because of their membership in a role.

Inherited privileges are recorded in the DBC.AccessRights table.

These privileges can be revoked at any time by any user with the privilege to perform the REVOKE (Role Form) statement on the role of which the user is a member (see "REVOKE (Role Form)" on page 2-54). This action removes specified users from membership in a role, thereby indirectly revoking the privileges they inherited from that membership.

Summary of Implicit, Automatic, Explicit, and Inherited Privilege Types

The following table summarizes this information:

Privilege Type	Stored in This Table	Revokable?	Granted To This Type of User
Implicit	Not recorded	No	Owner
Automatic	DBC.AccessRights	Yes	Creator
Explicit	DBC.AccessRights	Yes	User
Inherited	DBC.AccessRights	Yes	Role[a]

a. Roles are not users, but in this context are analogous to surrogate users.

System-Granted Privileges

You might see the term system-granted privileges used occasionally. This is not standard terminology, because these privileges are granted explicitly by a GRANT statement in a DIP script (see "Explicit Privileges" on page 2-6 for more information about explicit privileges).

Granting Privileges to Others

SQL DCL permits you to grant any privilege you own to another user if you are one of the following users:

- User DBC
- A user to whom user DBC has granted privileges
- The owner of the object
- Someone having WITH GRANT OPTION for the privilege to be granted plus all of the privileges that are to be granted on the object.

Conceptually, all access privileges are granted in the form of the following pseudocode:

```
GRANT privilege
ON object
TO user/database/role
WITH[OUT] GRANT OPTION[2]
```

This is true whether the privileges are granted implicitly, automatically, or explicitly.

The determination of whether a user can grant the ability to grant privileges to other users depends on whether how the WITH[OUT] GRANT OPTION clause is specified in the SQL GRANT statement.

IF you specify ...	THEN users ...
WITH GRANT OPTION[2]	can grant any privilege they own to another user.
nothing	cannot grant any privilege they own to another user.

2 For roles, the pseudocode for this clause is WITH[OUT] ADMIN OPTION.

Access Privileges and the Accessibility of Data By Means of a View, Macro, or Stored Procedure

The access privileges required for users to access data by means of a view, macro, or stored procedure are different than the access privileges required to access the same data directly.

Consider the following example scenario:

- You define a view v in database d2.
- The base tables referenced by v are also contained within d2.
- Any user that performs a SELECT statement through v must have the SELECT access privilege on v.
- To permit a user to perform this SELECT statement, d2 must also have the SELECT WITH GRANT OPTION access privilege on the base tables the view references.
- You revoke all explicit access privileges from d2 by performing the following REVOKE statement:

  ```
  REVOKE ALL PRIVILEGES ON d2 FROM PUBLIC;
  ```

Can users access the tables in d2 using v?

The answer is yes if they are accessing those tables by means of a view, macro, or stored procedure. In this case, d2 is accessed through v, so when a user with the SELECT privilege on v performs the SELECT … FROM d2.v statement, Teradata returns the requested rows.

The reason for this is that d2 owns all the views and tables it contains. Because d2 is the owner of these views and tables, it implicitly has the SELECT WITH GRANT OPTION access privilege on them, and implicitly granted privileges cannot be revoked. Teradata takes advantage of this to permit users to access data indirectly through views, macros, or stored procedures.

Before users can access a database object directly, they must possess explicit access privileges on that object. Direct access to database objects can be controlled tightly by granting or revoking user access privileges selectively to either minimize or eliminate it entirely. This ensures that users can neither access data they should not view or update, nor can they do something to an object that would compromise its integrity or the integrity of the database.

The situation is different when users access a database object indirectly by means of a view, macro, or stored procedure. These objects are designed specifically to circumvent direct user access of database objects while still providing them with a method to access selected aspects of the data they contain.

To support this scenario, Teradata permits a DBA to revoke all privileges on a base database object, whether granted explicitly or automatically, to ensure maximum security. At the same time, the immediate owner of a view, macro, or stored procedure implicitly and irrevocably possesses all privileges on those objects. This allows Teradata to permit users of views, macros, and stored procedures to access data indirectly using the implicit access privileges granted to the immediate owner of those objects.

In this case, it is the implicit access privileges held by the immediate owner of the view, macro, or stored procedure that control whether a user can perform an action using that view, macro, or stored procedure to access data. The explicitly held or automatically assigned access privileges the user has on those objects are not relevant.

GIVE

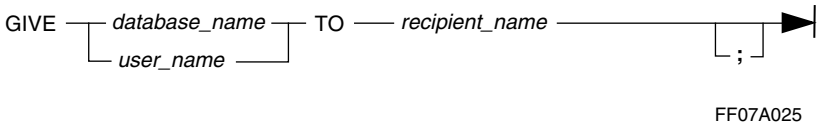
Purpose

Transfers ownership of a database or user space to another user. Also transfers all databases and users owned by the transferred database or user.

Invocation

Executable.

Syntax



FF07A025

where:

Syntax Element...	Specifies...
database_name *user_name*	the name of the database or user whose ownership is being transferred.
TO	an introduction to the name of the recipient.
recipient_name	the name of the new immediate owner for the transferred database or user.

ANSI Compliance

GIVE is an extension to the ANSI SQL-99 standard.

Authorization

You must have DROP DATABASE privileges on the given object, and CREATE DATABASE privileges on the recipient.

The GIVE statement does not revoke any explicit access privileges that the old ownership hierarchy had on the given database or user. No privileges on the given database or user are granted to the new ownership hierarchy as a result of the GIVE statement, nor does the database or user being given receive any explicit privileges.

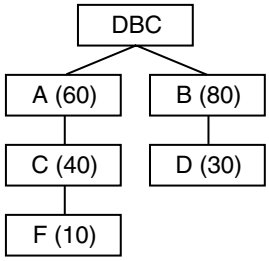
The recipient of a GIVE statement cannot be owned by the given object; if A owns B, A cannot be given to B.

Transfer of Space Allocation

A transfer of ownership also transfers the space allocated to the named database or user; this affects space allocation in Teradata as follows:

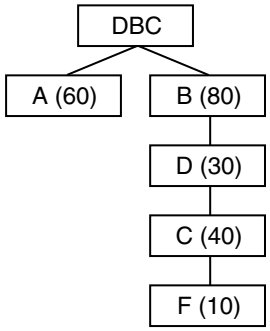
- The aggregate number of bytes available to the former owners (the owner who is entering the GIVE statement plus the owners above this owner in the hierarchy) is reduced by the number of bytes in the transferred database. This includes total space allocated to the database, plus that of all databases and users owned by the transferred database.
- If the transferred database is dropped, then the number of bytes allocated to the new immediate owner of the transferred database is increased by the number of bytes in the transferred database. In addition, the aggregate number of bytes available to owners above this owner in the hierarchy is increased by the number of bytes that had been allocated to the transferred database.

For example, consider the following hierarchy:



FF07A070

If ownership of database C is transferred to database D, the structure of the hierarchy changes as follows:



FF07A071

When database C is transferred, database F (which is owned by C) is also transferred.

- While the number or bytes allocated to database A remains the same, the aggregate number of bytes available to database A is reduced by 50 (the total number of bytes allocated to databases C and F).
- The number of bytes allocated to databases
- D and B remains the same.
- The available number of bytes, however, is increased by 50 (that is, if databases C and F were dropped, the bytes allocated to C and F are transferred to database D).

Example

The following statement transfers ownership of the Finance database from user Administrator to user Chin.

```
GIVE Finance TO Chin;
```

GRANT

GRANT establishes database privileges for one or more users.

GRANT has four forms that differ both in function and in syntax:

- "GRANT (Monitor Form)" on page 2-15, for performance monitoring of Teradata.
- "GRANT (Role Form)" on page 2-19, for granting role membership to users and other roles.
- "GRANT (SQL Form)" on page 2-22, pertaining to access control Teradata database objects.
- "GRANT LOGON" on page 2-43, for granting system-wide performance monitoring privileges.

The SQL and MONITOR forms of GRANT are separate functions.

To grant a user all privileges including MONITOR, you must perform both of the following statements:

```
GRANT ALL PRIVILEGES ON object
TO user
WITH GRANT OPTION;

GRANT MONITOR PRIVILEGES
TO user
WITH GRANT OPTION;
```

Note that GRANT MONITOR has no ON object clause. Because this statement allows its users to impact the entire system, there is an implicit ON PUBLIC clause in each statement issued.

GRANT (Monitor Form)

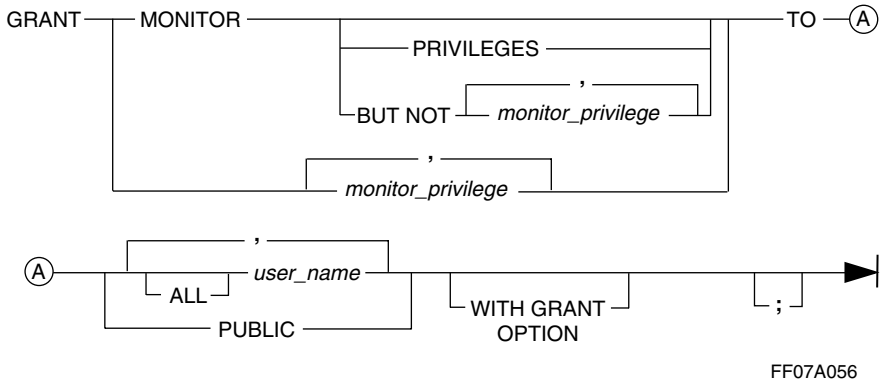
Purpose

Grants system-wide performance monitoring privileges.

Invocation

Executable.

Syntax



where:

Syntax Element . . .	Specifies . . .
MONITOR PRIVILEGES	the named recipients are to receive all MONITOR-related privileges. MONITOR [PRIVILEGES] does not permit the user to grant the indicated privilege to others without the WITH GRANT OPTION being specified.
BUT NOT	the named recipients are to receive all of the above grantable privileges except those specified after BUT NOT. If the ability to grant these privileges is to be included, the WITH GRANT OPTION must be specified explicitly.
ABORTSESSION	to abort any outstanding request or ongoing transaction of one or more Teradata sessions and, optionally, logs off the sessions.
MONRESOURCE	to gather information on the performance and availability of each PE and AMP.
MONSESSION	to gather information about logged on sessions and overall system usage on a session-by-session basis.
SETRESRATE	to set the frequency at which processor resource usage data is updated in Teradata.

Syntax Element . . .	Specifies . . .
SETSESSRATE	to set the frequency at which session-level performance data is updated in Teradata.
WITH GRANT OPTION	that the recipient will receive the listed privileges WITH GRANT OPTION.
ALL	to grant the specified object privilege set to the named database or user and to every database or user owned by that database or user now and in the future. If you specify ALL, then the privileges are granted to the named database or user and to every database or user owned by that database or user now and in the future. ALL DBC is equivalent to PUBLIC.
user_name	the name of a user or database to be granted the specified MONITOR privileges. *user_name* must be the identifier of a user already defined to Teradata. Up to 25 names can be specified.
PUBLIC	that the privileges are to be granted to all existing and future Teradata users.
WITH GRANT OPTION	that the grantee receives privileges WITH GRANT OPTION.

ANSI Compliance

The monitor form of GRANT is an extension to the ANSI SQL-99 standard.

Authorization

You must have MONITOR privileges to use the monitor form of GRANT.

Be very selective about granting MONITOR access privileges. These privileges should be granted only to those users who are cleared to monitor all applications on all sessions**.**

There is no lower level of MONITOR privileges. For example, the database administrator cannot grant user Addams the ability to do session-level monitoring of her applications only. Instead, the database administrator would have to grant Addams the permission to do session-level monitoring of all applications by all sessions.

To determine who is currently using the MONITOR partition, issue the following query:

```
SELECT UserName, IFPNo
FROM DBC.SessionInfo
WHERE Partition = 'MONITOR'
```

The GRANT statement is used only to assign specific privileges. To transfer ownership of a database or user, see "GIVE" on page 2-11.

GRANT and GRANT MONITOR

The SQL and MONITOR forms of GRANT are separate statements. To grant a user all privileges, including MONITOR, the grantor must perform *both* statements, as in the following example.

```
GRANT ALL PRIVILEGES ON object
TO user_name
WITH GRANT OPTION;

GRANT MONITOR PRIVILEGES
TO user_name
WITH GRANT OPTION;
```

Never specify the WITH GRANT OPTION if you do not want the recipient of the privilege to be able to grant those privileges to others.

ALL PRIVILEGES refers only to database-related privileges. MONITOR PRIVILEGES indicates all monitoring-related privileges.

GRANT Versus GRANT MONITOR

A major difference between the syntax of the GRANT and GRANT MONITOR privileges is that GRANT MONITOR has no clause that specifies what the access privilege has been granted on. Because GRANT MONITOR allows a user to impact the entire system, the permissions are implicitly ON PUBLIC.

It is illegal to specify an ON *object* clause in any GRANT MONITOR statement. However, it is illegal *not* to specify an ON *object* clause in any other GRANT statement. If you violate either one of these restrictions, a failure response returns to the requestor.

The two sets of GRANT privileges differ in function as well as in syntax. The GRANT privilege relates to controlling access to Teradata database objects. The GRANT MONITOR privilege relates to monitoring system-wide Teradata performance.

GRANT takes effect immediately when the grantee issues his next statement. It is not necessary to log out to receive the privilege just granted.

Privileges Granted Immediately

GRANT MONITOR takes effect immediately when the grantee issues his next statement. It is unnecessary to log out to receive the monitor privilege just granted.

Granting MONITOR Privileges

Use the WITH GRANT OPTION clause carefully. WITH GRANT OPTION allows the recipient to grant any MONITOR privilege to anyone. The recipient could grant himself privileges he was not intended to have, and subsequently grant any MONITOR privilege to anyone else without restriction.

For example, user DBC creates user Smith and grants Smith ABORTSESSION and MONSESSION privileges. User DBC then grants Smith the SETSESRATE privilege with the WITH GRANT OPTION clause. Smith can now grant not only the SETSESRATE, but also the ABORTSESSION and MONSESSION privileges. Smith retains these privileges until user DBC revokes the ability of Smith to GRANT any of these MONITOR privileges.

Exercise caution when granting MONITOR access privileges. This privilege should be granted only to those users who are cleared to monitor all applications on all sessions. There is no lower level of MONITOR privileges. For example, a database administrator cannot grant user Addams the ability to do session-level monitoring of her applications only. Instead, the DBA would have to grant Addams the permission to do session-level monitoring of all applications by all sessions.

To determine who is currently using the MONITOR partition, perform the following query.

```
SELECT UserName, IFPNo
FROM DBC.SessionInfo
WHERE Partition = 'MONITOR'
```

Security Logging of Access Attempts

If you need to maintain a security log of access attempts, see “BEGIN LOGGING” on page 1-67.

Related Statement

See “REVOKE (MONITOR Form)” on page 2-49 for information about revoking privileges granted by the Monitor form of GRANT.

GRANT (Role Form)

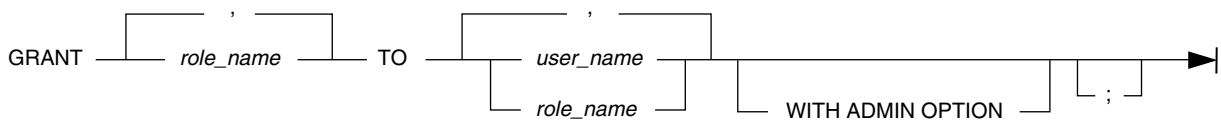
Purpose

Grants role membership to users or other roles.

Invocation

Executable.

Syntax



KZ01a008

where:

Syntax Element …	Specifies …
role_name	one or more comma-separated names of roles to which members are being added. Teradata ignores duplicate role names.
TO *user_name* *role_name*	the names of role grantees. Grantees can be users or roles; however, a role cannot be granted membership to itself. GRANT does not produce an error if a grantee is already a member of a specified role.
WITH ADMIN OPTION	that the role grantees have the privilege to use DROP ROLE, GRANT, and REVOKE statements to administer the roles to which they are becoming members. A GRANT statement that does not include WITH ADMIN OPTION does not revoke a previously granted WITH ADMIN OPTION privilege from grantee.

ANSI Compliance

GRANT is ANSI SQL-99-compliant.

Authorization

To grant membership to a role, you must have the WITH ADMIN OPTION privilege on the role. The following users can grant membership to a role:

- User DBC.
- A user who is a member of the specified role WITH ADMIN OPTION.

  The creator of a role automatically becomes a member of the specified role WITH ADMIN OPTION.
- A user who has an *active* role that is a member of the specified role WITH ADMIN OPTION. An active role can be the current role or a role that is a member of the current role.

Usage

Roles define access privileges on database objects. A user that is assigned a role can access all the objects on which the role and its role members have privileges. A user can only be assigned a role to which the user has been granted membership.

You can grant membership to a newly created role before the role has privileges on any database objects.

A role or user can be granted membership to an unlimited number of roles.

Role Hierarchy

Roles can only be nested one level. Thus, a role that has a role member cannot also be a member of another role.

This is a deviation from ANSI, which allows multiple nesting levels.

Example

The following statements create roles called services and sales:

```
CREATE ROLE services;
CREATE ROLE sales;
```

To make services a member of sales, use the following GRANT statement:

```
GRANT sales TO services;
```

To grant user marks membership to sales, and give marks the privilege to add other members to sales, use the following GRANT statement:

```
GRANT sales TO marks WITH ADMIN OPTION;
```

For More Information

FOR more information on …	SEE …
granting privileges on database objects to roles	"GRANT (SQL Form)" on page 2-22.
obtaining the CREATE ROLE system privilege	
assigning default roles to users	• "CREATE USER" on page 1-296. • "MODIFY USER" on page 1-364.
changing the current role for a session	"SET ROLE" on page 1-382.

Related Statement

See "REVOKE (Role Form)" on page 2-54 for information about revoking privileges granted by the Role form of GRANT.

GRANT (SQL Form)

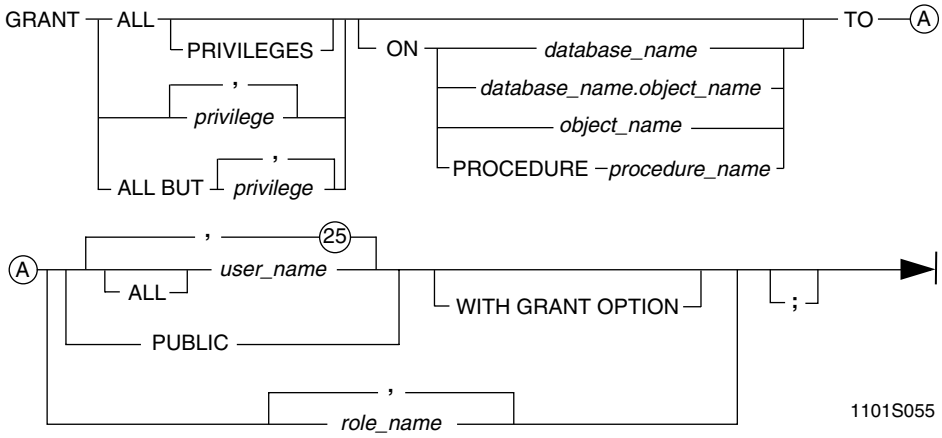
Purpose

Assigns one or more privileges on a database, user, table, hash index, join index, view, stored procedure, or macro to a role, group of roles, user, or group of users.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
ALL PRIVILEGES	that the designated user or role is to receive all privileges that can be granted on the specified object. To include the ability for the designated user to GRANT object privileges to other users, use the WITH GRANT OPTION clause. GRANT ALL means that ALL object privileges owned by the grantor, WITH GRANT OPTION, that pertain to the type of object, are granted. An error is returned if the grantor has no privileges WITH GRANT OPTION on the object. You must use the monitor form of the GRANT statement to grant monitor privileges. See "GRANT (Monitor Form)" on page 2-15.

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>privilege</td><td>one of the following access or execution privileges:

ALTER PROCEDURE
CHECKPOINT
CREATE DATABASE
CREATE MACRO
CREATE PROCEDURE
CREATE PROFILE
CREATE ROLE
CREATE TABLE
CREATE TRIGGER
CREATE USER
CREATE VIEW
DELETE
DROP
DROP DATABASE
DROP MACRO
DROP PROCEDURE
DROP PROFILE
DROP ROLE
DROP TABLE
DROP TRIGGER
DROP USER
DROP VIEW
DUMP
EXECUTE
EXECUTE PROCEDURE
INDEX
INSERT
REFERENCES column_list
REFERENCES ALL BUT ... column_list
RESTORE
SELECT
TRIGGER
UPDATE column_list[a]
UPDATE ALL BUT … column_list

where the term column_list indicates a parenthetically enclosed set of comma-separated column names to which the specified privilege applies.
<table>
<tr><th>OR specify this privilege …</th><th>TO grant all of these privileges …</th></tr>
<tr><td>DATABASE</td><td>• CREATE DATABASE
• DROP DATABASE</td></tr>
<tr><td>MACRO</td><td>• CREATE MACRO
• DROP MACRO</td></tr>
<tr><td>PROCEDURE</td><td>• CREATE PROCEDURE
• DROP PROCEDURE</td></tr>
<tr><td>PROFILE</td><td>• CREATE PROFILE
• DROP PROFILE</td></tr>
<tr><td>ROLE</td><td>• CREATE ROLE
• DROP ROLE</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td>privilege
(continued)</td><td>
<table>
<tr><th>OR specify this privilege ...</th><th>TO grant all of these privileges ...</th></tr>
<tr><td>TABLE</td><td>• CREATE TABLE
• DROP TABLE</td></tr>
<tr><td>TRIGGER</td><td>• CREATE TRIGGER
• DROP TRIGGER</td></tr>
<tr><td>USER</td><td>• CREATE USER
• DROP USER</td></tr>
<tr><td>VIEW</td><td>• CREATE VIEW
• DROP VIEW</td></tr>
</table>
Note that UPDATE and REFERENCES have both table- and column-level options. See “REFERENCES” on page 2-29.
The DELETE privilege applies only to the DELETE DML statement, not to DELETE USER or DELETE DATABASE, which are granted by DROP USER and DROP DATABASE.
See also “Rules for privilege Keywords” on page 2-28. However, for a full explanation of the privileges a specific statement requires, see the “Authorization” section of that statement in this manual.
The following privileges do not directly correspond to an SQL statement:
<table>
<tr><th>THIS privilege ...</th><th>REFERS to ...</th></tr>
<tr><td>CHECKPOINT</td><td>both to the execution of the SQL statement and the Host Utilities (HUT) command.</td></tr>
<tr><td>DUMP</td><td>the corresponding HUT command performed on the specified object.</td></tr>
<tr><td>EXECUTE PROCEDURE</td><td>the corresponding CALL statement.</td></tr>
<tr><td>INDEX</td><td>the CREATE and DROP INDEX and COLLECT and DROP STATISTICS statements.</td></tr>
<tr><td>REFERENCES</td><td>columns referenced in a FOREIGN KEY clause of a CREATE or ALTER TABLE statement.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td>privilege
(continued)</td><td>
<table>
<tr><th>THIS privilege ...</th><th>REFERS to ...</th></tr>
<tr><td>RESTORE</td><td>the corresponding HUT command performed on the specified object and to the execution of the following HUT commands:
DELETE JOURNAL
ROLLBACK
ROLLFORWARD</td></tr>
</table>
</td></tr>
<tr><td>ALL BUT</td><td>the named user is to receive all privileges that can be granted on the specified object except for those specified in the privilege list. As in ALL, only those object privileges, that the grantor owns WITH GRANT OPTION are granted.

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

Granting privileges on a database or user is a Teradata extension to the ANSI SQL-99 standard.</td></tr>
<tr><td>PROCEDURE
procedure_name</td><td>that the object is a stored procedure.

PROCEDURE must be specified when the object is a stored procedure. See “Stored Procedure-Specific Privileges” on page 2-36.
<table>
<tr><th>IF you specify PROCEDURE, you can specify ...</th><th>AS privilege to grant this privilege ...</th></tr>
<tr><td>EXECUTE</td><td>EXECUTE PROCEDURE.</td></tr>
<tr><td>DROP</td><td>DROP PROCEDURE.</td></tr>
</table>
</td></tr>
<tr><td>database_name</td><td>the name of a database or user on which the privileges are to be granted. All objects in this database or user space are affected.</td></tr>
<tr><td>database_name.
object_name</td><td>the name of the owning database and the name of the object (table, view, stored procedure, or macro) on which the privileges are to be granted. Only the named object is affected.</td></tr>
</table>

<table>
<tr><th>Syntax Element …</th><th>Specifies …</th></tr>
<tr><td>object_name</td><td>the name of the object (table, view, join index, stored procedure, or macro) on which the privileges are to be granted.

You should always qualify object names when granting privileges because Teradata checks for matching database names before checking for object names.

<table>
<tr><th>IF …</th><th>THEN Teradata …</th></tr>
<tr><td>the object name is not qualified</td><td>assumes it is a database name.</td></tr>
<tr><td>the object name is not qualified and no database having that name is found</td><td>assumes it is an object within the current database.</td></tr>
<tr><td>neither a database nor an object is found</td><td>returns an error to the requestor.</td></tr>
</table>
</td></tr>
<tr><td>ALL user_name</td><td>the name of a database or user that identifies the recipient. user_name must be the identifier of a user already defined to Teradata.

You can specify up to 25 names.

If you specify ALL, then the object privileges are granted to the named database or user and to every database or user owned by that database or user now and in the future.

ALL DBC is equivalent to PUBLIC.

ALL is a Teradata extension to the ANSI SQL-99 standard.</td></tr>
<tr><td>PUBLIC</td><td>that the privileges are to be granted to all existing and future Teradata users.</td></tr>
<tr><td>WITH GRANT OPTION</td><td>that the grantee will receive privileges WITH GRANT OPTION.

This option does not apply to grantees that are roles. A GRANT statement that specifies WITH GRANT OPTION when there is a role in the list of grantees aborts and returns an error message.</td></tr>
<tr><td>role_name</td><td>the name of a role. The role must already be defined to Teradata.

You cannot grant CREATE PROFILE, CREATE ROLE, CREATE USER, DROP PROFILE, DROP ROLE, or DROP USER privileges to roles.</td></tr>
</table>

a. You cannot grant the UPDATE privilege on a GENERATED ALWAYS identity column.

ANSI Compliance

The SQL form of GRANT is ANSI SQL-99-compliant with extensions.

ANSI SQL-99 supports the following privileges.

- DELETE
- INSERT
- SELECT
- REFERENCES
- UPDATE

The other privileges listed are all extensions to the ANSI SQL-99 standard.

Authorization

You must be one of the following to grant privileges on an object using the SQL GRANT statement:

- User DBC
- A user to whom user DBC has granted privileges
- The owner of the object

  There is a subtle security issue associated with this. See "Security Considerations With CREATE MACRO Privilege" on page 2-30 for details.
- A user having WITH GRANT OPTION for the privilege to be granted plus all of the privileges that are to be granted on the object.

Note the following details:

- A user need not be related to a grantor through ownership to receive a privilege.
- If a GRANT statement is at the database- or user-level, the privilege applies to all objects, both current and future, created in that space.

  If a REVOKE statement later removes the privilege, the privilege is dropped for all objects, regardless of when they were created.

  A REVOKE statement at the object level cannot remove a privilege from that object that was granted at the database- or user-level.
- An owner implicitly has the WITH GRANT OPTION privilege on a database, user, or object it owns.

  Therefore, an owner can explicitly grant any or all privileges on a database, user, or object to any other user or role.
- Any privilege granted automatically or explicitly can be revoked using the REVOKE statement. See "REVOKE (SQL Form)" on page 2-57.
- Implicit ownership privileges cannot be revoked.

Rules for *privilege* Keywords

The following rules apply to using the *privilege* keywords.

- Any combination of privileges can be specified. However, the user submitting the statement must have privileges on all of the specified privileges.
- The recipient of a *privilege* can perform the corresponding statement against the object on which the privilege was granted.

  For example, if User_1 receives the CREATE TABLE privilege on database DbTest, User_1 can then perform a CREATE TABLE statement in which the new table is directed to DBTest (where the target database is resolved either implicitly, as determined by the default database for User1, or explicitly with a fully qualified table name.
- The CHECKPOINT privilege refers to the execution of both the SQL statement and the Host Utilities (HUT) command.

  The DUMP and RESTORE privileges refer to the corresponding HUT command performed on the specified object.

  RESTORE also refers to execution of the HUT commands ROLLBACK, ROLLFORWARD, and DELETE JOURNAL.
- CREATE DATABASE, MACRO, TABLE, VIEW, PROCEDURE, or USER, and DROP DATABASE or USER are allowed only on databases or users.
- DROP TABLE includes ALTER TABLE and INDEX.
- DROP MACRO, PROCEDURE, or VIEW includes REPLACE MACRO, PROCEDURE, or VIEW.
- Only DROP MACRO and EXECUTE are allowed on macros.
- Only ALTER PROCEDURE, DROP PROCEDURE and EXECUTE PROCEDURE are allowed on stored procedures.
- DUMP, RESTORE, INDEX, and CHECKPOINT are not allowed on views.
- DATABASE, MACRO, PROCEDURE, PROFILE, ROLE, TABLE, VIEW, and USER confer both CREATE and DROP privileges.

  If the DATABASE, MACRO, PROCEDURE, PROFILE, ROLE, TABLE, USER, or VIEW keyword is specified without CREATE or DROP, both CREATE and DROP are assumed by default.
- INDEX allows CREATE and DROP INDEX and COLLECT and DROP STATISTICS statements on tables. If INDEX is granted on all of the base tables of a join index, INDEX also allows COLLECT and DROP STATISTICS on the join index.
- UPDATE and REFERENCES have both table- and column-level options.
- You cannot grant the UPDATE privilege on a GENERATED BY DEFAULT identity column.
- You *cannot* grant access privileges on a trigger, only on the database or table to which the trigger applies. DROP TRIGGER is granted to either the database or the subject table, not on the trigger itself.

- You can grant CREATE ROLE, DROP ROLE, CREATE PROFILE, and DROP PROFILE privileges to users only, not to roles or databases.
- CREATE ROLE, DROP ROLE, CREATE PROFILE, and DROP PROFILE are system privileges; you grant the privileges to a user, but not on a specific object.

Differences Between GRANT and GIVE

The GRANT statement is used only to assign specific privileges. To transfer ownership of a database or user, use the GIVE statement.

Privileges Granted Immediately

GRANT takes effect immediately when the grantee issues his next statement. You do not need to log out to receive a privilege that was just granted to you.

Object Names

The element *object_name* is the name of a table, join index, hash index, view, stored procedure, or macro. If the form of the privileges option includes a set of column names then *object_name* must be for either a table or view.

If *object_name* is not qualified by either a database name or user name and there is an object with that name both under the current database of the executing user and that of the grantee, then it is assumed that the object is that in the current database of the executing user.

An unqualified *object_name* is considered to be that of the current database if the name is that of the current database and also the name of a table either within the current database or within the database of the grantee.

REFERENCES

The REFERENCES privilege applies at both the table and column level. It is required on all columns referenced, whether implicitly or explicitly, in a FOREIGN KEY clause of a CREATE TABLE or ALTER TABLE statement. Teradata gives this privilege, with grant authority, to the owner and creator of a table.

If you specify a column list (a list of parenthetically enclosed, comma-separated column names, for example `(column_name_1, …, column_name_n)`) with the keywords REFERENCES or UPDATE, and the specified grantee set already has the privilege specified at the table level with the indicated grant authority, then teradata accepts the statement but takes no action. This is because having UPDATE or REFERENCES privileges at table level also enables that action against all columns of the table.

INDEX

The INDEX privilege only applies at table level. You must have either this privilege or DROP TABLE to perform the CREATE or DROP INDEX statements. Teradata implicitly gives this privilege, with grant authority, to the owner of the table.

Column Level Privileges

DBC.AccessRights table stores the field ID of a column on which UPDATE or REFERENCES privilege has been granted. The column type is non-null SMALLINT. If the privilege defined by the row is not at the column level then the value of the column is set to zero.

Column-level privileges are granted implicitly to the owner of a table. As is always true for implicitly granted privileges, a row is not inserted into the DBC.AccessRights table for implicitly-granted column-level privileges.

Rows are also not generated in DBC.AccessRights for individual columns when the UPDATE or REFERENCES privileges are granted at table level.

Rows are generated in DBC.AccessRights only for privileges granted by a GRANT statement at the column level when the grantee does *not* have the privilege at the table level. The only exception is the case of a user who has a privilege at table level but without GRANT authority and who then is granted the same privilege on an individual column of the same table *with* grant authority.

Privileges Required to Perform UPDATE, DELETE, and INSERT

Any user who needs to perform any form of update, delete, or insert operations must be granted SELECT access privileges on all tables from which there is a need to read values.

For the INSERT statement, SELECT privileges are required only if a table is to be reflexively grown by INSERT … SELECT statements performed on itself.

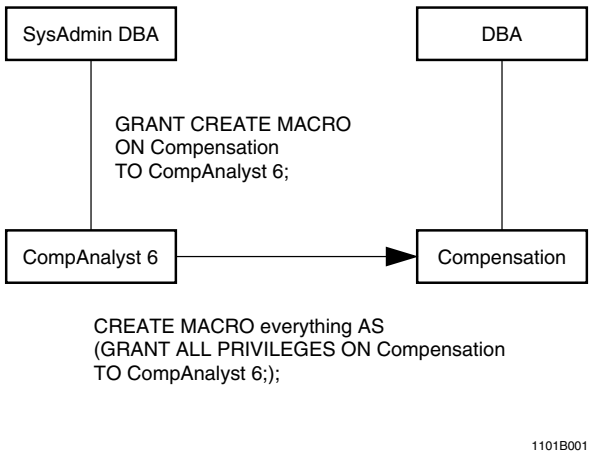
Be careful when granting the delete privilege through a view because data in columns that are not defined in the view (and which might be important to the enterprise) might unsuspectingly be deleted when a row is deleted through a view.

Security Considerations With CREATE MACRO Privilege

Before you grant CREATE MACRO on a database or user, it is extremely important to realize that the recipient of that privilege can create and perform macros that have all the privileges of that database or user. This is because for CREATE MACRO, the access privileges are inherited from the immediate owner of the macro, not from its creator.

As a result, the grantee can create macros that contain DCL and DDL statements that are not checked for the access privileges of the creator. This means that you are implicitly assigning privileges to the macro creator that they have no explicit, implicit, or automatic privilege to perform. This might not be a desirable result and you should be exceedingly careful when granting this privilege.

For example, consider the scenario presented in the following graphic.



The Compensation database is owned by user DBA.

User SysAdminDBA, the system administrator for Compensation, has privileges on Compensation, including CREATE MACRO WITH GRANT OPTION, and on all objects owned by Compensation. SysAdminDBA can also effectively grant himself any of the following.

- Privileges on objects owned by Compensation
- Privileges that Compensation has WITH GRANT OPTION
- Any implicit privileges owned by Compensation

SysAdminDBA creates user CompAnalyst6 for an entry level programmer who has been tasked to produce compensation reports for several routine audits performed by various state and federal regulatory agencies. To ensure that CompAnalyst6 does not have access to critical private employee base salary and bonus information, she has been granted only a restricted set of privileges on objects in the Compensation database.

To make it easier for CompAnalyst6 to create the reports, SysAdminDBA also grants her the CREATE MACRO privilege on Compensation as follows.

```
GRANT CREATE MACRO
ON compensation
TO companalyst6;
```

Because the access privileges for executing macros in Compensation derive from Compensation, CompAnalyst6 can create and perform macros that report on just the sort of private data she was meant to be restricted from viewing.

For example, CompAnalyst6 can grant herself full access to all tables in the database through a simple macro and then create any database object or perform a query that reports on salary and bonus data for each employee in the enterprise in three quick steps.

Step	Action

1	`CREATE MACRO everything AS`
	`(GRANT ALL PRIVILEGES ON compensation`
	`TO companalyst6;);`

2	`EXECUTE MACRO everything;`
3	`SELECT *` `FROM salary, bonus;`

Logging Access Attempts

If you need to maintain a security log of access attempts, see "BEGIN LOGGING" on page 1-67.

Restrictions on Granted Access Privileges

When a user explicitly grants access privileges to another user, there are rules that determine whether, how, and on what object the requested privilege can be implemented. The restrictions that apply to explicitly granted privileges are detailed in the following table.

The first column of the table lists the privilege type, the second column describes restrictions if the privilege is granted on a database or user, and the third describes restrictions if the privilege is granted on a table, view, stored procedure, or macro.

Privilege	Object (Database or User)	Table, Join Index, View, Procedure, or Macro
ALTER PROCEDURE	Privilege applies to all stored procedures in the specified space.	Privilege applies to the specified stored procedures.
CREATE DATABASE CREATE USER	CREATE granted for the specified space.	Not applicable.
CREATE MACRO CREATE TABLE CREATE VIEW CREATE PROCEDURE	CREATE granted for the object type for the specified space.	Not applicable.
CREATE PROFILE CREATE ROLE	CREATE PROFILE and CREATE ROLE cannot be granted on an object.	CREATE PROFILE and CREATE ROLE cannot be granted on an object.

Privilege	Object (Database or User)	Table, Join Index, View, Procedure, or Macro
CREATE TRIGGER	CREATE granted for the object type for the specified space.	CREATE granted for the specified table.
DROP	Not applicable.	DROP granted for specified stored procedures when PROCEDURE is specified before the procedure name.
DROP DATABASE DROP USER	DROP granted for the specified space.	Not applicable.
DROP MACRO DROP PROCEDURE DROP TABLE DROP TRIGGER DROP VIEW	DROP granted for the object type for the specified space.	DROP granted for the specified join index, hash index, macro, table, trigger, stored procedure, or view.
DROP PROFILE DROP ROLE	DROP PROFILE and DROP ROLE cannot be granted on an object.	DROP PROFILE and DROP ROLE cannot be granted on an object.
DATABASE MACRO PROCEDURE TABLE TRIGGER USER VIEW	CREATE and DROP granted for the type for the specified space.	Not applicable.
PROFILE ROLE	CREATE PROFILE, DROP PROFILE, CREATE ROLE, and DROP ROLE cannot be granted on an object.	CREATE PROFILE, DROP PROFILE, CREATE ROLE, and DROP ROLE cannot be granted on an object.
DELETE INSERT SELECT UPDATE	Privilege applies to all tables or views in the specified database. UPDATE applies to table or column of the table. For a grantee to use the granted privileges on a view, the owner of a view must have appropriate privileges on the underlying tables of the view.	Privilege applies only to the specified table or view. The owner of the view must have appropriate privileges on the underlying tables of the view. UPDATE cannot be granted on a GENERATED ALWAYS identity column.

Privilege	Object (Database or User)	Table, Join Index, View, Procedure, or Macro
EXECUTE	Privilege applies to all macros in the specified database. For the grantee to use the privilege on the macro, the owner of the macro must have appropriate privileges on the objects referenced by the macro.	Privilege applies to the specified macro if PROCEDURE is not specified. The owner of the macro must have the appropriate privileges on the objects referenced by the macro. Privilege applies to the specified stored procedures if PROCEDURE is specified.
EXECUTE PROCEDURE	Privilege applies to all stored procedures in the specified space. For the grantee to use the privilege on the procedure, the owner of the stored procedure must have the appropriate privileges on the objects referenced by the stored procedure.	Privilege applies to the specified stored procedures.
ALL	All privileges the grantor can grant on an object.	Grants EXECUTE and DROP on a macro. Grants DROP, DELETE, INDEX, INSERT, REFERENCES, SELECT, UPDATE, RESTORE, CREATE TRIGGER, DROP TRIGGER, and DUMP on a data table. Grants DROP on hash and join indexes. Grants DROP, DELETE, INSERT, SELECT, and UPDATE on a view. Grants INSERT, DUMP, RESTORE, and CHECKPOINT on a journal table.

Privilege	Object (Database or User)	Table, Join Index, View, Procedure, or Macro
ALL (continued)		Grants ALTER PROCEDURE, DROP PROCEDURE, and EXECUTE PROCEDURE privileges on a stored procedure.
DUMP RESTORE	Privilege applies to all tables in the specified database.	Privilege applies to the named data table or journal table only.
CHECKPOINT	Privilege applies to the journal table in the specified database.	Privilege applies to the named journal table.
REFERENCES	Not applicable	on a table or column of the table
INDEX	Not applicable	on a table, hash, or join index

Verifying Privileges on Views and Macros

WHEN this statement is submitted . . .	The Parser verifies that the creator has these privileges . . .
CREATE MACRO	any needed to perform the statements in the macro body.
CREATE PROCEDURE	any needed to perform the statements in the stored procedure body.
CREATE VIEW	select on the underlying base tables and views.

The Parser also verifies that the appropriate privileges exist on the target objects for any user who attempts to access a view, or perform a macro or stored procedure. This ensures that a change to a target object does not cause a violation of access privileges when the view, macro, or stored procedure referencing that object is invoked.

Granting View/Macro Privileges on Objects Owned by a Third User

To grant a privilege to a second user on a view or macro that references objects owned by a third user, the grantor must have the required privilege WITH GRANT OPTION on those objects.

Stored Procedure-Specific Privileges

The following rules apply to privileges specific to stored procedures:

- CREATE PROCEDURE is only a database- or user-level privilege.
- ALTER PROCEDURE, DROP PROCEDURE, and EXECUTE PROCEDURE apply to databases, users, or specified stored procedures.
- DROP and EXECUTE can be used as abbreviations for DROP PROCEDURE and EXECUTE PROCEDURE while granting privileges, if the object type PROCEDURE is specified before the stored procedure name.

The following happens if PROCEDURE is not specified before the object name:

IF the request is...	THEN...
GRANT EXECUTE	the object is assumed to be a macro. If no macro of that name exists, an error or failure is returned.
GRANT DROP	an error or failure is returned.

Granting Privileges on Stored Procedures

The following table describes the privileges of different types of users or grantors with respect to stored procedures-specific privileges:

This privilege...	Is granted to...	By...
ALTER PROCEDURE	user DBC implicitly	user DBC.
CREATE PROCEDURE	other users, databases explicitly	• user DBC. • owners of the user or database having this privilege WITH GRANT OPTION on themselves. • users having this privilege WITH GRANT OPTION.
DROP PROCEDURE	all users and databases	default.

This privilege...	Is granted to...	By...
EXECUTE PROCEDURE or EXEC PROCEDURE	user DBC implicitly	user DBC.
	the creator of a stored procedure	default (granted WITH GRANT OPTION). If the owner of the procedure is different from the creator, the owner does not receive this privilege by default.
	other users and databases	• user DBC. • owners of the user or database having this privilege WITH GRANT OPTION on themselves. • other users having this privilege WITH GRANT OPTION.

Granting Privileges to Roles

Roles define access privileges on database objects. A database administrator can create different roles for different job functions and responsibilities, grant specific privileges on database objects to the roles, and then grant membership to the roles to users. Users who are members of a role can access all the objects that the role has privileges to.

Roles cannot be granted the CREATE ROLE, CREATE PROFILE, CREATE USER, DROP ROLE, DROP PROFILE, or DROP USER privileges.

To grant role membership to users or other roles, use the GRANT (Role Form) statement. For more information, see "GRANT (Role Form)" on page 2-19.

GRANT and Global Temporary and Volatile Tables

GRANT always applies to the base global temporary table and never to a materialized instance. Just as with permanent tables, a user must have the appropriate privileges before submitting a GRANT statement.

Because Teradata does not check access privileges for volatile tables, you cannot GRANT privileges to them.

Example

Assume Allen creates two objects: a table named Allen.BaseTable, and a view derived from that table named Allen.ViewA. The Parser verifies that Allen has SELECT privilege on Allen.BaseTable when View A is created.

Also assume Allen grants the SELECT privilege on View A to Bobby. Bobby then creates Bobby.ViewB, which is derived from (and so references) Allen.ViewA. The Parser verifies that Bobby has SELECT privilege on Allen.ViewA when Bobby.ViewB is created.

Bobby now wants to grant the SELECT privilege on View B to Chuck. Before Bobby can do this, he must receive the SELECT privilege WITH GRANT OPTION on Allen.ViewA from Allen.

The same privilege is required when a macro references an object owned by a user other than its creator. For example, assume that the macro Bobby.MacroB deletes rows from Allen.ViewA. Bobby must have the GRANT OPTION for the EXECUTE privilege on Allen.ViewA in order to grant the EXECUTE privilege on Bobby.MacroB to another user.

Case Study: Views

Assume that the view Bobby.ViewB is derived from the view Allen.ViewA, which in turn is derived from Allen.BaseTable. When Bobby enters the following statement, the Parser checks that the privileges in the table following the statement exist.

```
GRANT INSERT, DELETE
ON Bobby.ViewB TO Chuck;
```

Stage	Process
1	Bobby has INSERT WITH GRANT OPTION on Bobby.ViewB
2	Bobby has DELETE WITH GRANT OPTION on Bobby.ViewB
3	Bobby has INSERT WITH GRANT OPTION on Allen.ViewA
4	Bobby has DELETE WITH GRANT OPTION on Allen.ViewA
5	Allen has INSERT WITH GRANT OPTION on Allen.BaseTable
6	Allen has DELETE WITH GRANT OPTION on Allen.BaseTable

Of these privileges, four are automatic and two are granted explicitly.

Ownership privileges cannot be revoked. For any other type, however, the Parser might not find one or more of the necessary privileges. In this case, an error message is returned to the user submitting the GRANT statement and the statement is not performed.

The breakdown of privilege types for this example is as follows.

FOR this category of privilege ...	THIS individual ...	HAS this privilege...	ON this object ...
Automatic	Bobby	INSERT WITH GRANT OPTION	Bobby.ViewB
		DELETE WITH GRANT OPTION	
	Allen	INSERT WITH GRANT OPTION	Allen.BaseTable
		DELETE WITH GRANT OPTION	
Explicit	Bobby	INSERT WITH GRANT OPTION	Allen.ViewA
		DELETE WITH GRANT OPTION	

Case Study: Macros

Assume that Allen creates the following macro.

```
CREATE MACRO Allen.MacroA AS
 (INSERT INTO Allen.BaseTable...
  DELETE FROM Allen.BaseTable...);
```

Bobby then creates this macro.

```
CREATE MACRO Bobby.MacroB AS
 (EXECUTE Allen.MacroA);
```

If Bobby now tries to grant the EXECUTE privilege on MacroB to Chuck, as in the following GRANT statement, Teradata checks that the privileges listed in the following table exist.

```
GRANT EXECUTE, DROP MACRO
ON Bobby.MacroB
TO Chuck;
```

THIS individual...	HAS this privilege...	ON this object...
Bobby	EXECUTE WITH GRANT OPTION	Bobby.MacroB
	DROP MACRO WITH GRANT OPTION	
	EXECUTE WITH GRANT OPTION	Allen.MacroA
Allen	INSERT WITH GRANT OPTION	Allen.BaseTable
	DELETE WITH GRANT OPTION	

Again, if Teradata does not find one or more of the required automatic or explicit privileges, the statement is not performed and an error is returned to the requestor.

Case Study: Stored Procedures

Assume that user_1 is to be granted CREATE PROCEDURE privilege on the database accounting. Assume that the grantor is the owner of the database. The following GRANT statement must be submitted so that user_1 can create procedures in the accounting database.

```
GRANT CREATE PROCEDURE ON accounting
TO user_1;
```

To grant CREATE PROCEDURE, DROP PROCEDURE, and EXECUTE PROCEDURE privileges to user1 on all stored procedures in the accounting database, submit the following GRANT statement.

```
GRANT CREATE PROCEDURE, EXECUTE PROCEDURE, DROP PROCEDURE ON
accounting
TO user_1;
```

If the keyword PROCEDURE is specified without CREATE or DROP in a GRANT statement, it confers both CREATE PROCEDURE and DROP PROCEDURE privileges (at the user or database level).

The following example grants CREATE PROCEDURE and DROP PROCEDURE privileges to user_1 on the accounting database.

```
GRANT PROCEDURE ON accounting
TO user_1;
```

Assume that user_1 needs to be granted EXECUTE PROCEDURE privilege WITH GRANT OPTION on the stored procedure daily_updates in the database accounting. Assume that all database objects exist, and the grantor is the owner of the database and also of the stored procedure object. The following GRANT statement needs to be submitted.

```
GRANT EXECUTE ON PROCEDURE accounting.daily_updates
TO user_1
WITH GRANT OPTION;
```

This statement can also be specified using the following syntax.

```
GRANT EXECUTE PROCEDURE ON accounting.daily_updates
TO user_1
WITH GRANT OPTION;
```

To grant ALTER PROCEDURE, EXECUTE PROCEDURE, and DROP PROCEDURE privileges to user_1 on the stored procedure weekly_updates, the necessary GRANT statement looks like either of these statements.

```
GRANT ALTER PROCEDURE, EXECUTE, DROP ON PROCEDURE
weekly_updates
TO user_1;

GRANT ALL ON PROCEDURE accounting.weekly_updates
TO user_1;.
```

Example 1: Granting Privileges to a Group of Users

The following statement grants privileges to a group of users. In this example, all users created under the Finance database are granted the privilege to SELECT data from the Department table, which is in the Personnel database:

```
GRANT SELECT ON Personnel.Department
TO ALL Finance;
```

Example 2: Granting Privileges to PUBLIC

The PUBLIC keyword grants privileges to all Teradata users:

```
GRANT SELECT ON Personnel.Department
TO PUBLIC;
```

Example 3: SELECT on Any Object

The following statement allows Moffit to retrieve information from any object in the Personnel database:

```
GRANT SELECT ON Personnel
TO Moffit;
```

Example 4: SELECT, INSERT, UPDATE, and DELETE on Any Object

The following statement allows Peterson to perform selects, inserts, updates, and deletes on any object in the Personnel database:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON Personnel
TO Peterson;
```

Example 5: CREATE, DROP Users

The following statement allows Phan to create and drop users in the Personnel database:

```
GRANT USER ON Personnel
TO Phan;
```

Example 6: GRANT SELECT TO ALL

The following statement allows all current and future users created under the Personnel database to select data from the Department table:

```
GRANT SELECT ON Personnel.Department
TO ALL Personnel;
```

Example 7: WITH GRANT OPTION

WITH GRANT OPTION extends the GRANT option for the indicated privileges only.

In this example, user George grants all privileges that he has on his own user space to user Marston with the following restrictions:

- George can grant only those privileges for which he has grant privilege at the user level.
- George must have the WITH GRANT OPTION privilege on the privileges he grants to Marston.

```
GRANT ALL ON Marston
TO Marston
WITH GRANT OPTION;
```

Example 8: CREATE and DROP Roles

The following statement allows user Franklin to create and drop roles:

```
GRANT ROLE
TO Franklin;
```

Example 9: Granting Privileges to a Role

The following statement grants privileges to a role. In this example, the Finance role is granted the privilege to SELECT data from the Department table, which is in the Personnel database:

```
GRANT SELECT ON Personnel.Department
TO Finance;
```

All users who are granted membership to the Finance role also have the privilege to SELECT data from the Department table in the Personnel database.

Related Statement

See "REVOKE (SQL Form)" on page 2-57 for information about revoking privileges granted by the SQL form of GRANT.

GRANT LOGON

Purpose

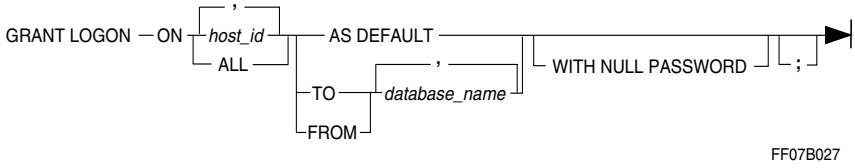
Does either of the following:

- Gives specific users permission to log on to Teradata from one or more specific client systems.
- Changes the current system logon defaults.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies...
host_id	a mainframe channel connection or a local area network connection that is currently defined to the Teradata RDBMS by the hardware configuration data. The interface need not be operational. The host ID for the Teradata RDBMS console is 0. For any other connector, *host_id* is a value ranging from 1 to 1023.
ALL	any source through which a logon is attempted, including the Teradata RDBMS console.
AS DEFAULT	that the current default for the specified host_id set is to be changed, without residual conditions, as defined in this GRANT LOGON statement. A statement with AS DEFAULT has no effect on the access granted to or revoked from particular user names. A statement that sets the default for a specific host_id takes precedence over a statement that sets the default for ALL client systems.

Syntax Element...	Specifies...
TO FROM *database_name*	to override the current default for the specified user name set on the specified host_ID set. • You cannot specify the name DBC as a user name in a GRANT LOGON statement. A statement that includes this name returns an error message. • The product of the number of host IDs times the number of user names cannot exceed 25.
WITH NULL PASSWORD	to permit a logon string that has no password to be accepted from the specified client system community. The initial Teradata RDBMS default is that all logon requests must include a password. The WITH NULL PASSWORD option, in conjunction with a TDP security exit procedure, negates that default.

ANSI Compliance

GRANT LOGON is an extension to the ANSI SQL-99 standard.

Authorization

You must be the system administrator or have the EXECUTE privilege on the DBC.LogonRule macro to perform GRANT LOGON.

System Privilege Checks

A statement that includes the AS DEFAULT option has no effect on the logon access granted to or revoked from specific user names. A user named in a GRANT LOGON statement can always access the applicable client system even if that client has a default of REVOKE, and a user named in a REVOKE LOGON statement cannot access the applicable client system even if that client has a default of GRANT.

When a GRANT LOGON statement is submitted, Teradata checks that the requesting user has EXECUTE privilege on the system macro associated with that statement. However, no checks are made on whether the user names defined in the statement apply to users owned by the requesting user. If the submitted statement cannot be verified because, for example, it specifies a non-valid user name or an invalid host ID, no action is taken on the statement.

GRANT LOGON for One or More User Names

When a GRANT LOGON statement is processed for one or more user names, a logon control record is created for each user_name/host_ID pair specified. Any existing control record for a particular user_name/host_ID pair is replaced. The logon control record created for a particular user name persists until that user is dropped (see "DROP DATABASE" on page 1-330).

If No Logon Control Record for a User Name

If there is no logon control record for a specific user name, access to the Teradata RDBMS under that name is governed by the current default for the host ID through which the logon is entered.

Logging On as User DBC

Logging onto the Teradata RDBMS as user DBC requires a password. If the submitted password is correct, the logon is accepted regardless of the current default for the applicable host ID. This prevents any opportunity to lock out all client systems from the Teradata RDBMS.

Attempts to GRANT LOGON for user name DBC return errors.

Logging On Without a Password

When the Teradata server is connected to multiple client systems, the initial default is that logon permission is granted to all users from all client systems, and that all logons must include a password.

Logging on without a password requires the following things.

- A GRANT LOGON statement that includes the WITH NULL PASSWORD option must be performed for the defined user name or must be the default for the host ID.
- A security exit installed in the client system must acknowledge that the logon string for this user is valid without a password.

FOR this client type ...	The security exit is installed in one of these client applications ...
Channel-attached	• TDP • CLI
Network-attached	• CLI • WINCLI

Logging On Using Single Sign-On

Single sign-on is available only in the Windows environment. This feature permits a user to log onto a computer one time and, for the duration of that logon, to be able to access a Teradata database without providing a user name, password, or account name. You do this by granting the user logon privileges with a null password.

See *Teradata RDBMS Utilities* for details about the single sign-on feature.

The following procedure creates a Teradata user who can log onto Teradata through a gateway that does not have the Append Domain Name option set using the Gtwcontrol utility. This user is already defined to Windows as user rhh.

Step	Action

1	Create user rhh using the following CREATE USER statement.
	`CREATE USER rhh AS`
	`PERM = 10000000,`
	`PASSWORD = rhh;`

2	Grant user rhh the following logon privileges using the GRANT LOGON statement.
	`GRANT LOGON ON ALL`
	`TO rhh`
	`WITH NULL PASSWORD;`

The following procedure creates a Teradata user who can log onto Teradata through a gateway that has Append Domain Name set. This user is already defined to Windows as user rhh and her account is in the esw2kdev domain.

Step	Action

1	Create user rhh using the following CREATE USER statement.
	`CREATE USER "rhh@esw2kdev" AS`
	`PERM = 10000000,`
	`PASSWORD = rhh;`

2	Grant user rhh the following logon privileges using the GRANT LOGON statement.
	`GRANT LOGON ON ALL`
	`TO "rhh@esw2kdev"`
	`WITH NULL PASSWORD;`

Related Statement

See "REVOKE LOGON" on page 2-70 for information about revoking logon privileges.

REVOKE

Purpose

REVOKE rescinds database privileges from one or more users. The privileges might have been conferred either automatically or by a previous GRANT statement.

ANSI Compliance

REVOKE (Role Form) is ANSI SQL-99-compliant. The other forms of REVOKE are extensions to the ANSI SQL-99 standard.

Revocation Limits

Implicit ownership privileges cannot be revoked.

Forms of REVOKE

REVOKE has four forms that differ both in function and in syntax. These forms are documented in:

- "REVOKE (SQL Form)" on page 2-57
- "REVOKE (MONITOR Form)" on page 2-49
- "REVOKE LOGON" on page 2-70
- "REVOKE (Role Form)" on page 2-54

Types of Privileges REVOKE Can Remove

The REVOKE statements operate on privileges documented in the DBC.AccessRights table. This means that you cannot revoke implicitly granted privileges because those privileges are not stored in DBC.AccessRights.

Only those privileges that have been granted automatically or explicitly can be revoked.

REVOKE (SQL Form) and REVOKE (MONITOR Form) Have Different Functions

The SQL and MONITOR forms of REVOKE perform separate functions.

To revoke all privileges, including MONITOR, for a user you must perform both of the following statements:

```
REVOKE ALL PRIVILEGES ON object
FROM user;

REVOKE MONITOR PRIVILEGES
FROM user;
```

REVOKE (MONITOR Form)

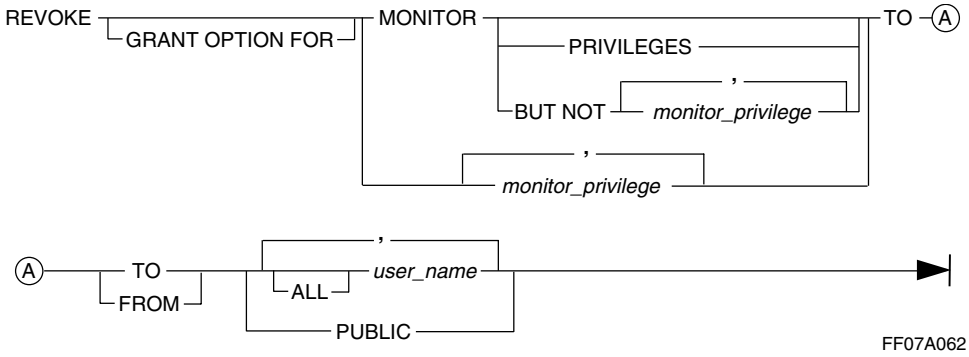
Purpose

Revokes system-wide performance monitoring privileges.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies...
GRANT OPTION FOR	that only the grant authority is removed from the specified privileges for the specified grantees. REVOKE GRANT OPTION FOR revokes the privilege of the recipient to grant, but does not revoke the stated privileges themselves.

MONITOR PRIVILEGES	to revoke from the specified user all privileges, except MONITOR, that can be granted on the specified object, and that are possessed WITH GRANT OPTION by the user executing the REVOKE.
	To revoke all s privileges for a user, including MONITOR, you must perform at least two statements as follows:
	`REVOKE ALL PRIVILEGES ON object FROM user_name;`
	`REVOKE MONITOR PRIVILEGES FROM user_name;`
	ALL PRIVILEGES means all *database* privileges.

<table>
<tr><th>Syntax Element...</th><th>Specifies...</th></tr>
<tr><td>MONITOR PRIVILEGES (continued)</td><td>If you specify the ALL PRIVILEGES option, then only the privileges possessed with grant authority are revoked from the grantee.

MONITOR PRIVILEGES indicates all monitoring privileges.

The ANSI SQL-99 standard requires ALL to be followed by PRIVILEGES.</td></tr>
<tr><td>MONITOR BUT NOT</td><td>to revoke all of the MONITOR privileges except those specified by monitor_privilege.</td></tr>
<tr><td>monitor_privilege</td><td>the monitoring privileges that are to be revoked.

You can specify as many of the following privileges as you wish, separated by commas.

<table>
<tr><th>Privilege</th><th>Description</th></tr>
<tr><td>ABORTSESSION</td><td>Aborts any outstanding request or ongoing transaction of one or more Teradata sessions and, optionally, logs off the sessions.</td></tr>
<tr><td>MONRESOURCE</td><td>Gathers information on the performance and availability of each AMP and PE.</td></tr>
<tr><td>MONSESSIONN</td><td>Gathers information about logged on sessions and overall system usage on a session-by-session basis.</td></tr>
<tr><td>SETRESRATE</td><td>Sets the frequency at which processor resource usage data is updated in Teradata.</td></tr>
<tr><td>SETSESSRATE</td><td>Sets the frequency at which session-level performance data is updated in Teradata.

Any combination of privileges can be revoked by a user who has those privileges with GRANT option.</td></tr>
</table></td></tr>
<tr><td>ALL</td><td>that the monitoring privileges are to be to revoked from the named database or user, and every database or user owned by that database or user now and in the future.

ALL user_name is a Teradata extension to the ANSI SQL-99 standard.</td></tr>
<tr><td>user_name</td><td>the database or user from whom monitoring privileges are to be revoked. You can specify up to 25 names.</td></tr>
</table>

Syntax Element...	Specifies...
PUBLIC	that the monitoring privileges are to be revoked from all currently defined and future users of Teradata. ALL DBC is equivalent to PUBLIC.

ANSI Compliance

The monitor form of REVOKE is an extension to the ANSI SQL-99 standard.

Authorization

For REVOKE (Monitor), the user submitting a REVOKE statement must have all the privileges that are to be revoked with GRANT option.

Privileges Revoked Immediately

REVOKE MONITOR takes effect immediately when the revoked user issues his next statement.

REVOKE and REVOKE MONITOR Differences

The SQL and MONITOR forms of REVOKE are separate statements. To revoke all privileges for a user, including MONITOR, the grantor must perform both of the following statements.

```
REVOKE ALL PRIVILEGES ON object
FROM user_name;

REVOKE MONITOR PRIVILEGES
FROM user_name;
```

Rules for Using REVOKE

These keywords ...	Mean that all ...
ALL PRIVILEGES	database-related privileges are affected.
MONITOR PRIVILEGES	system monitoring privileges are affected

If you violate either one of the following restrictions, a failure response occurs.

This action is not valid ...	For this type of REVOKE statement ...
Specify an ON *object* clause	REVOKE MONITOR.
Not to specify an ON *object* clause	any other type.

The result of performing a REVOKE statement takes effect as soon as the recipient issues his next statement.

Revokable Privileges

Ownership privileges, also called implicit privileges, cannot be revoked.

If a REVOKE statement removes privileges that were granted at the database- or user-level, the privileges are dropped for all objects, regardless of when they were created. A REVOKE statement at the object level cannot remove from that object a privilege that was granted at the database- or user-level.

Revoking the GRANT OPTION for a privilege from a user immediately revokes the right of that user to grant that privilege. However, it does not affect privileges that have already been granted by that user to others.

- If a user receives the same privilege from two or more grantors, any grantor with the necessary privileges can revoke that privilege from the user and from other grantees.

  The user who revokes a privilege from another need not be the grantor of that privilege.
- If a privilege that was granted to ALL user_name is revoked from user_name, the privilege is not granted automatically to future users owned by user_name.

Revoking Column-Level Privileges

Revocation of a column-level privilege is only allowed if there is a row in DBC.AccessRights for the columns on which the privilege is to be revoked. This means that if the user has UPDATE or REFERENCES privileges at the table level, revoking those privileges on individual columns is not allowed.

Examples

The following examples illustrate the use of REVOKE.

Example 1

This statement revokes from user_1 any access to any object in the Personnel database.

```
REVOKE ALL PRIVILEGES
ON Personnel
FROM user_1;
```

Example 2

This statement to leave user_1 with only the SELECT privilege on the Department table.

```
REVOKE ALL BUT SELECT
ON Personnel.Department
FROM user_1;
```

Example 3

This statement leaves user_1 with the SELECT privilege on every object in the Personnel database.

```
REVOKE ALL BUT SELECT
ON Personnel
FROM user_1;
```

Related Statement

See "GRANT (Monitor Form)" on page 2-15 for information about granting monitor privileges.

REVOKE (Role Form)

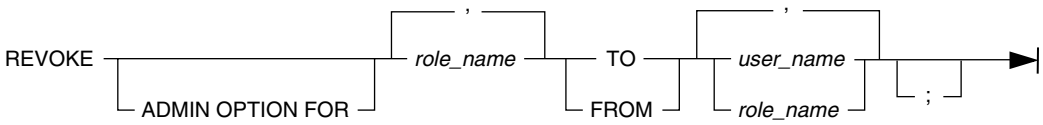
Purpose

Revokes role membership from users or other roles.

Invocation

Executable.

Syntax



KZ01a009

where:

Syntax Element …	Specifies …
ADMIN OPTION FOR	that the role members maintain membership status, but lose the privilege to use GRANT, REVOKE, and DROP ROLE statements to administer the roles to which they are members. If ADMIN OPTION FOR does not appear in the REVOKE statement, Teradata removes the specified roles or users as role members.
role_name	one or more comma-separated names of roles from which membership or administrative ability is being revoked. Teradata ignores duplicate role names.
TO FROM *user_name* *role_name*	the names of role members that are losing membership or administrative ability. Members can be users or roles. Teradata does not return errors for users or roles that are not members of a role from which membership or administrative ability is being revoked. The TO keyword is a Teradata extension to ANSI SQL-99.

ANSI Compliance

REVOKE (Role Form) is ANSI SQL-99-compliant.

Authorization

To revoke membership from a role, you must have the WITH ADMIN OPTION privilege on the role. The following users can revoke role membership:

- User DBC.
- A user who is a member of the specified role WITH ADMIN OPTION.

  The creator of a role automatically becomes a member of the role WITH ADMIN OPTION.
- A user who has an *active* role that is a member of the specified role WITH ADMIN OPTION. An active role can be the current role or a role that is a member of the current role.

Usage

Roles define access privileges on database objects. A user that is assigned a role can access all the objects on which the role and its role members have privileges. A user can only be assigned a role to which the user has been granted membership.

Users can undergo role changes within their organization. An administrator can revoke role membership when users no longer require access to the objects that the role has privileges to. An administrator can also revoke the WITH ADMIN OPTION privilege on a role when users no longer require the privilege to grant membership to the role.

Authorized users can revoke the WITH ADMIN OPTION privilege on a role from the creator of the role.

Effects of Revoking Role Membership

The effect of revoking role membership is immediate. Users who are logged on with the revoked role as the current role or a nested role of the current role lose the access privileges that the role defines.

Users who have a default role set to the revoked role do not receive errors or warnings the next time they log on. However, Teradata does not use the obsolete default role for access validation. If users again become members of the role, the default role again becomes the current role the next time users log on.

Example

If user marks get a promotion from sales to management, the DBA can use the following statements to revoke the sales role and grant the management role:

```
REVOKE sales
FROM marks;

GRANT management
TO marks;
```

For More Information

FOR more information on ...	SEE ...
adding members to roles	"GRANT (Role Form)" on page 2-19.
assigning default roles to users	• "CREATE USER" on page 1-296 • "MODIFY USER" on page 1-364
changing the current role for a session	"SET ROLE" on page 1-382

Related Statement

See "GRANT (Role Form)" on page 2-19 for information about granting role privileges.

REVOKE (SQL Form)

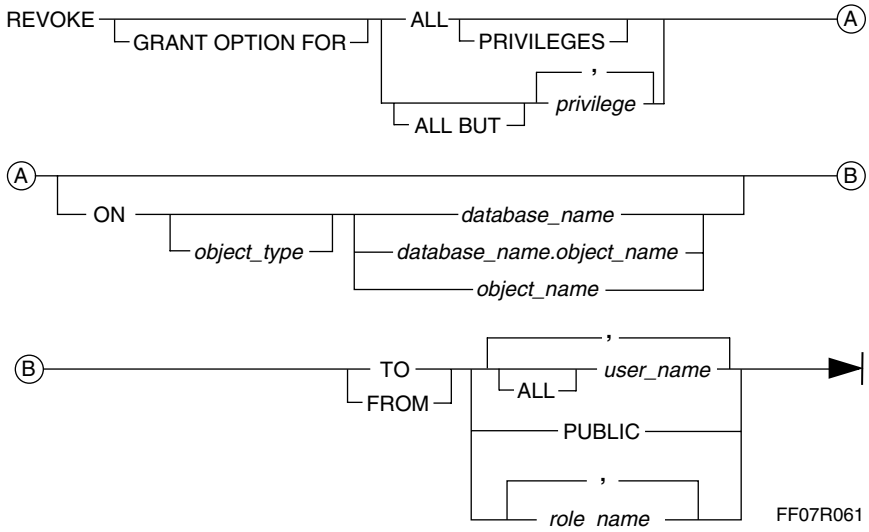
Purpose

Revokes one or more explicitly granted privileges on a database, user, table, view, stored procedure, or macro from a role, group of roles, user, or group of users.

Invocation

Executable.

Syntax



where:

Syntax Element ...	Specifies ...
GRANT OPTION FOR	that only the grant authority is removed from the specified privileges for the specified grantees. REVOKE GRANT OPTION FOR revokes the explicitly granted privilege to grant the specified privileges to others, but does not revoke the privileges themselves from the specified users or roles. This option does not apply to grantees that are roles.

Syntax Element …	Specifies …

ALL PRIVILEGES	to revoke from the specified user or role all explicitly granted privileges, except MONITOR, that can be granted on the specified object, and that are possessed WITH GRANT OPTION by the user executing the REVOKE.
	To revoke all privileges for a user, including MONITOR, the revoker must perform at least two statements, shown as follows:
	`REVOKE ALL PRIVILEGES ON` *object*
	`FROM` *user_name*`;`
	
	`REVOKE MONITOR PRIVILEGES`
	`FROM` *user_name*`;`
	ALL PRIVILEGES means only all database privileges.
	If the ALL PRIVILEGES is used, then only the explicitly granted privileges possessed with grant authority are revoked from the grantee.
	MONITOR PRIVILEGES indicates all monitoring privileges.
	ANSI SQL requires ALL to be followed by PRIVILEGES.

ALL BUT	to revoke all explicitly granted privileges from the specified user or role, except those listed, that can be granted on the specified object, with the exception of GRANT.

Syntax Element …	Specifies …
privilege	one of the following: • ALTER PROCEDURE • CHECKPOINT • CREATE DATABASE • CREATE MACRO • CREATE PROCEDURE • CREATE PROFILE • CREATE ROLE • CREATE TABLE • CREATE TRIGGER • CREATE USER • CREATE VIEW • DATABASE • DELETE • DROP • DROP DATABASE • DROP MACRO • DROP PROCEDURE • DROP PROFILE • DROP ROLE • DROP TABLE • DROP TRIGGER • DROP USER • DROP VIEW • DUMP • EXECUTE • EXECUTE PROCEDURE • INDEX • INSERT • MACRO • PROCEDURE • REFERENCES [ALL BUT] *column_name_list* • RESTORE • SELECT • TABLE • UPDATE [ALL BUT] *column_name_list* • USER • VIEW The ANSI SQL-99 standard supports only SELECT, INSERT, UPDATE, DELETE and REFERENCES as privileges. The other privileges listed are Teradata extensions to the ANSI SQL-99 standard. UPDATE and REFERENCES have table *and* column level options. See "GRANT (SQL Form)" on page 2-22 and "Revoking Column-Level Privileges" on page 2-52. The ANSI SQL-99 standard does not support REVOKE on a Database or User, or ALL BUT. These are Teradata extensions to the ANSI SQL-99 standard. You can specify any combination of privileges; however, the user submitting the statement must have all of the specified privileges WITH GRANT OPTION. If DATABASE, MACRO, PROCEDURE, PROFILE, ROLE, TABLE, USER, or VIEW is specified without CREATE or DROP, both CREATE and DROP are revoked.

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td>privileges
(continued)</td><td>If CHECKPOINT is specified, the privilege is revoked both for the SQL statement and for the HUT command. DUMP and RESTORE revoke the privilege to perform the corresponding HUT command.

If RESTORE is revoked, then the privilege to perform the HUT commands ROLLBACK, ROLLFORWARD, and DELETE JOURNAL is also revoked.

You cannot revoke access privileges on a trigger, only on the table to which the trigger applies.</td></tr>
<tr><td>object_type</td><td>the object type of the object name that follows.

This option is limited to stored procedures. PROCEDURE must be specified when the object is a stored procedure. See “Revoking Privileges on Stored Procedures” on page 2-65.</td></tr>
<tr><td>database_name</td><td>the name of a database or user on which the explicitly granted privileges are to be revoked. All objects in this database or user space are affected.</td></tr>
<tr><td>database_name.
object_name</td><td>the name of the owning database and the name of the object (table, view, stored procedure, join index, or macro) on which explicitly granted privileges are being revoked.

Only this object is affected.</td></tr>
<tr><td>object_name</td><td>the name of the table, view, stored procedures, join index, or macro from which explicitly granted privileges are being revoked.

You should always qualify object names when revoking privileges because Teradata checks database names before object names.

<table>
<tr><th>IF ...</th><th>THEN Teradata ...</th></tr>
<tr><td>the object name is not qualified</td><td>assumes it is a database name.</td></tr>
<tr><td>the object name is not qualified and no database having that name is found</td><td>assumes it is an object within the current database.</td></tr>
<tr><td>neither a database nor an object is found</td><td>returns an error to the requestor.</td></tr>
</table>
</td></tr>
<tr><td>TO
FROM</td><td>the clause that specifies the recipient of the REVOKE results.

For compatibility, ANSI SQL requires you to specify FROM.</td></tr>
</table>

Syntax Element …	Specifies …
ALL *user_name*	the database or user from whom explicitly granted privileges are revoked. You can specify up to 25 names. ALL *user_name* specifies that the privileges are to be granted to or revoked from the named database or user, and every database or user owned by that database or user now and in the future. If you do not specify ALL username, then the revocation does not cascade through the hierarchy. ALL *user_name* is a Teradata extension to ANSI SQL.
PUBLIC	that the explicitly granted privileges are to be revoked from all currently defined and future users of Teradata. ALL DBC is equivalent to PUBLIC.
role_name	the name of a role from which privileges are revoked.

ANSI Compliance

REVOKE is ANSI SQL-99-compliant with extensions.

Authorization

To revoke a privilege, you must have the privilege to grant the privilege. You must either own the database object, or someone must first grant the privilege to you using WITH GRANT OPTION.

If the object is a view or macro, the submitting user also must have the applicable privileges, WITH GRANT OPTION, on the objects referenced by the view or macro.

Rules for *privileges* Keywords

The following rules apply to using the privilege statements listed for the operations variable.

- You can specify any combination of operations. However, the user submitting the statement must have privileges on all of the specified operations.
- The CHECKPOINT operation refers to the performance of both the SQL statement and the Host Utilities (HUT) Archive/Recovery commands.

  The DUMP and RESTORE operations refer to the corresponding HUT command performed on the specified object. RESTORE also refers to execution of the HUT commands ROLLBACK, ROLLFORWARD, and DELETE JOURNAL.

  If CHECKPOINT is specified, then the privilege is revoked both for the SQL statement and for the Archive/Recovery utility commands. The DUMP and RESTORE privileges refer to the corresponding utilities commands performed on the specified object.

- CREATE DATABASE, MACRO, PROCEDURE, TABLE, VIEW or USER, and DROP DATABASE or USER are allowed only on databases or users.
- DROP TABLE includes ALTER TABLE.
- DROP MACRO, PROCEDURE, or VIEW includes REPLACE MACRO, PROCEDURE, or VIEW.
- Only DROP and EXECUTE are allowed on specified stored procedures or macros.
- Only ALTER PROCEDURE, DROP PROCEDURE, and EXECUTE PROCEDURE are allowed on stored procedures.
- DUMP, RESTORE, and CHECKPOINT are not allowed on views.
- If RESTORE is revoked, then the privilege to perform the ROLLBACK, ROLLFORWARD, and DELETE JOURNAL utility commands is also revoked.
- DATABASE, JOIN INDEX, MACRO, PROCEDURE, PROFILE, ROLE, TABLE, VIEW, and USER confer both CREATE and DROP operations. If the DATABASE, JOIN INDEX, MACRO, PROCEDURE, PROFILE, ROLE, TABLE, USER, or VIEW keyword is specified without CREATE or DROP, both CREATE and DROP are revoked.
- ANSI SQL supports only SELECT, INSERT, UPDATE, DELETE, and REFERENCES as operations.

  The other operations are Teradata extensions to ANSI SQL.
- In most cases, the privilege keyword agrees with the keyword of a Teradata SQL statement. However, the DATABASE, MACRO, PROCEDURE, TABLE, USER, VIEW, DUMP, and RESTORE operations do not correspond to an SQL statement.
- UPDATE and REFERENCES have table and column level options.
- You can revoke CREATE ROLE, DROP ROLE, CREATE PROFILE, and DROP PROFILE privileges from users only, not from roles or databases.
- CREATE ROLE, DROP ROLE, CREATE PROFILE, and DROP PROFILE are system privileges; you can revoke the privileges from a user, but not on a specific object.

Privileges Revoked Immediately

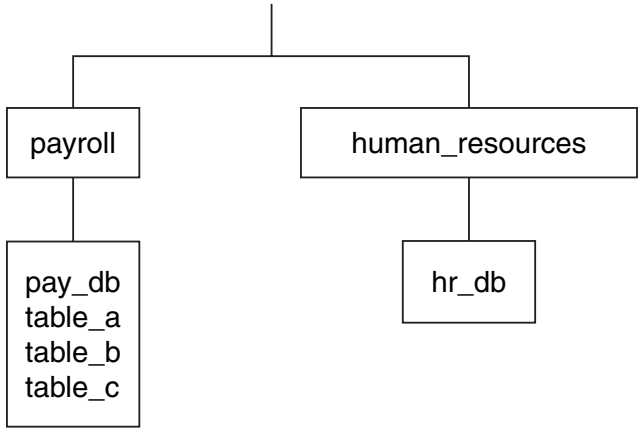
REVOKE takes effect immediately when the revoked user issues his next statement.

Privileges Must Be Revoked at the Level They Were Granted

The rows in DBC.AccessRights correlate privileges to the level in the system space hierarchy at which they were created. As a result, it is possible to submit a valid REVOKE statement that does not revoke the privileges you specify because the revocation was not requested at the correct level.

For example, if privileges on an object were granted at the database or user level, and you later submit a REVOKE statement to rescind those privileges on a particular table in that database or user space, the statement completes, but no rows are deleted from DBC.AccessRights because there are no rows correlating those privileges with that table for the specified database or user. Instead, the rows in DBC.AccessRights correlate the privileges with the containing user or database.

Consider the following space hierarchy:



1101A092

Suppose user payroll logs onto Teradata and grants the SELECT privilege on the payroll child database pay_db to user human_resources and all its descendents.

The BTEQ LOGON command and GRANT statement look like this:

```
.LOGON tdpid/payroll,password
GRANT SELECT
ON pay_db
TO ALL human_resources;
Grant accepted.
```

User payroll later decides to revoke the SELECT privilege on table_c in database pay_db from human_resources and its descendents. To revoke this privilege, user payroll submits the following REVOKE statement:

```
REVOKE SELECT
ON table_c
FROM ALL human_resources;
Revoke accepted.
```

Payroll now believes that the SELECT privilege on table_c has been revoked for human_resources and its descendents, but it has not because there were never any rows in DBC.AccessRights granting that privilege to human_resources on table_c. The row granting the SELECT privilege on the entire pay_db database to human_resources remains in DBC.AccessRights, so the originally granted SELECT privilege remains in effect.

Rules for Revoking Privileges

The following rules apply to various aspects of the SQL REVOKE statement.

- Ownership privileges cannot be revoked.
- Any combination of privileges can be revoked by a user who has those privileges and WITH GRANT OPTION.
- Teradata does not automatically revoke privileges previously granted by a user after that user is dropped from the system.
- Revoked privileges do not cascade through the hierarchy unless you specify the ALL *user_name* option.

  Conversely, if a privilege that was granted to ALL users is revoked from *user_name*, the privilege is not granted automatically to future users who are owned by *user_name*.
- The user submitting a REVOKE statement must be the owner of the object, or must have WITH GRANT OPTION and all of the privileges that are to be revoked on the object.
- If the object is a view, stored procedure, or macro, the requesting user also must have WITH GRANT OPTION and all other applicable privileges on the objects referenced by the view, stored procedure, or macro.
- If a REVOKE statement removes privileges that were granted at the database- or user-level, the privileges are dropped for all objects, regardless of when they were created. A REVOKE statement at the object level cannot remove a privilege from that object that was granted at the database- or user-level (see "Privileges Must Be Revoked at the Level They Were Granted" on page 2-62).
- Revoking an explicit GRANT privilege immediately rescinds the privilege to perform the GRANT statement. However, it does not affect privileges that have already been granted by the target user to other users.
- If a user receives the same privilege from one or more grantors, any user who has the necessary privileges can revoke that privilege from the user and from other grantees. Thus, one who revokes a privilege from another need not be the grantor of that privilege.
- If a privilege was granted to PUBLIC, the privilege can only be revoked from PUBLIC, not from individual users.

Revoking Privileges on Global Temporary and Volatile Tables

REVOKE always applies to the base global temporary table and never to a materialized instance. Just as with permanent tables, a user must have the appropriate privileges before submitting a REVOKE statement.

Because no access privilege checking is done for volatile tables, you cannot REVOKE any privileges on them.

Revoking Privileges on Stored Procedures

The following rules apply to privileges specific to stored procedures:

- CREATE PROCEDURE is only a database- or user-level privilege.
- ALTER PROCEDURE, DROP PROCEDURE, and EXECUTE PROCEDURE apply to databases, users, or specified stored procedures.
- DROP and EXECUTE can be used as abbreviations for DROP PROCEDURE and EXECUTE PROCEDURE while rescinding privileges, if the *object_type* PROCEDURE is specified before the *object_name*.
- The following happens if PROCEDURE is not specified before *object_name*:

IF the request is ...	THEN ...
REVOKE EXECUTE	the object is assumed to be a macro. If no macro by that name exists, an error or failure is returned.
REVOKE DROP	an error/failure is returned.

Revoking Privileges From Roles

Revoking a privilege from a role does not necessarily revoke the privilege from all of its role members. A member who has been granted the same privilege on an individual basis or through another role retains that privilege until it is explicitly revoked.

Example 1

This statement revokes the INSERT privilege from UserA on the Employee table:

```
REVOKE INSERT ON Personnel.Employee
FROM UserA;
```

Example 2

This statement revokes from UserA any access to any object in the Personnel database:

```
REVOKE ALL PRIVILEGES ON Personnel
FROM UserA;
```

Example 3

This statement leaves UserA with at least the SELECT privilege on the Department table:

```
REVOKE ALL BUT SELECT ON Personnel.Department
FROM UserA;
```

Example 4

This statement leaves UserA with the SELECT privilege on every object in the Personnel database:

```
REVOKE ALL BUT SELECT ON Personnel
FROM UserA;
```

Example 5: Stored Procedures

The EXECUTE PROCEDURE privilege of user_2 on the stored procedure stored_procedure_name in the database database_name must be revoked. Assume that all the specified database objects exist and that the grantor owns both the database and the stored procedure. You can submit any of the following GRANT statements to revoke the specified privileges:

```
REVOKE EXECUTE ON PROCEDURE
database_name.stored_procedure_name
FROM user_2;

REVOKE EXECUTE PROCEDURE database_name.stored_procedure_name
FROM user_2;

REVOKE EXECUTE PROCEDURE ON PROCEDURE
database_name.stored_procedure_name
FROM user_2;
```

Submit the following statement to revoke CREATE PROCEDURE, DROP PROCEDURE, and EXECUTE PROCEDURE privileges simultaneously from user_2 on the database database_name:

```
REVOKE CREATE PROCEDURE, DROP PROCEDURE, EXECUTE
PROCEDURE
ON database_name
FROM user_2;
```

To revoke ALTER PROCEDURE, EXECUTE, and DROP privileges simultaneously from user_2 on the stored procedure stored_procedure_name in database database_name, you can perform either of the following statements:

```
REVOKE ALTER PROCEDURE, EXECUTE, DROP ON PROCEDURE
database_name.stored_procedure_name
FROM user2;

REVOKE ALL ON PROCEDURE database_name.stored_procedure_name
FROM user2;
```

If you specify PROCEDURE without also specifying either CREATE or DROP in a REVOKE statement, Teradata drops both the CREATE PROCEDURE and DROP PROCEDURE privileges on database database_name.

For example, the following statement drops both CREATE and DROP PROCEDURE privileges on database_name from user_2:

```
REVOKE PROCEDURE ON database_name
FROM user_2;
```

Example 6: Roles

This statement revokes the privilege from the Finance role to change the Personnel database:

```
REVOKE UPDATE, DELETE, INSERT
ON Personnel
FROM Finance;
```

GRANT/REVOKE Order and Duration of Privileges

If a GRANT ALL ON object TO PUBLIC statement is issued by any user on an object that is lower in the hierarchy than user DBC, all other users are granted privileges on that object, including users created after the GRANT statement is issued.

If user DBC then issues a REVOKE ALL ON object_name FROM DBC, users created after the REVOKE statement was issued are not granted privileges on that object. However, all previously created users retain the privileges until a REVOKE ALL ON object_name FROM PUBLIC is issued.

For example:

```
.LOGON dbc.password

CREATE USER sys_admin AS
PERM=900000 PASSWORD=sys_admin;

GRANT ALL ON sys_admin
TO sys_admin
WITH GRANT OPTION;

.LOGON sys_admin,sys_admin

CREATE USER dept AS
PERM=500000 PASSWORD=dept;

GRANT ALL ON dept
TO dept
WITH GRANT OPTION;
```

```
.LOGON DEPT,DEPT

CREATE USER user_1 AS
PERM=100000 PASSWORD=user_1;

.LOGON USER1,USER1

CREATE TABLE table_1
 (column_1 INTEGER,
  column_2 INTEGER)
PRIMARY INDEX(column_1);

INSERT table_1(1,2);

INSERT table_1(3,4);

GRANT ALL ON table_1
TO PUBLIC;

.LOGON dept,dept

CREATE USER user_2 AS
PERM=100000 PASSWORD=user_2;

.LOGON user_2,user_2

SELECT *
FROM user_1.table_1;
```

The contents of table_1 are returned as expected.

```
.LOGON dbc,password

REVOKE ALL ON user_1.table_1
FROM dbc;

.LOGON dept,dept

CREATE USER user_3 AS
PERM=100000 PASSWORD=user_3;
```

```
.LOGON user_3,user_3

SELECT *
FROM user_1.table_1;
```

An error is returned because user_3 was created after the REVOKE was issued.

```
.LOGON user_2,user_2
SELECT *
FROM user_1.table_1;
```

The contents of table_1 are returned as expected.

Related Statement

See "GRANT (SQL Form)" on page 2-22 for information about granting SQL privileges.

REVOKE LOGON

Purpose

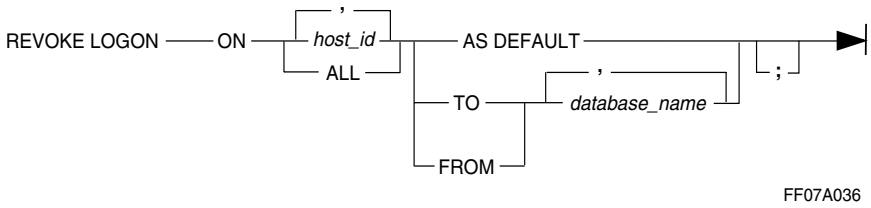
Does either of the following.

- Revokes permission to log onto Teradata from one or more specific client systems.
- Changes the current system logon defaults.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies...
host_id	an integer that identifies a mainframe channel connection or a LAN connection that is currently defined to Teradata by the hardware configuration data. The interface need not be operational. The value for the Teradata console is 0. For any other connector, the value for *host_id* ranges from 1 to 1023.
ALL	any source through which a logon is attempted, including the Teradata console.
AS DEFAULT	that the current default for the specified host ID set is to be changed, without residual conditions, as defined in this REVOKE LOGON statement. A statement with AS DEFAULT has no effect on the access revoked from or granted to particular user names. A statement that sets the default for a specific host ID takes precedence over a statement that sets the default for ALL client systems.
TO FROM	keywords introduced to override the current default for the specified database_name_list or user_name set on the specified host_ID set.

Syntax Element...	Specifies...
database_name_list	a list of databases from which logons are to be revoked. The name DBC cannot be specified either as a database name or as a user name in a REVOKE LOGON statement. Any statement that includes the name DBC returns an error message. The product of the number of host IDs and the number of database and user names cannot exceed 25.

ANSI Compliance

REVOKE LOGON is an extension to the ANSI SQL-99 standard.

Authorization

You must be the system administrator or have the EXECUTE privilege on the DBC.LogonRule macro to perform REVOKE LOGON.

No checks are made on whether the database names or user names defined in the statement apply to users owned by the requesting user.

If the submitted statement cannot be verified because it specifies an non-valid database or user name or a non-valid host ID, then it is not performed.

Default Logon Permissions

When Teradata is connected to multiple client systems, the initial default is that logon permission is granted to all users from all host IDs, and that all logons must include a password.

- The GRANT LOGON and REVOKE LOGON statements control which users have access from which client system connections.
- A REVOKE LOGON statement inhibits only future logon attempts; it does not affect users who are currently logged on.

Each REVOKE LOGON Statement Creates A New Control Record

When you enter a REVOKE LOGON statement for one or more database or user names, a logon control record is created for each database or user name-host ID pair specified.

Any existing control record for a particular pair is replaced. The logon control record set created for a particular database or user name exists until that user is dropped (see "DROP DATABASE" on page 1-330).

REVOKE LOGON on User DBC Is Not Valid

Any attempt to REVOKE LOGON for user name DBC returns an error. If an attempt is made to log onto Teradata with user name DBC and the correct password is submitted, then the logon is accepted regardless of the current default for the applicable host ID. This prevents any opportunity to lock out all hosts from user DBC.

AS DEFAULT Option

A statement that includes the AS DEFAULT option has no effect on the logon access granted to or revoked from specific database or user names.

The following rules apply to the AS DEFAULT option:

A user named in this type of statement ...	Assumes these access privileges ...
REVOKE LOGON	cannot access the applicable client even if that client has an assigned default of GRANT.
GRANT LOGON	can always access the applicable client even if that client has an assigned default of REVOKE.

Related Statement

See "GRANT LOGON" on page 2-43 for information about granting logon privileges.

Chapter 3:

Query and Workload Analysis Statements

Introduction

This chapter documents Teradata SQL statements used to collect or analyze data demographics and statistics. The collected data is then used in several possible ways, for example:

- To provide the Optimizer with data it needs to generate the best query plans possible.
- To populate user-defined QCD tables with data used by various Teradata utilities to analyze query workloads as part of the ongoing process to reengineer the database design process.

  For example, the Teradata Index Wizard determines optimal secondary index sets to support the query workloads you ask it to analyze.

The chapter also documents diagnostic statements used to support the Teradata Index Wizard and the cost- and sample-based components of the target level emulation facility used to emulate a Teradata production environment on a test system. Once the test environment has been configured and enabled with the appropriate production system statistical and demographic data, you can perform various workload analyses to determine optimal sets of secondary indexes to support those workloads in the production environment.

Further Information

The query capture database is described in the chapter “Query Capture Facility” in *Teradata RDBMS SQL Reference, Volume 2*.

The Teradata Index Wizard is described in the following documents:

- The chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2*.
- *Teradata Index Wizard User Guide*

The Teradata Statistics Wizard is described in the following document:

- *Teradata Statistics Wizard User Guide*

Target Level Emulation is described in the chapter “Target Level Emulation” in *Teradata RDBMS SQL Reference, Volume 2*.

Teradata data definition language (DDL) statements are described in Chapter 1: “SQL Data Definition Language Statement Syntax.”

Teradata data control language (DCL) statements are described in Chapter 2: “SQL Data Control Language Statement Syntax.”

Teradata data manipulation language (DML) statements are described in *Teradata RDBMS SQL Reference, Volume 6*.

Database query logging is described in "Tracking Processing Behavior With the Database Query Logs (DBQL)" in *Teradata RDBMS Database Administration*.

Experienced SQL users can also consult the simplified statement descriptions in *Teradata RDBMS SQL/Data Dictionary Quick Reference*.

Statements Described In This Chapter

This chapter describes the following SQL statements:

- "BEGIN QUERY LOGGING" on page 3-4
- "COLLECT DEMOGRAPHICS" on page 3-11
- "COLLECT STATISTICS (Optimizer Form)" on page 3-14
- "COLLECT STATISTICS (QCD Form)" on page 3-27
- "DROP STATISTICS (Optimizer Form)" on page 3-31
- "DROP STATISTICS (QCD Form)" on page 3-37
- "DUMP EXPLAIN" on page 3-40
- "END QUERY LOGGING" on page 3-44
- "INITIATE INDEX ANALYSIS" on page 3-46
- "INSERT EXPLAIN" on page 3-55
- "RESTART INDEX ANALYSIS" on page 3-62

The chapter also describes the following SQL diagnostic statements that support target level emulation and the Teradata Index Wizard:

- "DIAGNOSTIC "Validate Index"" on page 3-83
- "DIAGNOSTIC DUMP COSTS" on page 3-66
- "DIAGNOSTIC HELP COSTS" on page 3-68
- "DIAGNOSTIC SET COSTS" on page 3-71
- "DIAGNOSTIC DUMP SAMPLES" on page 3-76
- "DIAGNOSTIC HELP SAMPLES" on page 3-78
- "DIAGNOSTIC SET SAMPLES" on page 3-80

Query and Workload Analysis Statements

This section documents three related categories of query and workload analysis statements:

- Those used to collect and drop statistical and demographic data used by the Optimizer or by the workload analysis utilities
- Those used to collect query-related data and insert it into a query capture or query logging database
- Those used to perform an index analysis

The following statements are generally performed indirectly using a Teradata client database query analysis utility, but can also be performed as standalone SQL statements:

- "COLLECT DEMOGRAPHICS" on page 3-11
- "COLLECT STATISTICS (QCD Form)" on page 3-27
- "DROP STATISTICS (QCD Form)" on page 3-37
- "DUMP EXPLAIN" on page 3-40
- "INITIATE INDEX ANALYSIS" on page 3-46
- "INSERT EXPLAIN" on page 3-55
- "RESTART INDEX ANALYSIS" on page 3-62

The following statements are generally performed directly using SQL, but can also be performed using client database query analysis utilities:

- "BEGIN QUERY LOGGING" on page 3-4
- "COLLECT STATISTICS (Optimizer Form)" on page 3-14
- "DROP STATISTICS (Optimizer Form)" on page 3-31
- "DUMP EXPLAIN" on page 3-40
- "END QUERY LOGGING" on page 3-44
- "INSERT EXPLAIN" on page 3-55

BEGIN QUERY LOGGING

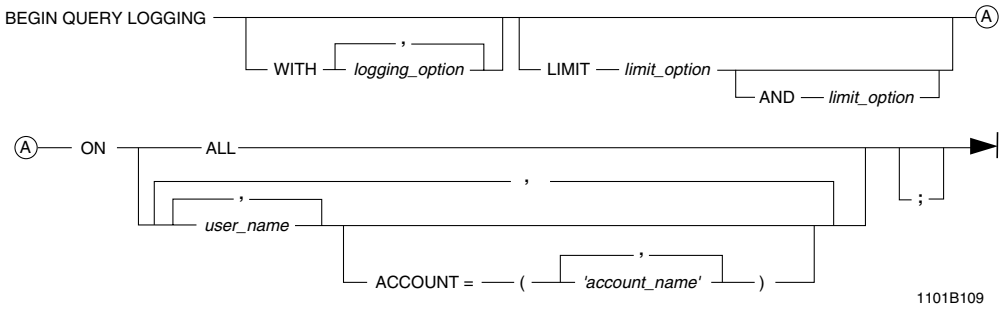
Purpose

Starts the logging of database query information.

Invocation

Executable.

Syntax



where:

Syntax element ...	Specifies ...
logging_option	options for logging the objects used by a query, the AMP steps produced to perform the query, or the text of the query itself. If you do not specify a logging option, then only the default set of query information is logged in a defined default row (see *Teradata RDBMS Data Dictionary* for a description of the fields defined for this row) for each specified user.

Option	Description
ALL	Log query object information, step information, and all SQL statements for the specified user set. This is equivalent to specifying OBJECTS, STEPINFO, and SQL individually. This option does *not* invoke either the SUMMARY or THRESHOLD limit options. If you specify ALL, then you should also specify a value of 0 for SQLTEXT to ensure that the text for the SQL statement is not logged in both the DBQLogTbl and DBQLSqlTbl tables.
OBJECTS	Log information about databases, tables, columns, and indexes accessed by SQL queries for the specified user set.

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>logging_option
(continued)</td><td>
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>SQL</td><td>Log the full text of all SQL statements performed by the specified user set in system table DBQLSqlTbl.</td></tr>
<tr><td>STEPINFO</td><td>Log AMP step-level information for all SQL queries performed by the specified user set in system table DBQLStepTbl.</td></tr>
</table>
</td></tr>
<tr><td>limit_option</td><td>one of the following options for determining various boundary conditions for query logging.
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>SQLTEXT[=n]</td><td>Sets the maximum number of SQL text characters to log in the default row. The default value is 200.

The value specified cannot be negative. If you specify 0, then no characters are logged.

If you specify SQLTEXT without specifying a numeric value to limit the number of characters logged, then the entire SQL statement is logged up to a maximum of 10 000 characters. If the statement exceeds 10 000 characters, then the excessive characters are not logged.

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 DBQLogTbl and DBQLSqlTbl tables.</td></tr>
<tr><td>SUMMARY=n1,n2,n3</td><td>Designed for use with short, OLTP-like, queries. Counts the number of queries for the session that fall into each of four time intervals. Interval values must be specified in seconds.

If you specify SUMMARY, then you cannot also specify THRESHOLD.

You must specify the first three intervals explicitly. The fourth interval is created by default.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>limit_option
(continued)</td><td>
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>SUMMARY=n1,n2,n3
(continued)</td><td>
<table>
<tr><th>The range of this interval …</th><th>Is from …</th></tr>
<tr><td>first</td><td>0 to n_1 seconds.</td></tr>
<tr><td>second</td><td>n_1 to n_2 seconds.</td></tr>
<tr><td>third</td><td>n_2 to n_3 seconds.</td></tr>
<tr><td>fourth</td><td>> n_3 seconds.</td></tr>
</table>
Counts for each interval are stored in the QueryCount field of four separate rows in DBQLSummaryTbl.

SUMMARY is the only option whose cache is flushed at regular, system-defined, intervals.</td></tr>
<tr><td>THRESHOLD[=n]</td><td>Sets a threshold time in seconds that determines whether a query is to be logged fully or just counted.

If a query completes earlier than or equal to the threshold value, then it is only logged as a count in DBQLogTbl. The Threshold row in DBQLSummaryTbl is identified by a HighHist field value of 0.

If a query completes later than the threshold value, then a full entry is logged for it in DBQLogTbl with values for all fields of the row.

If you specify THRESHOLD without also specifying a value for n, then the value 5 seconds is assigned by default.

If you specify THRESHOLD, then you cannot also specify SUMMARY.

If the threshold limit is not used, then all queries are logged fully.

The maximum value for n is 32767 seconds (roughly 9 hours).</td></tr>
</table>
</td></tr>
<tr><td>user_name</td><td>the name of a user for whom SQL query information is to be logged.

The more users you specify, the greater the impact on system performance.
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>ALL</td><td>Log query information for all users.</td></tr>
<tr><td>user name</td><td>Log query information for the specified user set.</td></tr>
</table>
</td></tr>
</table>

Syntax element …	Specifies …
account_name	the name of a user account for which SQL query information is to be logged. Account names must be enclosed by LEFT and RIGHT PARENTHESIS characters. When you specify a list of accounts, each account name must be delimited by APOSTROPHE characters and separated by COMMA characters.

ANSI Compliance

BEGIN QUERY LOGGING is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have EXECUTE privileges on the macro DBQLAccessMacro to perform BEGIN QUERY LOGGING.

Uses for Database Query Logging

The intent of the database query log is to provide the DBA 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 a large amount of valuable system resources.

Teradata provides several tools to support these analyses, including Meta Data Services utilities, various Teradata Manager utilities, and Database Query Analysis tools. In particular, the Teradata Statistics Wizard can use database query log information to analyze SQL workloads for determining recommendations for collecting statistics on particular tables, columns, and indexes.

These analytic instruments can use the logged information together with other system tables to probe and report system information which can then be used to support the enhancement of system throughput.

Limits of Dynamically Enabled Query Logging

Query logging can be begun or ended dynamically only for the first 100 users specified in the user name list of a BEGIN QUERY LOGGING statement. For these users, query logging begins as soon as they perform their next SQL request.

Users 101 and above in the user name list are also enabled for query logging, but must log off and then log on again before their queries can be logged or before ongoing logging can be terminated. Once this has been done, the next SQL request the logging-enabled user issues causes the appropriate logging rule and user account string to be retrieved and begins to capture query logging data.

For the situation in which more than 100 users must be enabled for query logging, the best practice is to perform additional BEGIN QUERY LOGGING statements for each additional block of 100 users.

Minimum Level of Query Logging

When query logging is enabled, 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 View

The tables and view that log SQL query data are in the data dictionary. The following table summarizes them.

Dictionary Object Name	Object Type	Description
DBC.DBQLRuleTbl	Table	Stores the rules for the Database Query Log.
DBC.DBQLRuleCountTbl		Stores the cardinality of DBQLRuleTbl.
DBC.DBQLogTbl		Stores default rows for the Database Query Log. Query-level CPU and I/O counts are not captured in the 5.0 release.
DBC.DBQLObjTbl		Stores query object information.
DBC.DBQLSummaryTbl		Stores summary and threshold information for the logged query.
DBC.DBQLStepTbl		Stores query step information.
DBC.DBQLSqlTbl		Stores SQL query text.
DBC.DBQLRules	View	A view onto DBC.DBCQLRuleTbl.

Table DBC.DBQLExpTbl exists, but is not populated:

For details about the definitions of the query log tables, see *Teradata RDBMS Data Dictionary*.

Query Logging User Rules

Because you can log different information for different users, the system table DBQLRuleTbl is used to maintain the different logging rules for each user individually. These rules are based on the logon account string for each logged user and are maintained in cache to promote their quick access.

User access to DBQLRuleTbl is limited to the DBQLRules view: you cannot access either DBQLRuleTbl or DBQLRuleCountTbl directly.

Query Logging Is Resource Intensive

You should analyze your analytic needs carefully before you begin to log queries using the Database Query Log because the process can exert a relatively high cost on system performance.

Example 1

Log default query information on all users and accounts.

```
BEGIN QUERY LOGGING ON ALL;
```

Example 2

Log query information for AMP steps, database objects, and SQL text up to 1 000 characters on user_1.

```
BEGIN QUERY LOGGING WITH STEPINFO, OBJECTS LIMIT SQLTEXT = 1000
ON user_1;
```

Example 3

Log query information for AMP steps, database objects, and SQL text up to 100 characters on user_1 and user_2.

Queries that complete in less than 4 seconds are logged in DBQLSummaryTbl, but only recorded as counts in the QueryCount field.

All fields in DBQLSummaryTbl are logged for queries that take longer than 4 seconds to complete.

```
BEGIN QUERY LOGGING WITH STEPINFO, OBJECTS LIMIT THRESHOLD = 4
AND SQLTEXT = 100 ON user_1, user_2;
```

Example 4

Log query information for AMP steps, database objects, and SQL text for all users logged on under account_1 and account_2.

```
BEGIN QUERY LOGGING WITH ALL ON ALL ACCOUNT = ('account_1',
'account_2');
```

Example 5

Log counts of the number of queries performed by user_1 into the following interval set in DBQLSummaryTbl:

- 0 - 1 second
- 1 - 5 seconds
- 5 - 10 seconds
- > 10 seconds

```
BEGIN QUERY LOGGING WITH LIMIT SUMMARY=1,5,10 ON user_1;
```

For More Information

See the following references for more information about query logging:

- "END QUERY LOGGING" on page 3-44
- "Tracking Processing Behavior With the Database Query Logs (DBQL)" in *Teradata RDBMS Database Administration*
- *Teradata RDBMS Data Dictionary*
- *Teradata Statistics Wizard User Guide*

COLLECT DEMOGRAPHICS

Purpose

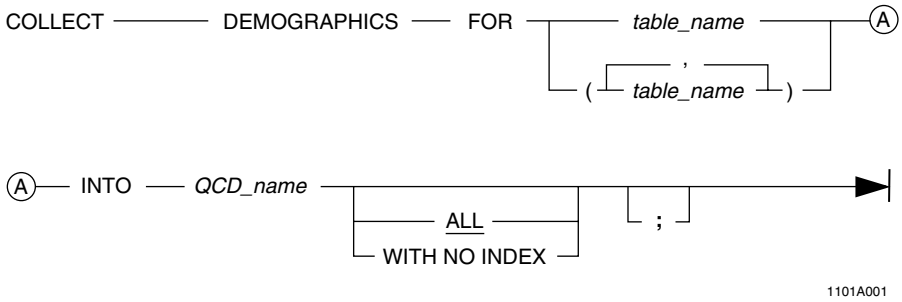
Collects various table demographics and writes the data to the DataDemographics table of a user-defined QCD database for subsequent analysis by the Teradata Index Wizard.

Invocation

Executable.

Normally invoked via the Teradata Index Wizard utility or the Visual EXPLAIN tool.

Syntax



where:

Syntax element ...	Specifies ...
table_name	the names of tables for which data demographics are to be collected.
QCD_name	the name of the QCD database into which the collected demographic data is to be written.
ALL	to collect both primary and index subtable demographics. This is the default.
WITH NO INDEX	to exclude index subtable demographics from the collection and collect only primary data.

ANSI Compliance

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

Authorization

You must have both of the following privileges to perform COLLECT DEMOGRAPHICS:

- INSERT on the DataDemographics table in *QCD_name* or INSERT on the *QCD_name* database.
- SELECT on the specified tables or containing databases.

Demographics Collected

COLLECT DEMOGRAPHICS determines and writes the following information for each specified table on an AMP-wise basis[1] into the DataDemographics table of the specified query capture database:

- Subtable type
- Subtable ID
- Cardinality
- Average row length
- Various system-related information

COLLECT DEMOGRAPHICS does not capture information for the QCD table TableStatistics. TableStatistics is used only during the collection of statistics invoked by performing an INSERT EXPLAIN … WITH STATISTICS or COLLECT STATISTICS (QCD Form) statement.

See "DataDemographics" in the chapter "Query Capture Facility" in *Teradata RDBMS SQL Reference, Volume 2* for more detailed information.

Relationship to INSERT EXPLAIN WITH STATISTICS AND DEMOGRAPHICS

The information retrieved for COLLECT DEMOGRAPHICS and INSERT EXPLAIN WITH STATISTICS AND DEMOGRAPHICS is identical. The principal differences are:

- INSERT EXPLAIN WITH STATISTICS AND DEMOGRAPHICS uses an SQL query to collect the information, while COLLECT DEMOGRAPHICS obtains the information directly.
- Demographics captured by INSERT EXPLAIN WITH STATISTICS AND DEMOGRAPHICS are automatically deleted whenever you delete the relevant query plans.

  Demographics captured by COLLECT DEMOGRAPHICS are not deleted when you perform associated DROP actions on the subject table and must be deleted explicitly.

---

1 One row of information is collected in the appropriate DataDemographics subtable of the specified QCD for each AMP that is online at the time the statement is performed.

Example 1

Collect demographics on table_1 in database db_1 and write the results into the DataDemographics table of user-defined QCD MyQCD.

```
COLLECT DEMOGRAPHICS FOR db_1.table_1 INTO MyQCD;
```

Example 2

An entry already exists in the DataDemographics table for the specified table, then it is updated with the currently collected data. The TimeStamp is updated to reflect the current time.

```
COLLECT DEMOGRAPHICS FOR db_1.table_1 INTO MyQCD; /* Inserts
data into DataDemographics table of MyQCD */

COLLECT DEMOGRAPHICS FOR db_1.table_1 INTO MyQCD; /* Updates
data in DataDemographics table of MyQCD */
```

For More Information

See the following topics for more information about index analysis:

- “COLLECT STATISTICS (QCD Form)” on page 3-27
- “INITIATE INDEX ANALYSIS” on page 3-46
- “INSERT EXPLAIN” on page 3-55
- “RESTART INDEX ANALYSIS” on page 3-62
- The chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2*.
- The chapter "Query Capture Facility" in *Teradata RDBMS SQL Reference, Volume 2*.
- *Teradata Index Wizard User Guide.*

COLLECT STATISTICS (Optimizer Form)

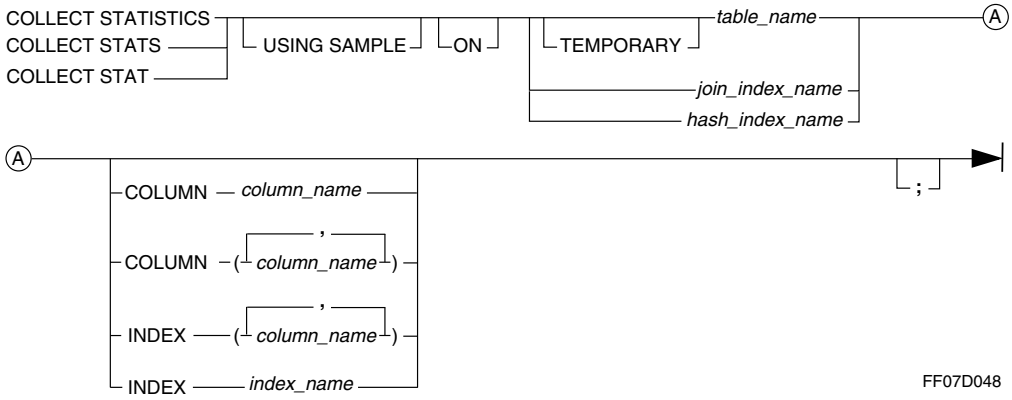
Purpose

Collects demographic data for one or more columns of a table, hash index, or join index, computes a statistical profile of the collected data, and stores the synopsis in the data dictionary. The Optimizer uses the synopsis data when it generates its table access and join plans.

Invocation

Executable.

Syntax

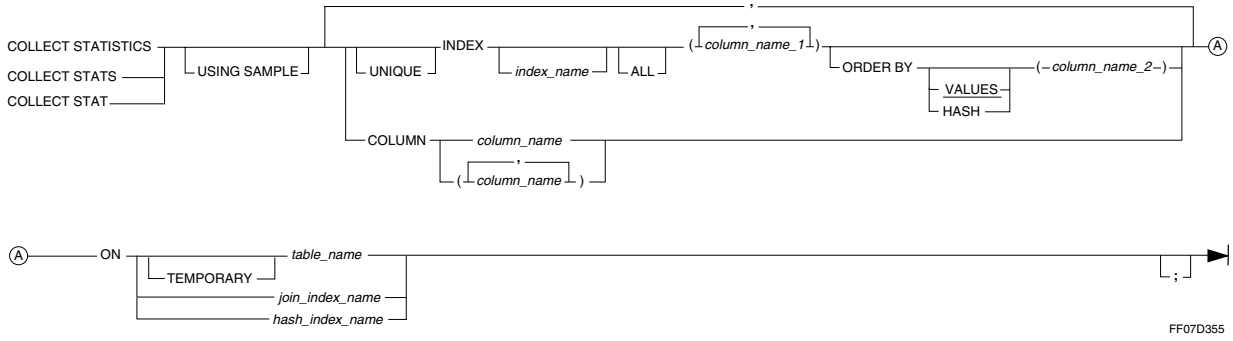


where:

Syntax Element …	Specifies …
USING SAMPLE	to scan a system-selected percentage of table rows rather than scanning all rows for a table. You can only specify this clause if you also specify an explicit column or index. In other words, you cannot specify a USING SAMPLE clause for a standard recollection of statistics on implicitly specified column and index sets (see "Collecting Statistics When No COLUMN or INDEX Clause Is Specified" on page 3-22 for details about recollecting statistics). You cannot specify a USING SAMPLE clause with global temporary tables, hash indexes, or join indexes. See "Reducing the Cost of Collecting Statistics by Sampling" on page 3-19 for more information.

Syntax Element …	Specifies …
TEMPORARY	that statistics are to be collected for a materialized temporary table. This keyword is not valid for permanent tables and if you use it with a table defined as permanent, an error is returned. If you do not specify TEMPORARY, but the referenced table is a temporary table, the statistics are collected for an empty table, but on the base temporary table rather than its materialized instance. This is allowed so that when an instance is created, statistics can be re-collected by simply entering COLLECT STATISTICS on the temporary table without having to specify the desired columns and indexes. If you specify TEMPORARY correctly, but no materialized instance of the definition exists in the system, Teradata materializes an instance with all of the statistics defined on the base temporary table before collecting statistics for it.
table_name	the name of the table or temporary table for which statistics are to be collected.
join_index_name	the name of the join index for which statistics are to be collected.
hash_index_name	the name of the hash index for which statistics are to be collected.
COLUMN	that statistics are to be collected for the specified column set.
column_name	the names of the non-indexed columns for which statistics are to be collected. The maximum number of non-indexed columns on which joint statistics can be collected for a single COLLECT STATISTICS statement is 64.
INDEX	that statistics are to be collected for one or more index columns.
column_name	the names of the index columns for which statistics are to be collected.
index_name	the name of the index for which statistics are to be collected. When you specify a named index, you do not specify column names.

CREATE INDEX-Style Syntax



where:

Syntax Element ...	Specifies ...
UNIQUE	the index for which statistics are to be collected is unique.
index_name	the name of the named index for which statistics are to be collected. Note that you specify column names with a named index only when it is defined with an ORDER BY clause.
ALL	that ALL was used to create the index. ALL is required for NUSIs on join indexes with compressed fields, but ALL can also be used for indexes on tables.
column_name_1	the names of one or more columns on which the index exists and for which statistics are to be collected.
ORDER BY	that ORDER BY was used to create the index. ORDER BY orders the index by the contents of a single column. If you specify ORDER BY but not VALUES or HASH, then VALUES is assumed.
VALUES	that ORDER BY VALUES was used to create the index. VALUES can order a single numeric column of four bytes or less.
HASH	that ORDER BY HASH was used to create the index. HASH hash-orders a single column instead of hash-ordering all columns (the default).
column_name_2	the name of the column on which the index is ordered. If you use ORDER BY, but not *column_name_2*, then the first column in the *column_name_1* list is assumed by default.
COLUMN	that statistics are to be collected for a non-indexed column set.

Syntax Element …	Specifies …
column_name	the names of the non-indexed columns for which statistics are to be collected. The maximum number of non-indexed columns on which joint statistics can be collected for a single COLLECT STATISTICS statement is 64.
TEMPORARY	that statistics are to be collected for a materialized temporary table. This keyword is not valid for permanent tables and if you use it with a table defined as permanent, an error is returned. If you do not specify TEMPORARY, but the referenced table is a temporary table, the statistics are collected for an empty table, but on the base temporary table rather than its materialized instance. This is allowed so that when an instance is created, statistics can be re-collected by simply entering COLLECT STATISTICS on the temporary table without having to specify the desired columns and indexes. If you specify TEMPORARY correctly, but no materialized instance of the definition exists in the system, Teradata materializes an instance with all of the statistics defined on the base temporary table before collecting statistics for it.
table_name	the name of the table or temporary table for which column or index statistics are to be collected.
join_index_name	the name of the join index for which statistics are to be collected.
hash_index_name	the name of the hash index for which statistics are to be collected.

ANSI Compliance

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

Authorization

To use COLLECT STATISTICS on …	You must have INDEX or DROP TABLE privileges on …
a table	the table.
a join index	both of the following: • The join index • All of the base tables of the join index

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 statement parsing.

Accurate demographics enable the query optimizer to determine the least costly access and join plans for a query.

It is particularly important for the Optimizer to have accurate table demographic information for those situations in which data to be accessed using a column or index might be skewed.

You can also collect, drop, and view table statistics using the Statistics Collection module of Teradata Manager.

Keep Statistics Current

If a table has been extensively modified (more than 10 percent of the rows added or deleted) since its statistics were last computed, residual statistics are likely to cause the Optimizer to generate poor access and join plans. Therefore, statistics should be periodically refreshed or dropped.

You can update previously collected statistics by performing a COLLECT STATISTICS statement without specifying an explicit list of columns or indexes (see "Collecting Statistics When No COLUMN or INDEX Clause Is Specified" on page 3-22). Teradata then automatically recollects statistics for all existing statistics.

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 EXPLAINs 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 statement to determine which columns and indexes currently have statistics and to see a portion of the collected information. You can view more of the statistics for a column using the Statistics Collection module of Teradata Manager.

You can also use the client Teradata Statistics Wizard utility to analyze workloads in order to provide recommendations for collecting statistics on particular tables, columns, and indexes. This utility provides modules for creating SQL workloads and validating the performance enhancements it predicts by testing those statistics (see *Teradata Statistics Wizard User Guide* for further information about the Teradata Statistics Wizard).

Collecting Statistics Can Be Time Consuming

Because Teradata always does a full-table scan when it collects full statistics on a table, the process can take a long time to complete. The duration of the scan is dependent on the size of the table, the system configuration, and the work load 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 pertinent characteristics of the column.

Columns that are not indexed or that are unique indexes take more time to process than nonunique secondary indexes.

You should collect statistics on newly created, empty tables. This defines the columns and indexes as well as the synoptic data structure for subsequent collection.

When you perform a HELP STATISTICS statement on an empty table, the Unique Value column returns 1, which is the default value.

Reducing the Cost of Collecting Statistics by Sampling

Depending on the workload on your system, you might want to consider collecting statistics on a sampled subset of table rows instead of collecting demographics based on all the rows for a table.

The performance enhancement gained by sampling a portion, rather than all, of table data is offset by decreased confidence in the accuracy of the statistics.

The percentage of data scanned is based on cardinality and skew and ranges between 1 and 10 percent of the rows depending on the demographics of the table. When significant skew is detected, the system dynamically increases the sample size to ensure that the statistics accurately account for it.

Once initial sampled statistics have been collected, recollected statistics on the columns and indexes specified by that operation are collected in the same mode as they were originally collected. The system does *not* store both sampled and complete statistics for the same index or column set.

Advantages of the CREATE INDEX Syntax

The CREATE INDEX syntax for COLLECT STATISTICS has two advantages.

- Permits statistics to be collected for multiple indexes of a table within a single SQL statement.
- Permits you to copy a CREATE INDEX statement and change the keywords CREATE INDEX to COLLECT STATISTICS. The syntax is otherwise nearly identical to that of CREATE INDEX

If you specify the USING SAMPLE option with this syntax, Teradata collects sampled statistics on each of the indexes listed within the COLLECT STATISTICS statement.

Rules for COLLECT STATISTICS

The following lists rules for using COLLECT STATISTICS:

- COLLECT STATISTICS must be entered in one of three ways:
  - As a single-statement request.
  - As the only statement in a macro.
  - As the only or last statement in an explicit transaction bracketed by BEGIN and END TRANSACTION statements.
- If you perform a COLLECT STATISTICS statement on a table for which statistics have been dropped and neither a COLLECT STATISTICS COLUMN nor a COLLECT STATISTICS INDEX clause has been specified, then an error is reported.
- You can collect most statistics on a global temporary table; however, you cannot collect statistics on multiple non-index columns for global temporary tables.
- You *cannot* collect statistics on a volatile table.
- You cannot specify a USING SAMPLE clause for the following:
  - Standard recollection of statistics on implicitly specified column and index sets.
  - Global temporary tables.
  - Hash and join indexes.
- You can only collect statistics on multiple non-indexed columns from permanent base tables.

  You cannot collect these statistics for global temporary tables or hash and join indexes.
- The maximum number of column and index sets you can recollect on a table without explicitly specifying those column and index sets is 512.

- The following rules apply to collecting statistics on PPI tables:
  - You can collect statistics on both the primary index and 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.
  - When you are using range partitioning, you should collect statistics on the partitioning column set.

Locks and Concurrency

When you perform a COLLECT STATISTICS statement, an ACCESS lock is placed on the table from which the demographic data is being collected.

In general, COLLECT STATISTICS statements can run concurrently with DML statements, other COLLECT STATISTICS statements, DROP STATISTICS statements, and HELP STATISTICS statements against the same table.

The following bullets list the exceptions to this rule:

- If a transaction containing a DML statement requests an EXCLUSIVE lock on a table, the DML statement blocks COLLECT STATISTICS statements, or a COLLECT STATISTICS statement blocks the DML statement, depending on which one requests its lock on the table 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 statement.
- A COLLECT STATISTICS statement is blocked from obtaining its row-hash WRITE locks on DBC.TVFields or DBC.Indexes. These locks are requested near the completion of the COLLECT STATISTICS. This can cause other requests to be blocked in the parser if the data dictionary information for the table is not already in the cache. Note that once the lock is obtained, the COLLECT STATISTICS should finish quickly and parsing of the blocked requests should resume.

In general, COLLECT STATISTICS can run concurrently with a CREATE INDEX statement against the same table (as long as they are not for the same index). The exceptions are listed in the following bulleted items. These exceptions assume that no locking modifier on the table is used with the COLLECT STATISTICS statement (or if one is specified, it specifies ACCESS locking)

- If an explicit EXCLUSIVE LOCK is requested for the table for the CREATE INDEX statement, the CREATE INDEX statement will block COLLECT STATISTICS statements or vice versa depending on which one requests its lock on the table first. Note that COLLECT STATISTICS uses an ACCESS lock on the table by default. Usually, it is not recommended to specify an EXCLUSIVE lock on a table in conjunction with a CREATE INDEX statement.

- During final processing, CREATE INDEX requests an upgrade of its lock on the table to EXCLUSIVE (if it does not already have it). If any COLLECT STATISTICS statement is running against the table at the time the CREATE INDEX requests the upgrade of the lock on the table, the COLLECT STATISTICS statement will need to finish before the CREATE INDEX can complete, and any subsequent COLLECT STATISTICS against the table will have to wait to start until the CREATE INDEX can complete. Once the lock is obtained, CREATE INDEX should finish quickly and the blocked requests should resume running.
- COLLECT STATISTICS is blocked from obtaining its row hash WRITE locks on DBC.TVFields and DBC.Indexes. These locks are requested near the completion of the statement. These lock requests might cause other requests to be blocked if the data dictionary information for the table is not already cached. Once the lock is obtained, the statement should finish quickly and parsing of the blocked requests should resume.

Collecting Statistics When No COLUMN or INDEX Clause Is Specified

If no COLUMN or INDEX clause is specified, COLLECT STATISTICS updates, or recollects, statistics for columns and indexes that had been previously collected, with the following limitations:

- Statistics that were deleted by a DROP STATISTICS statement are not updated. To replace deleted statistics, specify the column or index names in a COLLECT STATISTICS statement.
- Statistics can be recollected on a combined maximum of 512 implicitly specified columns and indexes. An error results if statistics exist for more than 512 columns and indexes in the table.
- Teradata recollects statistics in the same mode (full file scan or sampled) that you specified when the information was originally collected.
- You cannot specify a USING SAMPLE clause to recollect statistics.

Collecting Statistics on Multiple Non-Indexed Columns

If you frequently perform queries with search conditions specified on multiple non-indexed columns, you should collect statistics on those columns jointly (see "Example 4: Collecting Statistics on Multiple Nonindexed Columns" on page 3-26). Statistics collected in this manner permit the Optimizer to more accurately estimate the number of qualifying rows for queries that specify those columns.

Note that the statistics generated for column_1 and column_2 when collected jointly are different from those generated 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 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.

The following table provides a general guideline for determining how to collect statistics on non-indexed columns.

IF a set of non-indexed columns is ...	THEN you should 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.

No more than 32 sets of non-indexed multicolumn statistics sets at a time can be collected and maintained concurrently for a table.

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 for Star Joins

You should always collect statistics on the primary 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 statement, 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 Hash and Join Indexes

Single- and multitable join indexes are fundamentally different database objects in many ways, and they are used for different purposes.

Column statistics for single-table join indexes and hash indexes are often best collected directly on the base table columns rather than separately as is always required for multitable join indexes.

The following table points to the various topics that explain the different recommendations for collecting statistics for hash indexes and single-table and multitable join indexes.

FOR this type of index ...	The following topics explain how best to collect statistics on the index ...
hash	• "Collecting Statistics on a Hash Index" on page 1-91. • "Collecting Statistics on Base Table Columns Instead of Hash Index Columns" on page 1-92. • "Guidelines for Collecting Statistics On Hash Index Columns" on page 1-93.
single-table join	• "Collecting Statistics on a Single-Table Join Index" on page 1-119. • "Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns" on page 1-120. • "Guidelines for Collecting Statistics On Single-Table Join Index Columns" on page 1-120.
multitable join	"Collecting Statistics on Multitable Join Indexes" on page 1-121.
all join indexes[a]	*Teradata Statistics Wizard User Guide*.

a. The Teradata Statistics Wizard does not support hash indexes.

Note that statistics for base tables and multitable join indexes are *not* interchangeable. If a given column is defined for both 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 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.

It is often useful to think of a multitable join index table as a special base table that represents a derived or pre-join. 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 to evaluate 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.

At minimum, you should collect statistics on the primary index of the join index. This provides 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 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 the 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 any regular join query.

Using Named Indexes

You can specify an index name or index definition in a COLLECT (or DROP STATISTICS) statement. To collect (or drop statistics) from an unnamed index, you must specify the complete index definition.

Examples

The following examples illustrate the use of COLLECT STATISTICS. In each case, the example is first given using the special syntax and then using the full syntax.

Example 1: Collecting Full Statistics

Assuming that an index is defined on the EmpNo and Name columns of the Employee table, the following statements collect statistics for that index.

```
COLLECT STATISTICS INDEX (EmpNo, Name) ON Employee;
COLLECT STATISTICS Employee INDEX (EmpNo, Name);
```

Example 2: Collecting Sampled Statistics

The following statements are similar to those of "Example 1: Collecting Full Statistics", but they collect sampled statistics on the index instead of scanning the entire table.

```
COLLECT STATISTICS USING SAMPLE INDEX (EmpNo, Name) ON
Employee;
COLLECT STATISTICS USING SAMPLE Employee INDEX (EmpNo, Name);
```

Example 3: Recollecting Statistics

Subsequent entry of the following statement on the same table refreshes the statistics for the EmpNo-Name index, along with any other statistics previously collected for the Employee table.

```
COLLECT STATISTICS ON Employee;
COLLECT STATISTICS Employee;
```

Example 4: Collecting Statistics on Multiple Nonindexed Columns

These statements collect statistics on two nonindexed columns of table_1.

```
COLLECT STATISTICS ON table_1 COLUMN (column_1, column_2);
COLLECT STATISTICS COLUMN (column_1, column_2) ON table_1;
```

Example 5: Named Indexes

This example collects statistics on an index named unique_1.

```
COLLECT STATISTICS INDEX unique_1 ON table_1;
COLLECT STATISTICS ON table_1 INDEX unique_1;
```

Example 6: Unnamed Indexes

The following example illustrates collecting statistics on an unnamed index that includes the index definition.

```
COLLECT STAT INDEX (field_1,field_2) ON table_1;
COLLECT STAT ON table_1 INDEX (field_1,field_2);
```

Example 7: Non-Valid Use of Named Index

These statements are not valid because the specified index does not exist for the table. The system returns an error to the requestor.

```
COLLECT STATISTICS INDEX no_such_index ON table_1;
COLLECT STATISTICS ON table_1 INDEX no_such_index;
```

For More Information

See the following for more information about collecting optimizer statistics:

- "DROP STATISTICS (Optimizer Form)" on page 3-31
- Online help for Teradata Manager Statistics Collection module
- Online help for Teradata Statistics Wizard
- *Teradata Statistics Wizard User Guide*

COLLECT STATISTICS (QCD Form)

Purpose

Collects demographic data for one or more columns and indexes of a table, computes a statistical profile of the collected data, and stores the synopsis in the TableStatistics table of the specified QCD database.

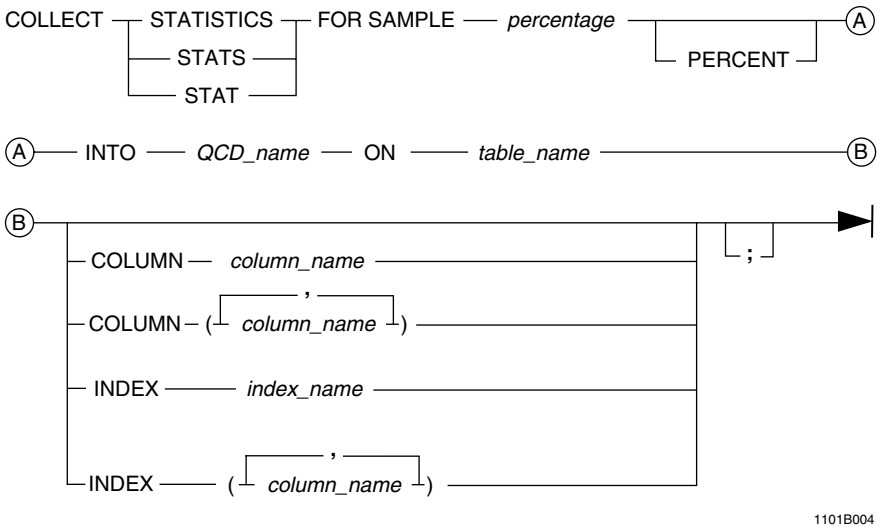
Statistics collected by this statement are used for index analysis and validation tasks performed by various database query analysis tools: they are *not* used by the Optimizer to process queries.

Invocation

Executable.

Normally invoked via client-based database query analysis tools.

Syntax



1101B004

where:

Syntax Element …	Specifies …
FOR SAMPLE *percentage*	the percentage of table rows to be sampled to build the statistics for the specified column and index sets. For example, if you specify a value of 5, then 5 percent of the rows for *table_name* on each AMP are read to collect their statistics. The specified value must be less than 100.

Syntax Element …	Specifies …
PERCENT	a noise keyword to indicate that the value specified for *percentage* is a percentage.
INTO *QCD_name*	the name of the QCD database into which the collected sampled statistics are to be written. The sampled statistics are written to the TableStatistics table of the specified QCD database.
table_name	the name of the table for which the specified column and index sampled statistics and sample size are to be collected.
COLUMN *column_name*	the name of the non-indexed column on which sampled statistics are to be collected.
INDEX *index_name*	the name of the index on which sampled statistics are to be collected.
INDEX *column_name*	the names of the columns in the column set on which sampled statistics are to be collected.

ANSI Compliance

COLLECT STATISTICS (QCD Form) is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have the following privileges to perform COLLECT STATISTICS (QCD Form):

- INDEX or DROP TABLE on *table_name* or its containing database.
- INSERT and UPDATE on the TableStatistics table or its containing QCD database.

COLLECT STATISTICS (QCD Form) Does Not Collect Statistics For The Optimizer

Unlike the standard COLLECT STATISTICS statement, COLLECT STATISTICS (QCD Form) does not write the statistics it collects into the data dictionary for use by the Optimizer. Instead, COLLECT STATISTICS (QCD Form) writes its statistics into the TableStatistics table of the specified QCD database, where the information is used by various database query analysis tools to perform index analysis and validation.

Because COLLECT STATISTICS (QCD Form) does not write its statistics to the data dictionary, it does not place dictionary locks. Through appropriate control of the sample size, you can collect sample statistics for index analysis during production hours without having a noticeable negative effect on system performance.

Implicit Recollection of QCD Statistics Is Not Supported

Unlike COLLECT STATISTICS (Optimizer Form), you cannot recollect statistics implicitly on indexes and columns for which statistics have been collected in the past. You must specify an explicit index or column name each time you collect statistics using COLLECT STATISTICS (QCD Form).

COLLECT STATISTICS (QCD Form) Is Not Treated As DDL By The Transaction Manager

You can place a COLLECT STATISTICS (QCD Form) statement anywhere within a transaction because it is not treated as a DDL statement by the Transaction Manager.

COLLECT STATISTICS (QCD Form) Collects New Statistics Even If Statistics Exist In the Data Dictionary or QCD

COLLECT STATISTICS (QCD Form) always collects its own fresh statistics on candidate index columns, even if a statistical profile of the specified columns or index columns already exists in the data dictionary or QCD for *table_name*, to ensure that index analyses are always performed using current statistics.

Fresh statistics are also collected whenever you perform an INSERT EXPLAIN statement and specify the WITH STATISTICS clause. See "INSERT EXPLAIN" on page 3-55 for further information.

Teradata Statistics Wizard and QCD

The Teradata Statistics Wizard does not support QCD.

Quality of Statistics as a Function of Sample Size

The larger the sample size used to collect statistics, the more likely the sampled statistics are an accurate representation of population statistics. There is a tradeoff for this accuracy: the larger the sample size, the longer it takes to collect the statistics.

Example 1

The following example collects statistics on a single-column NUSI named orderDateNUPI on the order_date column from a random sample of 10 percent of the rows in the orders table and writes them to the TableStatistics table of the QCD database named MyQCD.

```
COLLECT STATISTICS FOR SAMPLE 10 PERCENT
INTO MyQCD
ON orders INDEX orderDateNUPI;
```

Example 2

The following example collects sampled statistics on the same index as "Example 1" on page 3-30 using different syntax:

```
COLLECT STATISTICS FOR SAMPLE 10 PERCENT
INTO MyQCD
ON orders INDEX (order_date);
```

For More Information

See the following topics for more information about index analysis:

- "COLLECT DEMOGRAPHICS" on page 3-11
- "INITIATE INDEX ANALYSIS" on page 3-46
- "INSERT EXPLAIN" on page 3-55
- "RESTART INDEX ANALYSIS" on page 3-62
- The chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2*.
- The chapter "Query Capture Facility" in *Teradata RDBMS SQL Reference, Volume 2*.

See the following related statements for more information:

- "DROP STATISTICS (QCD Form)" on page 3-37.
- "HELP STATISTICS (Optimizer Form)" on page 4-60.

See the following manuals for more information about client-based query analysis tools:

- *Teradata Index Wizard User Guide*
- *Teradata Statistics Wizard User Guide*

DROP STATISTICS (Optimizer Form)

Purpose

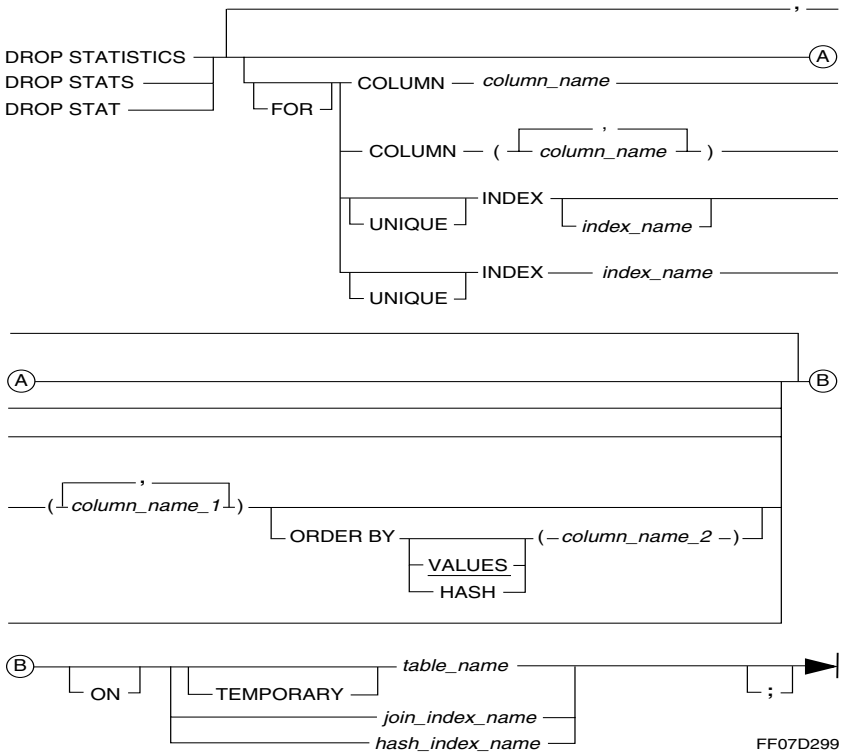
Drops the statistical data that was created for one or more columns of a table by a COLLECT STATISTICS (Optimizer Form) statement.

Also see "COLLECT STATISTICS (Optimizer Form)" on page 3-14.

Invocation

Executable.

Recommended Syntax



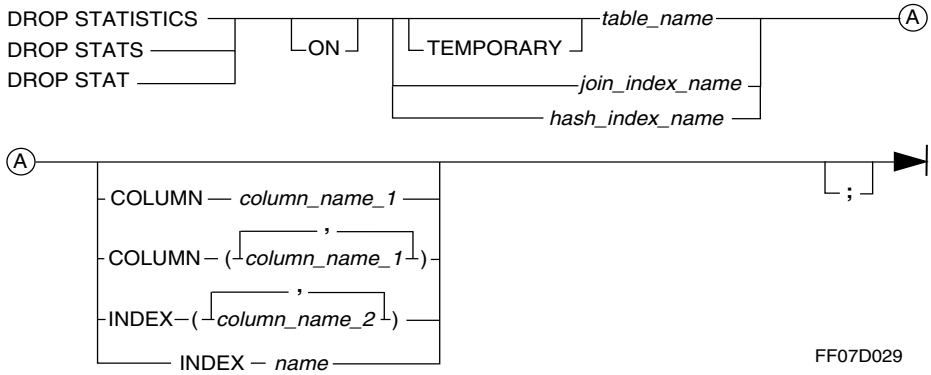
where:

Syntax Element …	Specifies …
COLUMN	that statistics are to be dropped for the specified column set.
column_name	the names of the non-indexed columns for which statistics are to be dropped.
UNIQUE	the index for which statistics are to be dropped is unique.

Syntax Element …	Specifies …
INDEX	that statistics are to be dropped for an index on column set *column_name_1*.
index_name	the name of the named index for which statistics are to be dropped. Note that you only specify column names with a named index when it is defined with an ORDER BY clause.
column_name_1	the name of the column set on which the index exists and for which statistics are to be dropped.
ORDER BY	that index rows are ordered on each AMP by a single value- or hash-ordered column. If you specify ORDER BY, then VALUES, HASH, *column_name_2*, or a combination of VALUES or HASH with *column_name_2* must also be specified.
VALUES	value-ordering for the ORDER BY column. If you specify neither VALUES nor HASH, then VALUES is used as the default.
HASH	hash-ordering for the ORDER BY column.
column_name_2	a single column from the *column_name_1* list that specifies the sort order to be used for either HASH or VALUES. If you specify VALUES, then the size of the column is limited to four or fewer bytes. Supported data types for *column_name_2* are the following: • DATE • INTEGER • SMALLINT • BYTEINT • DECIMAL If you omit *column_name_2*, the first column in the *column_name_1* list is used by default.
TEMPORARY	that statistics are to be dropped for a materialized temporary table. This keyword is not valid for permanent tables and if you use it with a table defined as permanent, an error is returned. If you do not specify TEMPORARY, but the referenced table is a temporary table, the statistics are dropped, but on the base temporary table rather than its materialized instance. If you specify TEMPORARY correctly, but no materialized instance of the definition exists in the system, Teradata materializes an instance with all of the statistics defined on the base temporary table before dropping statistics for it.

Syntax Element …	Specifies …
table_name	the name of the table or temporary table for which column or index statistics are to be dropped.
join_index_name	the name of the join index for which statistics are to be dropped.
hash_index_name	the name of the hash index for which statistics are to be dropped.

Legacy Syntax



where:

Syntax Element …	Specifies …
TEMPORARY	that statistics are to be dropped on a materialized temporary table named *table_name*. This keyword is not valid for permanent tables and if you use it with a table defined as permanent, an error is returned. If you do not specify TEMPORARY, but the referenced table is a temporary table, the statistics are dropped, but on the base temporary table rather than its materialized instance. If you specify TEMPORARY correctly, but no materialized instance of the definition exists in the system, Teradata materializes an instance with all of the statistics defined on the base temporary table before dropping statistics for it.
table_name	the name of the table on which statistics are to be dropped.
join_index_name	the name of the join index for which statistics are to be dropped.
COLUMN	that statistics are to be dropped for a non-indexed column set.
column_name	the name of the non-indexed columns for which statistics are to be dropped.
INDEX	that the statistics to be dropped are for one or more columns that comprise an index on the table.
column_name2	the index columns for which statistics are to be dropped.

Syntax Element …	Specifies …
index_name	the name of the index for which statistics are to be dropped. When you specify a named index, you do not specify column names.

ANSI Compliance

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

Authorization

To use DROP STATISTICS on a …	You must have INDEX or DROP TABLE privileges on …
table	the table.
• hash index • join index	either of the following: • The hash or join index • Any of the base tables of the hash or join index

Global Temporary and Volatile Tables

You can drop statistics on a global temporary table but not for a volatile table.

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 transaction enclosed by BEGIN TRANSACTION and END TRANSACTION statements.

When to Use DROP and COLLECT STATISTICS

If you suspect that your table statistics are not current, then use DROP STATISTICS to delete them from the data dictionary.

Note that performing COLLECT STATISTICS 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 re-collect statistics for an entire table.

When you perform DROP STATISTICS on a column or index, you must perform a COLLECT STATISTICS statement that specifies the COLUMN or INDEX option to collect new demographics for those columns.

You must specify either the COLUMN or INDEX option because a COLLECT STATISTICS statement that only lists a table name recomputes only the existing statistics for a table.

The Teradata Statistics Wizard client utility provides recommendations for collecting and dropping statistics based on particular SQL statement workloads. See *Teradata Statistics Wizard User Guide* for further information.

Advantages of the Preferred Over the Legacy Syntax

The preferred syntax for DROP STATISTICS has several advantages over the older form.

It permits statistics to be collected for multiple columns for a table within a single SQL statement.

The syntax is nearly identical to that for CREATE INDEX, permitting you to copy and modify an existing CREATE INDEX statement for collecting statistics on indexed columns.

Because of its similarity to CREATE INDEX, the syntax is easier to remember.

Single Column Index = INDEX (*column_name*)

If an index has a single column, *column_name*, dropping statistics for COLUMN *column_name* is the same as dropping statistics for INDEX (*column_name*).

If neither COLUMN nor INDEX is specified, the effect of the statement is to drop any statistics that exist for the table.

The following are rules for using named indexes with the COLLECT and DROP STATISTICS statements:

- You can specify an index name in the DROP STATISTICS statement.
- You can specify multiple non-indexed columns in the DROP STATISTICS statement.
- To COLLECT or DROP STATISTICS from an unnamed index, you must specify the complete index definition.

Examples

The following examples illustrate the use of DROP STATISTICS. In each case, the example is first given using the preferred syntax and then using the legacy syntax.

Example 1

The following statements drop statistics on an index defined on the EmpNo and Name columns of the Employee table:

```
DROP STATISTICS INDEX(EmpNo, Name) ON Employee;
DROP STATISTICS Employee INDEX(EmpNo,Name);
```

Example 2

The following statements drop any statistics that exist for the Employee table.

```
DROP STATS ON Employee;
DROP STATS Employee;
```

Example 3: Named Index

The following example drops any statistics on a named index.

```
DROP STATISTICS INDEX unique_1 ON table_1;
DROP STATISTICS ON table_1 INDEX unique_1;
```

Example 4: Defined Index

The following example drops any statistics on a defined index.

```
DROP STATISTICS INDEX (field_1,field_2) ON table_1;
DROP STATISTICS ON table_1 INDEX (field_1,field_2);
```

Example 5: Non-Valid

These statements are not valid because the index named does not exist for the table.

```
DROP STATISTICS INDEX no_such_index ON table_1;
DROP STATISTICS ON table_1 INDEX no_such_index;
```

Teradata returns the error message:

```
The specified constraint name does not exist in table.
```

DROP STATISTICS (QCD Form)

Purpose

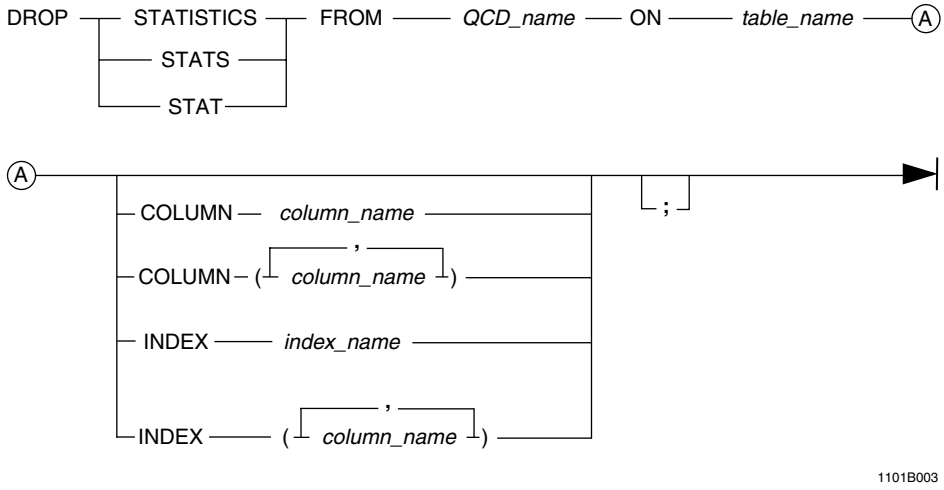
Drops sampled statistics on the specified table from TableStatistics in the specified *QCD_name*.

Invocation

Executable.

Normally invoked via client-based database query analysis tools.

Syntax



where:

Syntax Element …	Specifies …
FROM *QCD_name*	the name of the QCD database from which statistics on the specified table columns and indexes are to be dropped from the TableStatistics table.
table_name	the name of the table for which sampled column and index statistics are to be dropped.
COLUMN *column_name*	the set of non-indexed columns for which sampled statistics are to be dropped.
INDEX *index_name*	the name of the index for which sampled statistics are to be dropped.
INDEX *column_name*	the column set for which sampled statistics are to be dropped.

ANSI Compliance

DROP STATISTICS (QCD Form) is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have the following privileges to perform DROP STATISTICS (QCD Form):

- INDEX or DROP TABLE on *table_name* or its containing database
- DELETE on the TableStatistics table or its containing QCD database

DROP STATISTICS (QCD Form) Does Not Drop Statistics Used By The Optimizer

Unlike the standard DROP STATISTICS statement, DROP STATISTICS (QCD Form) does not drop the statistics kept in the data dictionary for use by the Optimizer. Instead, DROP STATISTICS (QCD Form) drops statistics from the TableStatistics table of the specified QCD database.

DROP STATISTICS (QCD Form) Does Not Drop Statistics Collected During Query Capture

DROP STATISTICS (QCD Form) does not drop statistics from the QCD that were captured using an INSERT EXPLAIN WITH STATISTICS statement. These statistics are dropped automatically whenever their corresponding query plan is deleted from the QCD.

DROP STATISTICS (QCD Form) Is Not Treated As DDL By The Transaction Manager

You can place a DROP STATISTICS (QCD Form) statement anywhere within a transaction because it is not treated as a DDL statement by the Transaction Manager.

Example 1

The following example drops statistics on all columns and indexes for the orders table from the TableStatistics table of the QCD database named MyQCD.

```
DROP STATISTICS FROM MyQCD
ON orders;
```

Example 2

The following example drops statistics on a single-column NUSI named orderDateNUPI from the TableStatistics table of the QCD database named MyQCD.

```
DROP STATISTICS FROM MyQCD
ON orders INDEX orderDateNUPI;
```

For More Information

See the following related statements for more information:

- "COLLECT STATISTICS (QCD Form)" on page 3-27
- "HELP STATISTICS (QCD Form)" on page 4-63

DUMP EXPLAIN

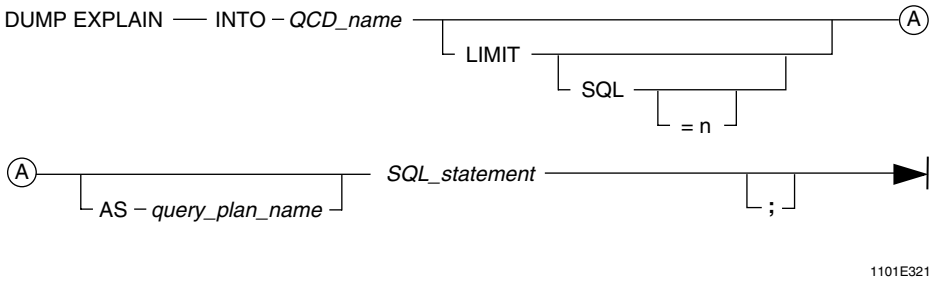
Purpose

Captures the Optimizer plan information for a query and returns it to the requestor in the form of a script containing a series of INSERT statements.

Invocation

Executable.

Syntax



where:

Syntax element . . .	Specifies . . .
INTO *QCD_name*	a user-defined query capture database to be used instead of the default QCD database. The database named *QCD_name* need not exist on the target system; however, a database named *QCD_name* must exist on the test system on which the generated script is performed. Use the Control Center feature of the Visual EXPLAIN tool to create your QCD databases.
LIMIT [SQL [=*n*]]	to place a limit on the size of the query, DDL, and view text captured for the QCD tables Query, Relation, and ViewText, respectively The value for *n* indicates the upper limit on the amount of text to capture. If you specify LIMIT SQL without a value, then the clause means LIMIT SQL = 0, and no text captured is captured. If you do not specify this clause, then complete text is captured.

Syntax element ...	Specifies ...
AS *query_plan_name*	an optional user-defined name under which the query plan information is to be stored. If no *query_plan_name* is specified, the query plan information is stored with a null name. Because each query plan is stored with a unique non-null Query ID (see "Query" in the chapter "Query Capture Facility" in *Teradata RDBMS SQL Reference, Volume 2*), there is no problem distinguishing among the various query plans within a given database. Note that Query IDs are *not* unique across databases. *query_plan_name* is not constrained to be unique. *query_plan_name* cannot exceed 30 characters in length. You can store *query_plan_name* as *query plan name* if you enclose the pad character-separated words in QUOTATION MARKS as "query plan name".
SQL_statement	the DML statement whose Optimizer plan information is to be captured and returned to the requestor as a series of INSERT statements. *SQL_statement* is limited to the following four statements. • DELETE • INSERT • SELECT • UPDATE

ANSI Compliance

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

Authorization

You must have INSERT privileges on all the tables in the specified query capture database.

Before You Perform DUMP EXPLAIN for the First Time

Ensure that there is an existing *QCD_name* database on the test system before running the script produced by DUMP EXPLAIN for the first time. Use the Control Center feature of the Visual EXPLAIN tool to create your QCD databases.

If a *QCD_name* database or any of its tables does not exist, or if you do not have INSERT privileges on one or more *QCD_name* database tables when you perform the script produced by DUMP EXPLAIN, then Teradata displays an appropriate error message and terminates performance of the script.

What DUMP EXPLAIN Does

DUMP EXPLAIN performs the following actions in the order indicated.

1. Runs an EXPLAIN on the SQL DML statement specified by *SQL_statement*.
2. Captures the Optimizer plan output of that EXPLAIN.
3. Returns the output to the requestor as a script containing a series of INSERT statements designed to be used to update the appropriate tables in a query capture or other user-specified database.

You might want to use DUMP EXPLAIN rather than INSERT EXPLAIN if you are collecting information from several different machines and you want to ensure that only the selected QCD on the appropriate machine is updated with the results or if you do not want to update the QCD during heavy workload windows. In this case, you could submit the INSERT statements as part of a batch job during a less burdened workload window.

Examples

The following example set uses the same SELECT statement throughout to illustrate how the different syntax options of DUMP EXPLAIN produce different results.

Each DUMP EXPLAIN statement generates a set of INSERT statements for the query plan produced by the Optimizer for the SELECT statement that it modifies.

Example 1

The output of this DUMP EXPLAIN statement is referenced under the query plan name EmployeeSmithQuery in the TLE_queries database.

```
DUMP EXPLAIN INTO TLE_queries AS EmployeeSmithQuery
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

Example 2

The output of this DUMP EXPLAIN statement has a null query plan name in the TLE_queries database. The difference between this result and that of "Example 1" on page 3-42 is that the query plan for this example has no name.

```
DUMP EXPLAIN INTO TLE_queries
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

Example 3

The output of this DUMP EXPLAIN statement is referenced under the query plan name Employee Smith Query in the TLE_queries database.

```
DUMP EXPLAIN INTO TLE_queries AS "Employee Smith Query"
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

Related Statement

See "INSERT EXPLAIN" on page 3-55 for information on how to populate query capture databases automatically with Optimizer plan information rather than returning the information in a series of INSERT statements.

Further Information

See the following topics for more information about index analysis:

- "COLLECT DEMOGRAPHICS" on page 3-11
- "COLLECT STATISTICS (QCD Form)" on page 3-27
- "INITIATE INDEX ANALYSIS" on page 3-46
- "INSERT EXPLAIN" on page 3-55
- "RESTART INDEX ANALYSIS" on page 3-62
- The chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2*
- The chapter "Query Capture Facility" in *Teradata RDBMS SQL Reference, Volume 2*
- *Teradata Index Wizard User Guide*

END QUERY LOGGING

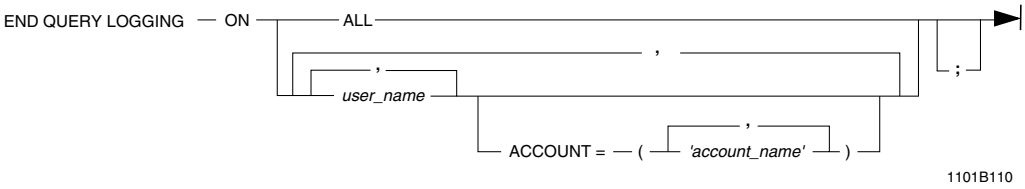
Purpose

Stops the logging of SQL queries initiated by the BEGIN QUERY LOGGING statement and commits the query log cache.

Invocation

Executable.

Syntax



where:

Syntax element …	Specifies …
ALL	to stop logging query information for all users specified by a rule created by a BEGIN QUERY LOGGING ON ALL statement.
user_name	the name of a specific user or set of users for whom logging of SQL query information is to be stopped.
account_name	the name of one or more specific accounts for which logging of SQL query information is to be stopped. Account names must be enclosed by LEFT and RIGHT PARENTHESIS characters. When you specify a list of accounts, each account name must be delimited by APOSTROPHE characters and separated by COMMA characters. If you began query logging on a specific user-account pair, then you must also specify that user-account pair to end query logging.

ANSI Compliance

END QUERY LOGGING is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have EXECUTE privileges on the macro DBQLAccessMacro to perform END QUERY LOGGING.

Determining Whether an Account Was Specified for BEGIN QUERY LOGGING

If you begin query logging on a specify account or account set, then you must also specify that account or account set to end query logging. You can determine whether an account has been specified explicitly or not by performing the following query:

```
SELECT *
FROM DBQLRules;
```

This query reports all active rules, including whether accounts are specified for them or not.

Example 1

Stop logging query information for all users specified by a rule created by a BEGIN QUERY LOGGING ON ALL statement.

```
END QUERY LOGGING ON ALL;
```

Example 2

Stop logging query information for user_1.

```
END QUERY LOGGING ON user_1;
```

Example 3

Stop logging query information for user_1 and user_2.

```
END QUERY LOGGING ON user_1, user_2;
```

Example 4

If you started query logging on a specific account, then you must also specify that account explicitly when you end query logging.

This statement stops logging query information for user_1 logged on with account order_entry.

```
END QUERY LOGGING ON user_1 ACCOUNT = ('order_entry');
```

For More Information

See "BEGIN QUERY LOGGING" on page 3-4.

INITIATE INDEX ANALYSIS

Purpose

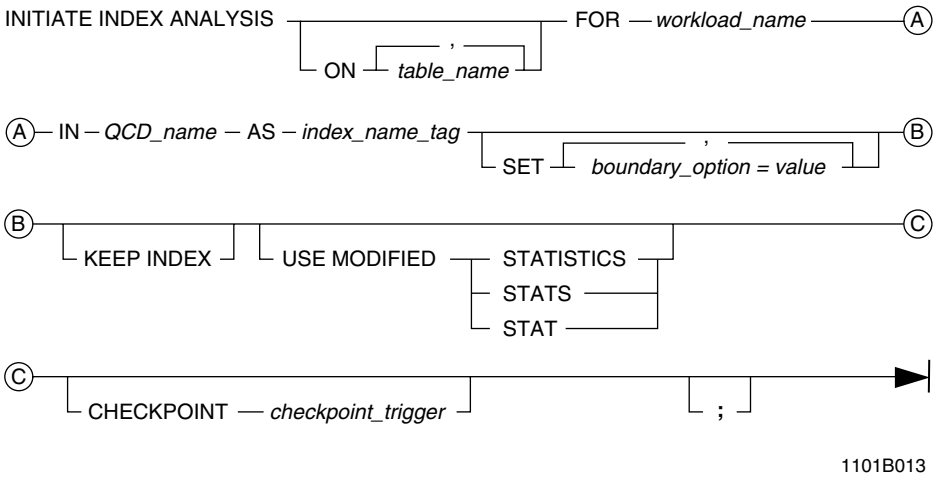
Analyzes a query workload and generates a set of recommended indexes for optimizing its processing.

Invocation

Executable.

Normally invoked via the Teradata Index Wizard utility.

Syntax



where:

Syntax element …	Specifies …
table_name	the table set to be analyzed for index recommendations. If no table is specified, then all tables referenced in the specified *workload_name* are analyzed for index recommendations. This option permits you to implicitly instruct INITIATE INDEX ANALYSIS not to create indexes on certain tables.
workload_name	the name of the workload to which the queries to be analyzed belong.
QCD_name	the QCD workload database in which *workload_name* exists. INITIATE INDEX ANALYSIS stores the index recommendations for the specified workload in this QCD database.
index_name_tag	a name to be assigned to the index recommendations within *QCD_name*.

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>boundary_option
= value</td><td>a clause that sets upper bounds on the specified option.

The following options and maximum bounds are valid specifications:

<table>
<tr><th>Option</th><th>Definition</th></tr>
<tr><td>IndexesPerTable</td><td>The maximum number of indexes that can be defined for a table referenced in this workload.

The number includes the number of indexes currently defined on the table, so this value defines the upper limit for the total number of indexes defined on the table, not just a limit on the number of newly recommended indexes.

The default is 4.

The system limit is 32.</td></tr>
<tr><td>SearchSpace</td><td>The maximum number of candidate indexes to be evaluated for a given table.

The default is 256.

The valid range is 128 to 512.</td></tr>
<tr><td>ChangeRate</td><td>The threshold limit on whether a column can or cannot participate in an index.

The default is 5.

The valid range is 0 to 9, where 0 means the column is not updated and 9 means the column is highly volatile with respect to the specified workload.</td></tr>
<tr><td>ColumnsPerIndex</td><td>The maximum number of columns allowed for a composite index.

The default is 4.

The valid range is 1 to 64.</td></tr>
</table></td></tr>
<tr><td>KEEP INDEX</td><td>that the index recommendations are not to contain any DROP INDEX or DROP STATISTICS recommendations.

The default is to recommend DROP INDEX and DROP STATISTICS recommendations when the analysis indicates their usefulness.</td></tr>
</table>

Syntax element …	Specifies …
USE MODIFIED STATISTICS	to perform the index analysis with the user-modified statistics stored in the ModifiedStats column of QCD table TableStatistics rather than the statistics gathered with COLLECT STATISTICS (QCD Form). If the ModifiedStats column is not populated, then the statistics stored in StatisticsInfo are used instead and the result is the same as if you had not specified USE MODIFIED STATISTICS. This option permits you to perform speculative "what if" analyses on the data when you want to anticipate how the Optimizer will handle various situations you anticipate might occur.
CHECKPOINT *checkpoint_trigger*	the number of queries after which a checkpoint snapshot is to be taken. The value for *checkpoint_trigger* must be a positive integer.

IF the value is …	THEN …
0	the CHECKPOINT clause is ignored.
> the total number of queries to be analyzed	no checkpoint snapshot is taken.

ANSI Compliance

INITIATE INDEX ANALYSIS is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have the INSERT privilege on the IndexRecommendations and IndexColumns tables in QCD or on database *QCD_name*.

If you specify a checkpoint trigger, you must have both INSERT and DELETE privileges on the AnalysisLog table.

Rules for Performing INITIATE INDEX ANALYSIS

The following set of rules applies to INITIATE INDEX ANALYSIS:

- The specified QCD database must exist in the system. Otherwise, an error is returned.
- The specified workload must exist in the specified *QCD_name* database. Otherwise, an error is returned.
- If you specify a table list, then only that list of tables is considered for index analysis.

  If the workload does not contain any tables that match at least one table in the table list, then an error is reported. Otherwise, only the matching list of tables is considered for index analysis.
- A maximum of 1 024 tables can be analyzed per workload.
- No error is reported if the same table is specified more than once for index analysis.
- The index analysis parameters must not be repeated.

  If they are repeated, or if a value is out of range for a parameter, then an error is reported.
- If the session is logged off or restarted while INITIATE INDEX ANALYSIS is being performed and no checkpoint is specified, then no index recommendations are saved.

  You must perform the INITIATE INDEX ANALYSIS statement again in the restarted session to create the index recommendations.
- You must disable access logging on any referenced database before performing INITIATE INDEX ANALYSIS. If you attempt to perform an INITIATE INDEX ANALYSIS statement that references a database on which access logging is enabled, Teradata returns an error message and does not undertake the index analysis.

  See "END LOGGING" on page 1-347 for information about disabling access logging and "BEGIN LOGGING" on page 1-67 for information about re-enabling access logging after an index analysis has completed.

Sequence of Events for an Index Analysis

The following table indicates the general phases of an index analysis.

Stage	Process
1	The analysis begins when an INITIATE INDEX ANALYSIS statement is submitted.
2	The database query analysis tool analyzes the submitted workload for indexing opportunities.
3	The index recommendations are written to the IndexRecommendations and IndexColumns tables of the specified QCD database.

Stage	Process

4	The recommendations are retrieved for evaluation using a set of SQL SELECT statements defined within a number of macros provided within each QCD.
	
	For example, the following SELECT statement is the core of the Recommendations macro.
	
	`SELECT TableName, DatabaseName, IndexTypeText,`
	`SpaceEstimate, IndexDDL`
	`FROM IndexRecommendations`
	`WHERE WorkloadID = :WorkloadID;`
	
	You can find a complete list of the index recommendation query macros in the chapter "Database Query Analysis" in *Teradata RDBMS SQL Reference, Volume 2*.

Retrieving a Completed Index Analysis

After the index analysis completes successfully, you can retrieve the index recommendations from the IndexRecommendations and IndexColumns tables in the appropriate QCD.

Using the CHECKPOINT Option

For long index analyses or cases where the workload has many SQL statements that must be analyzed, you should always specify a checkpoint using the CHECKPOINT option.

The checkpoint trigger value indicates the frequency with which a checkpoint is logged in the AnalysisLog table. A checkpoint is taken after every *n* statement occurrences, where *n* is the checkpoint trigger value, and the interim index analysis information is logged.

See "RESTART INDEX ANALYSIS" on page 3-62 for information about how to restart a checkpointed index analysis.

How ChangeRate Is Evaluated by INITIATE INDEX ANALYSIS

The following table indicates the method INITIATE INDEX ANALYSIS uses to evaluate column change rate values for index recommendation. Only those columns with a change rate rank ≤ the specified change rate rank are considered for recommendation as an index or composite index component. If no change rate is specified, the default value is 5.

Change Rate Rank	Description	Percentage of Updates on Column
0	No updates on the column.	0
1	1 update per 1 000 000 selects on the table.	.0001
2	1 update per 100 000 selects on the table.	.001
3	1 update per 10 000 selects on the table.	.01
4	1 update per 1 000 selects on the table.	.1
5	1 update per 100 selects on the table.	1
6	5 updates per 100 selects on the table.	5
7	10 updates per 100 selects on the table.	10
8	25 updates per 100 selects on the table.	25
9	More than 25 updates per 100 selects on the table.	>25

Example 1

Assume that the QCD named MyQCD exists on the system. The following INITIATE INDEX ANALYSIS statement performs successfully.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;
```

If MyQCD does not exist in the system, the same request fails.

```
INITIATE INDEX ANALYSIS ON table_1
FROM MyWorkload
IN MyQCD
AS table_1Index;
*** Failure 3802 Database 'MyQCD' does not exist.
```

Example 2

Assume that the workload named MyWorkload exists in MyQCD. The following INITIATE INDEX ANALYSIS statement performs successfully.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;
```

If MyWorkload does not exist in MyQCD, the same request fails.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;
*** Failure 5637 Specified workload does not exist in 'MyQCD'
database.
```

Example 3

Assume that you have the INSERT privilege on the IndexRecommendations and IndexColumns tables or on database QCD_name. The following INITIATE INDEX ANALYSIS statement performs successfully.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;
```

If you do not have INSERT privilege on MyQCD, the same request fails.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;
*** Failure 3523 The user does not have INSERT access to
MyQCD.IndexRecommendations.
```

Example 4

When you specify a table list, then only that list of tables is considered for index analysis. If the workload does not have any tables matching at least one table in the list, then an error is reported. Otherwise, only the matching list of tables is considered for selecting the indexes.

Assume that MyWorkload describes the workload for tables, table_1, table_2 and table_3. The following INITIATE INDEX ANALYSIS statement performs successfully and the recommendations are considered only for table_1 and table_2 because table_3 is not specified in the table list.

```
INITIATE INDEX ANALYSIS ON table_1, table_2
FOR MyWorkload
INTO MyQCD
AS table_1Index;
```

Suppose MyWorkload describes the workload for table_1 and table_2 only. The following INITIATE INDEX ANALYSIS statement fails because table_3 and table_4 are not defined in the workload.

```
INITIATE INDEX ANALYSIS ON table_3, table_4
FOR MyWorkload
IN MyQCD
AS table_1Index;
*** Failure 5638 No match found for specified tables in the
workload 'MyWorkload'.
```

Example 5

No error is reported if the same table is specified more than once for index selection. In the following example, table_1 is specified twice.

```
INITIATE INDEX ANALYSIS ON table_1, table_2, table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;
```

Example 6

If index analysis parameters are specified, they must not be repeated. If the parameters are repeated, an error is reported.

In the following example, the only index analysis parameter specified, IndexesPerTable, is specified once.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index
SET IndexesPerTable = 2;
```

In the following example, the index analysis parameter IndexesPerTable is specified twice, so an error is returned.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index
SET IndexesPerTable=2, ChangeRate=4, IndexesPerTable=4;
*** Failure 3706 Syntax error: Multiple "IndexesPerTable"
options.
```

Note that the error is not caused by the discrepancy between the specified boundary conditions for the IndexesPerTable parameter: the same error would be reported if the boundary conditions matched in both specifications.

Example 7: Using a CHECKPOINT

You should specify the CHECKPOINT option for long operations and for workloads containing many SQL statements that must be analyzed. The *checkpoint_trigger* value indicates the frequency with which a checkpoint is taken. After every *checkpoint_trigger* number of SQL statements, a checkpoint is taken and completed IndexRecommendation information is saved in the AnalysisLog table in the QCD.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index
SET "IndexesPerTable" = 2
KEEP INDEX
CHECKPOINT 10;
```

For More Information

See the following topics for more information about index analysis:

- “COLLECT DEMOGRAPHICS” on page 3-11.
- “COLLECT STATISTICS (QCD Form)” on page 3-27.
- “RESTART INDEX ANALYSIS” on page 3-62.
- “INSERT EXPLAIN” on page 3-55.
- The chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2*.
- The chapter "Query Capture Facility" in *Teradata RDBMS SQL Reference, Volume 2*.
- *Teradata Index Wizard User Guide.*

INSERT EXPLAIN

Purpose

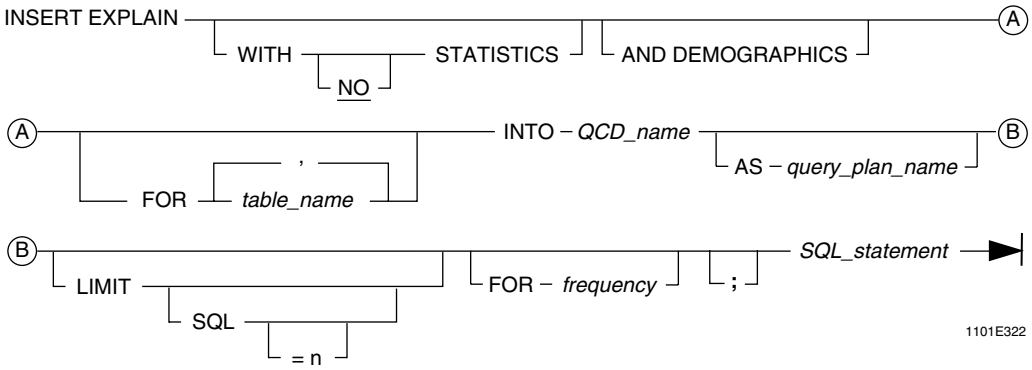
Captures the Optimizer white tree information for a query and writes it to a user-defined query capture database.

Invocation

Executable.

Normally invoked via client-based database query analysis tools with the WITH STATISTICS, FOR *table_name*, and FOR *frequency* clauses.

Syntax



where:

Syntax element …	Specifies …
WITH STATISTICS	to capture the statistics for the table set accessed by *SQL_statement*. Specifying this option is equivalent to performing COLLECT STATISTICS (QCD Form) on the query table columns. The columns inserted into the QCD are those determined to be index candidates by the Teradata Index Wizard.
WITH [NO] STATISTICS	not to capture statistics for the table set accessed by *SQL_statement*.
AND DEMOGRAPHICS	to collect table demographics as well as statistics. Specifying this option is equivalent to performing COLLECT DEMOGRAPHICS on the query tables. If you do not specify this option, then table demographics are not collected.

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>FOR table_name</td><td>the set of tables for which data demographics and statistics details are either to be included or excluded for analysis by the Teradata Index Wizard, depending on the WITH [NO] STATISTICS clause specification.

Statistics are collected on columns that are index candidates[a] and for all columns and indexes referenced explicitly with values in SQL_statement regardless of whether you specify WITH STATISTICS or WITH NO STATISTICS.

<table>
<tr><th>IF this clause is specified …</th><th>THEN …</th></tr>
<tr><td>WITH NO STATISTICS</td><td>the specified tables are excluded from the collection of column statistics and data demographics.

Note that column statistics and data demographics are collected for any tables referenced in SQL_statement that are not specified here.</td></tr>
<tr><td>WITH STATISTICS</td><td>column statistics and data demographics are to be collected only on the tables specified in the table list.</td></tr>
</table>
</td></tr>
<tr><td>INTO QCD_name</td><td>a user-defined query capture database.

A database named QCD_name must exist on both the target system and the test system on which the generated script is performed.

Use the Control Center feature of the Visual EXPLAIN tool to create your QCD databases.</td></tr>
<tr><td>AS
query_plan_name</td><td>an optional user-defined name under which the query plan information is to be stored.

If no query_plan_name is specified, the query plan information is stored with a null name. Because each query plan is stored with a unique non-null Query ID (see "Query" in the chapter "Query Capture Facility" in Teradata RDBMS SQL Reference, Volume 2), there is no problem distinguishing among the various query plans within a given database. Note that Query IDs are not unique across databases.

query_plan_name is not constrained to be unique.

query_plan_name cannot exceed 30 characters in length.

You can store query_plan_name as query plan name if you enclose the pad character-separated words in QUOTATION MARKS as "query plan name".</td></tr>
</table>

Syntax element …	Specifies …
LIMIT [SQL [=*n*]]	to place a limit on the size of the query, DDL, and view text captured for the QCD tables Query, Relation, and ViewText, respectively. The value for *n* indicates the upper limit on the amount of text to capture. If you specify LIMIT SQL without a value, then the clause means LIMIT SQL = 0, and no text is captured. If you do not specify this clause, then complete text is captured.
FOR *frequency*	the number of times *SQL_statement* is typically performed within its identified workload. This value is used to weight the respective benefits of each column analyzed for inclusion in the index recommendation computed by the Teradata Index Wizard utility. Any positive integer up to 4 bytes is valid. If this clause is not specified, then *frequency* defaults to 1.
SQL_statement	the DML statement whose Optimizer white tree information is to be captured and written to the QCD. *SQL_statement* is limited to the following four statements. • DELETE • INSERT • SELECT • UPDATE

a. A candidate index column is defined as a column for which the value or join range access frequencies stored in the Field table of the specified QCD are greater than 0.

ANSI Compliance

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

Authorization

You must have INSERT privileges on the tables in the user-defined query capture database.

You must also have the SELECT privilege on all referenced tables to capture data demographics and column statistics.

Rules

The following rules apply to the use of INSERT EXPLAIN:

- When you specify a table list, but none of the tables referenced by *SQL_statement* are in the specified table list, an error is returned to the requestor.
- When you specify a table list, but not all of the tables referenced by *SQL_statement* are in the specified table list, no error is returned, but the nonmatching tables are ignored.

Before You Perform INSERT EXPLAIN for the First Time

Ensure that there is an existing *QCD_name* database before running INSERT EXPLAIN for the first time. Use the Control Center feature of the Visual EXPLAIN tool to create your QCD databases.

If a *QCD_name* database or any of its tables does not exist, or if you do not have the appropriate privileges on one or more *QCD_name* database tables when you perform the INSERT EXPLAIN statement, then Teradata displays an appropriate error message and terminates performance of the statement.

Always Use the WITH STATISTICS AND DEMOGRAPHICS and FOR *frequency* Options for Creating Index Analysis Workloads

When you specify the WITH STATISTICS AND DEMOGRAPHICS option with INSERT EXPLAIN, sample table cardinality estimates and column statistics are captured in the DataDemographics and TableStatistics tables of your QCD databases. This information is critical to the index recommendation analyses performed by the Teradata Index Wizard.

Note that the sampled statistics captured using the WITH STATISTICS clause are identical to those captured by performing a COLLECT STATISTICS (QCD form) statement and a COLLECT DEMOGRAPHICS statement. See "COLLECT STATISTICS (Optimizer Form)" on page 3-14 and "COLLECT DEMOGRAPHICS" on page 3-11 for further information.

The FOR *frequency* clause provides an estimate of the average number of times the SQL statement being analyzed is typically performed during the submitted workload. This information is used by the Teradata Index Wizard utility to determine the benefit of including various columns as indexes.

INSERT EXPLAIN WITH STATISTICS Collects New Statistics Even If Statistics Exist In the Data Dictionary or QCD

INSERT EXPLAIN WITH STATISTICS always collects its own fresh statistics on candidate index columns, even if a statistical profile of the specified columns or index columns already exists in the data dictionary or QCD for *table_name*, to ensure that index analyses are always performed using current statistics.

What INSERT EXPLAIN Does

INSERT EXPLAIN performs the following actions in the order indicated:

1. Runs an EXPLAIN on the SQL DML statement specified by *SQL_statement*.
2. Captures the Optimizer white tree output of that EXPLAIN.
3. Writes the output to the appropriate tables in the user-specified query capture database.

If you are collecting demographics for use by the Teradata Index Wizard utility, then INSERT EXPLAIN performs the following actions in the order indicated:

1. Runs an EXPLAIN on the SQL DML statement specified by *SQL_statement*.
2. Captures the Optimizer white tree output of that EXPLAIN.
3. Captures the table cardinality and column statistics for the tables referenced by *SQL_statement*, excluding any tables explicitly excluded by the FOR *table_name* clause. The columns are those identified as potential index candidates.
4. Captures the number of times *SQL_statement* is performed in the identified workload as specified by the FOR *frequency* clause.
5. Writes the output to the appropriate tables in the user-specified query capture database.

Examples

The following example set uses the same SELECT statement throughout to illustrate how the different syntax options of INSERT EXPLAIN produce different results.

Each INSERT EXPLAIN statement inserts the data for the query plan produced by the Optimizer for the SELECT statement that it modifies directly into the specified database.

Example 1

The output of this INSERT EXPLAIN statement is referenced under the query plan name EmployeeSmithQuery in the TLE_Queries database.

```
INSERT EXPLAIN INTO TLE_Queries AS EmployeeSmithQuery
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

Example 2

The output of this INSERT EXPLAIN statement has a null query plan name in the TLE_Queries database.

```
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

Example 3

The output of this INSERT EXPLAIN statement is referenced under the query plan name Employee Smith Query in the QCD database.

```
INSERT EXPLAIN INTO QCD AS "Employee Smith Query"
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

Example 4

The output of this INSERT EXPLAIN statement is referenced under the query plan name Wizard Test in the Wizard_QCD database. Demographics are collected for table_1 and column statistics are collected for column_2 and column_3 in table_1. The frequency of performance of the specified SQL SELECT statement is 10 times per workload.

```
INSERT EXPLAIN WITH STATISTICS FOR table_1 INTO Wizard_QCD AS
"Wizard Test" FOR 10
SELECT *
FROM table_1
WHERE table_1.column_2 = 10
AND table_1.column_3 = 20
FOR 10;
```

Example 5

The output of this INSERT EXPLAIN statement is saved under the query plan name WizardTest2 in the Wizard_QCD database. Demographics are collected for the employee table and column statistics are collected for the emp_lastname column. No demographics are captured for the department table.

```
INSERT EXPLAIN WITH STATISTICS FOR employee INTO Wizard_QCD AS
WizardTest2
SELECT employee.first_name, department.dept_num
FROM employee, department
WHERE employee.emp_lastname = 'Smith'
AND department.dept_name IN ('Engineering', 'Support');
```

Example 6

This INSERT EXPLAIN statement returns an error because none of the tables specified in the table list are referenced in the specified query.

```
INSERT EXPLAIN WITH STATISTICS FOR employee1 INTO Wizard_QCD
AS WizardTest2
SELECT employee.first_name, department.dept_num
FROM employee, department
WHERE employee.emp_lastname = 'Smith'
AND department.dept_name IN ('Engineering', 'Support');

****Failure 5644: No match found for specified tables in the
request.
```

Related Statements

See "DUMP EXPLAIN" on page 3-40 for information on how to return Optimizer white tree information as a series of individual INSERT statements for the specified QCD rather than populating the database automatically.

Further Information

See the following topics for more information about index analysis:

- "COLLECT DEMOGRAPHICS" on page 3-11
- "COLLECT STATISTICS (QCD Form)" on page 3-27
- "DUMP EXPLAIN" on page 3-40
- "INITIATE INDEX ANALYSIS" on page 3-46
- "RESTART INDEX ANALYSIS" on page 3-62
- The chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2*.
- The chapter "Query Capture Facility" in *Teradata RDBMS SQL Reference, Volume 2*.
- *Teradata Index Wizard User Guide.*

RESTART INDEX ANALYSIS

Purpose

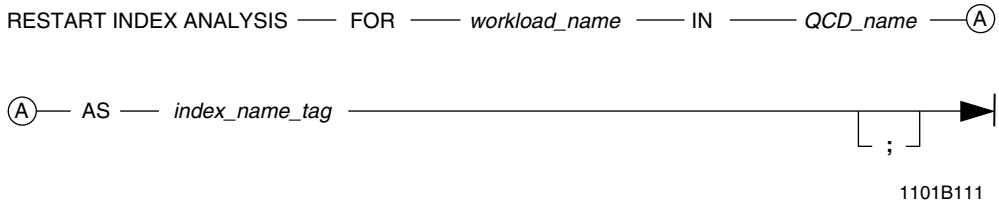
Restarts a previously halted index analysis started by an INITIATE INDEX ANALYSIS statement that specified a checkpoint.

Invocation

Executable.

Normally invoked via the Teradata Index Wizard utility.

Syntax



where:

Syntax element …	Specifies …
workload_name	the name of the workload on which index analysis is to be restarted.
QCD_name	the name of the query capture database in which *workload_name* is found.
index_name_tag	the name of the index analysis to be restarted. A row with this name must exist in the QCD AnalysisLog table or you cannot restart the index analysis.

ANSI Compliance

RESTART INDEX ANALYSIS is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have the INSERT privilege on the IndexRecommendations and IndexColumns tables in QCD or on database *QCD_name*.

You must also have INSERT and DELETE privileges on the AnalysisLog table.

The user who performs a RESTART INDEX ANALYSIS statement must be the same user who performed the checkpointed INITIATE INDEX ANALYSIS statement.

Rules for Performing RESTART INDEX ANALYSIS

The rules for RESTART INDEX ANALYSIS are the same as the rules for performing INITIATE INDEX ANALYSIS. See "Rules for Performing INITIATE INDEX ANALYSIS" on page 3-49 for further information.

RESTART INDEX ANALYSIS Requires a Checkpoint

If you do not specify a checkpoint clause in your INITIATE INDEX ANALYSIS statement, then you cannot restart a previously halted index analysis.

How RESTART INDEX ANALYSIS Works

RESTART INDEX ANALYSIS works exactly like INITIATE INDEX ANALYSIS. The only difference is that RESTART INDEX ANALYSIS begins its analysis with the first SQL statement in the workload that was not checkpoint logged in the AnalysisLog table.

If the restart aborts for any reason before completion, then the next RESTART INDEX ANALYSIS statement begins its analysis with the first SQL statement in the workload that was not checkpointed in the AnalysisLog table.

In all cases, the same boundary options and checkpoint trigger values as were specified for INITIATE INDEX ANALYSIS are used by default.

Once the restart completes successfully, its index recommendations are written to the IndexRecommendations table within the specified QCD.

Example 1

Assume that the query capture database named MyQCD exists on the system. The workload named MyWorkload in MyQCD consists of 100 SQL statements. The following SQL statement causes the index analysis information to be saved after every 10 SQL statements because of its CHECKPOINT clause specification:

```
INITIATE INDEX ANALYSIS ON tab1
FOR MyWorkload
IN MyQCD
AS table_1Index
CHECKPOINT 10;
```

Suppose a database restart occurs while processing the 55th SQL statement. Teradata then records the information pertaining to the first 50 SQL statements processed.

The following SQL statement causes the analysis to be done from the 51st SQL statement.

```
RESTART INDEX ANALYSIS
FROM MyWorkload
IN MyQCD
AS table1_Index;
```

Example 2

Again assume that the query capture database MyQCD exists on the system and that the workload named MyWorkload consists of 100 SQL statements.

The following SQL statement does not specify a checkpoint, so no incremental information is written to the AnalysisLog table in MyQCD.

```
INITIATE INDEX ANALYSIS ON tab1
FOR MyWorkload
IN MyQCD
AS table_1Index;
```

Suppose once again that a database restart occurs while processing the 55th SQL statement.

The following SQL statement returns an error because there is no row in AnalysisLog checkpointing the specified index analysis name, table_1Index.

```
RESTART INDEX ANALYSIS
FROM MyWorkload
IN MyQCD
AS table_1Index;
*** Failure 5669 No restart information found for specified
index analysis 'table_1Index'.
```

For More Information

See the following topics for more information about index analysis:

- "COLLECT DEMOGRAPHICS" on page 3-11.
- "COLLECT STATISTICS (QCD Form)" on page 3-27.
- "INITIATE INDEX ANALYSIS" on page 3-46.
- "INSERT EXPLAIN" on page 3-55.
- The chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2*.
- The chapter "Query Capture Facility" in *Teradata RDBMS SQL Reference, Volume 2*.
- *Teradata Index Wizard User Guide.*

Diagnostic Tools for Target Level Emulation and the Teradata Index Wizard

Introduction

This section documents statements used for various target level emulation applications.

See the chapter "Target Level Emulation" in *Teradata RDBMS SQL Reference, Volume 2* for more information about target level emulation.

Diagnostic Statements for Cost-Based Target Level Emulation

Target level emulation uses the following SQL diagnostic statements to extract, test, and obtain help about table information obtained from environmental cost parameters collected on a production system:

- "DIAGNOSTIC DUMP COSTS" on page 3-66
- "DIAGNOSTIC HELP COSTS" on page 3-68
- "DIAGNOSTIC SET COSTS" on page 3-71

Diagnostic Statements for Random-AMP Sample-Based Target Level Emulation

To work with table information obtained from random AMP sample statistics collected on a production system, see the descriptions of the following statements:

- "DIAGNOSTIC DUMP SAMPLES" on page 3-76
- "DIAGNOSTIC HELP SAMPLES" on page 3-78
- "DIAGNOSTIC SET SAMPLES" on page 3-80

Diagnostic Statement for Setting Index Validation Mode/Validating Indexes

The following statement sets the index validation mode for target level validation and then validates the secondary indexes proposed by the Teradata Index Wizard:

- "DIAGNOSTIC "Validate Index"" on page 3-83

DIAGNOSTIC DUMP COSTS

Purpose

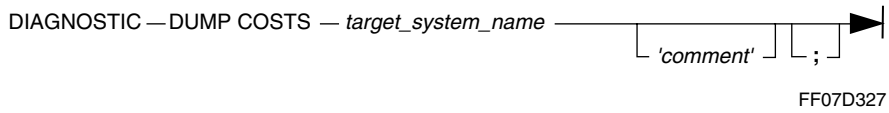
Extracts TPA-calculated optimizer cost information from a target system and writes it to SystemFE.Opt_Cost_Table on that system.

Invocation

Executable.

You cannot perform DIAGNOSTIC DUMP COSTS in ANSI transaction mode or within explicit transaction boundaries in Teradata transaction mode.

Syntax



where:

Syntax element …	Specifies …
target_system_name	a name used to designate the target (production) system from which optimizer cost information is calculated by the TPA. *target_system_name* is not case sensitive and can be no longer than 30 characters. *target_system_name* is used as the primary key for SystemFE.Opt_Cost_Table.
'comment'	an optional quoted literal description for the statement. The default character string is the specification for *target_system_name*. *comment* can be no longer than 255 bytes in length.

ANSI Compliance

DIAGNOSTIC DUMP COSTS is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have INSERT privileges on SystemFE.Opt_Cost_Table to dump costs.

TLE Need Not Be Enabled To Use DIAGNOSTIC DUMP COSTS

DIAGNOSTIC DUMP COSTS does not require Target Level Emulation to be enabled in order to extract target system environmental cost data.

Example

Dump the system environmental cost values for the optimizer on the current system to SystemFE.Opt_Cost_Table. For this example, you are using the name Production_1 for the current system.

Note that *target_system_name* is used as the primary key for the row in SystemFE.Opt_Cost_Table.

```
DIAGNOSTIC DUMP COSTS Production_1;
```

DIAGNOSTIC HELP COSTS

Purpose

For each scope level, returns the value the Optimizer is using for its cost estimator functions.

Invocation

Executable.

You cannot perform DIAGNOSTIC HELP COSTS in ANSI transaction mode or within explicit transaction boundaries in Teradata transaction mode.

Syntax



ANSI Compliance

DIAGNOSTIC HELP COSTS is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

TLE Need Not Be Enabled To Use DIAGNOSTIC HELP COSTS

DIAGNOSTIC HELP COSTS does not require Target Level Emulation to be enabled, but it cannot report scope level cost estimation data unless TLE is enabled. Instead, it reports an error message telling you that TLE is not enabled.

Example 1

No costs have been set.

```
DIAGNOSTIC HELP COSTS;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
Status of Cost Parameters
-------------------------------------------------
SESSION: IS IFP
IFP: IS SYSTEM
SYSTEM: IS TPA
```

Note: Any combination of costs can be set. The lowest level cost set has precedence over all higher levels, so IS IFP means the session level costs default to whatever IFP level costs have been set.

Example 2

Enable costs for target system Production_1 at SESSION level.

```
DIAGNOSTIC SET COSTS Production_1 ON FOR SESSION;

***Null statement accepted.
***Total elapsed time was 3 seconds.

DIAGNOSTIC HELP COSTS;
***Help information returned. 3 rows.
***Total elapsed time was 1 second.

Status of Cost Parameters
-------------------------------------------------
SESSION: Production_1
IFP: IS SYSTEM
SYSTEM: IS TPA
```

Example 3

You are unaware of the identity of the target system currently being emulated.

```
DIAGNOSTIC HELP COSTS;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
Status of Cost Parameters
-------------------------------------------------
SESSION: Production_1
IFP: JCP2
SYSTEM: TDAT1
```

By precedence, you can determine that the system being emulated is Production_1.

Example 4

The environmental cost set has been set for Production_1 at SYSTEM level as indicated.

```
DIAGNOSTIC SET COSTS Production_1 ON FOR SYSTEM;
DIAGNOSTIC HELP COSTS;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
Status of Cost Parameters
-------------------------------------------------
SESSION: IS IFP
IFP: IS SYSTEM
SYSTEM: Production_1
```

Example 5

The environmental cost set has been set for TPA at SYSTEM level as indicated.

```
DIAGNOSTIC SET COSTS TPA ON FOR SYSTEM;
DIAGNOSTIC HELP COSTS;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
Status of Cost Parameters
-------------------------------------------------
SESSION: IS IFP
IFP: IS SYSTEM
SYSTEM: IS TPA
```

Example 6

A series of environmental cost sets has been set as follows.

```
DIAGNOSTIC SET COSTS Production_1 ON FOR SESSION;
DIAGNOSTIC SET COSTS Production_2 ON FOR SESSION;
DIAGNOSTIC SET COSTS Production_3 ON FOR IFP;
DIAGNOSTIC SET COSTS Production_4 ON FOR SYSTEM;
DIAGNOSTIC HELP COSTS;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
Status of Cost Parameters
-------------------------------------------------
SESSION: Production_2
IFP: Production_3
SYSTEM: Production_4
```

DIAGNOSTIC SET COSTS

Purpose

Instructs the query optimizer on the test system to use the optimizer cost parameters from *target_system_name* for its emulations for a period defined by a scope-level keyword.

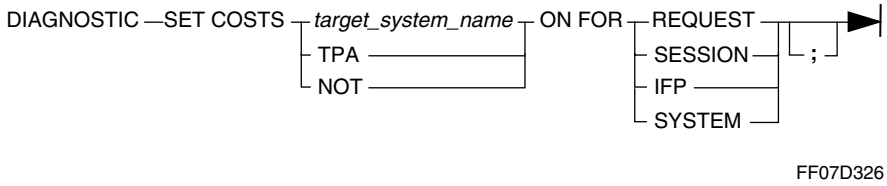
DIAGNOSTIC SET COSTS requires Target Level Emulation to be enabled in order to set scope level cost estimation data.

Invocation

Executable.

You cannot perform DIAGNOSTIC SET COSTS in ANSI transaction mode or within explicit transaction boundaries in Teradata transaction mode.

Syntax



FF07D326

where:

Syntax element …	Specifies …
target_system_name	the name of the target (production) system to be emulated. *target_system_name* is not case sensitive and can be no longer than 30 characters. *target_system_name* must be defined in SystemFE.Opt_Cost_Table for this statement to be valid.
TPA	to use the default system environmental cost parameters defined by the current Trusted Parallel Application environment. These values are calculated at system startup time for the current system. TPA and NOT have the same function.
NOT	
REQUEST	cost values for the target system are in effect only for the current multistatement request.

Syntax element …	Specifies …
SESSION	cost values for the target system are in effect for the duration of the current session until one of the following events occurs. • You log off the session • You restart the database • You change the costs in effect by performing another DIAGNOSTIC SET COSTS statement • You reset costs to the same values as those for IFP level by performing a DIAGNOSTIC SET COSTS NOT ON FOR SESSION statement
IFP	cost values for the target system are in effect on the current IFP for the session for all sessions until one of the following events occurs. • The next database restart • The costs are changed to a different target for IFP level • You reset costs to the same values as those for SYSTEM level by performing a `DIAGNOSTIC SET COSTS NOT ON FOR IFP` statement
SYSTEM	cost values for the target system are in effect for all sessions across all restarts and reboots until one of the following events occurs. • The costs are changed to a different target • The costs are set to TPA level by performing a `DIAGNOSTIC SET COSTS NOT ON FOR SYSTEM` or `DIAGNOSTIC SET COSTS TPA ON FOR SYSTEM` statement, both of which reset costs to TPA level

ANSI Compliance

DIAGNOSTIC SET COSTS is a Teradata extension to the ANSI SQL-99 standard.

Authorization

Access to SystemFE.Opt_Cost_Table is restricted to users with the appropriate database and table-level privileges.

The following scope levels can only be set by user DBC or user SystemFE only.

- IFP
- SYSTEM

All scope levels require SELECT privileges on SystemFE.Opt_Cost_Table.

Note that SystemFE.Opt_Cost_Table must exist for any of these scope levels to be defined.

Scope Level Precedence

Scope levels have a sequential precedence. The following table lists the precedence ratings for the various scope levels.

Precedence	Option
1	REQUEST
2	SESSION
3	IFP
4	SYSTEM

The following table indicates how precedences cascade.

For this scope definition ...	The scope for the costs definition defaults to this level ...
NOT ON FOR REQUEST	SESSION
NOT ON FOR SESSION	IFP
NOT ON FOR IFP	SYSTEM

DIAGNOSTIC SET COSTS Performance Precedence in Multistatement Requests

Each DIAGNOSTIC SET COSTS statement in a multistatement request is performed in the order of its appearance, but prior to the other statements in the same multistatement request. Therefore, all DIAGNOSTIC SET COSTS statements in a multistatement are in effect before the other statements are processed.

Example 1

Disable SESSION costs. The Optimizer reverts to the environmental cost value set defined by the next higher level in the hierarchy. In this case, that level is IFP. The costs parameters used for the emulation are those for *target_system_name* JCP2.

```
DIAGNOSTIC SET COSTS NOT ON FOR SESSION;

DIAGNOSTIC HELP COSTS;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
Status of Cost Parameters
--------------------------------------------------
SESSION: IS IFP
IFP: JCP2
SYSTEM: TDAT1
```

Example 2

Enable the environmental cost value set defined at SESSION level for *target_system_name* Production_1.

```
DIAGNOSTIC SET COSTS Production_1 ON FOR SESSION;

DIAGNOSTIC HELP COSTS;
***Help information returned. 3 rows.
***Total elapsed time was 1 second.

Status of Cost Parameters
-------------------------------------------------
SESSION: Production_1
IFP: IS SYSTEM
SYSTEM: IS TPA
```

Example 3

Enable the environmental cost value set defined at REQUEST level for *target_system_name* Production_2 and EXPLAIN the SELECT that follows.

```
DIAGNOSTIC SET COSTS Production_2 ON FOR REQUEST
;EXPLAIN SELECT * FROM order_table;
```

After the EXPLAIN SELECT statement completes, the optimizer cost values used revert to what they had been before being reset temporarily to REQUEST.

Example 4

Enable the environmental cost value set from *target_system_name* Production_3 as defined for the current IFP and EXPLAIN the SELECT that follows.

```
DIAGNOSTIC SET COSTS Production_3 ON FOR IFP
 ;EXPLAIN SELECT * FROM order_table;
```

Example 5

Set the environmental cost value for the current IFP to be the same as the next highest level in the scope precedence. In this case, that level is SYSTEM.

```
DIAGNOSTIC SET COSTS NOT ON FOR IFP;
```

Example 6

Create (or write over) the optimizer cost GDO with the cost values associated with the system name Production_1 in SystemFE.Opt_Cost_Table and use the optimizer cost GDO for the entire system.

```
DIAGNOSTIC SET COSTS Production_1 ON FOR SYSTEM;
```

Example 7

Enable the environmental cost value set defined by TPA startup for the current SESSION.

```
DIAGNOSTIC SET COSTS TPA ON FOR SESSION;
```

Example 8

Enable the environmental cost value set defined by TPA startup for the current SYSTEM.

```
DIAGNOSTIC SET COSTS TPA ON FOR SYSTEM;
```

Example 9

Reset the environmental cost value set to the values defined by TPA startup for the current SYSTEM.

The semantics of this statement are identical with the semantics of the statement in “Example 8.”

```
DIAGNOSTIC SET COSTS NOT ON FOR SYSTEM;
```

DIAGNOSTIC DUMP SAMPLES

Purpose

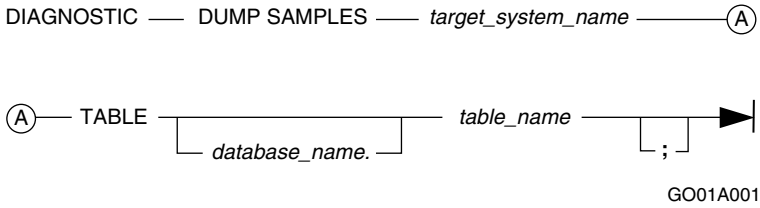
Extracts random AMP samples from a target system and writes it to SystemFE.Opt_RAS_Table on that system.

Invocation

Executable.

You cannot perform DIAGNOSTIC DUMP SAMPLES in ANSI transaction mode or within explicit transaction boundaries in Teradata transaction mode.

Syntax



where:

Syntax element …	Specifies …
target_system_name	a name used to designate the target (production) system from which random AMP sample information is to be estimated. *target_system_name* is not case sensitive and can be no longer than 30 characters. If the name contains pad characters or nonstandard characters, it must be demarcated by double quotation mark characters. The primary key for the sample row in the table SystemFE.Opt_RAS_Table is *target_system_name* + *database_name* + *table_name*.
database_name	the name of the database in which *table_name* is defined. If the name contains pad characters or non-standard characters, it must be demarcated by double quotation mark characters.
table_name	the name of the table for which random AMP sample statistics emulation is to be performed. If the name contains pad characters or non-standard characters, it must be demarcated by double quotation mark characters.

ANSI Compliance

DIAGNOSTIC DUMP SAMPLES is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have INSERT privileges on SystemFE.Opt_RAS_Table to dump random AMP samples.

TLE Need Not Be Enabled To Use DIAGNOSTIC DUMP SAMPLES

DIAGNOSTIC DUMP SAMPLES does not require Target Level Emulation to be enabled in order to extract target system random AMP samples.

Example

Dump the random AMP statistical samples on the current system for the Payroll table in the Employee database to SystemFE.Opt_RAS_Table. For this example, you are using the name Production_1 for the current system.

```
DIAGNOSTIC DUMP SAMPLES Production_1 TABLE Employee.Payroll;
```

DIAGNOSTIC HELP SAMPLES

Purpose

For each scope level, returns the random AMP statistical samples that have been set.

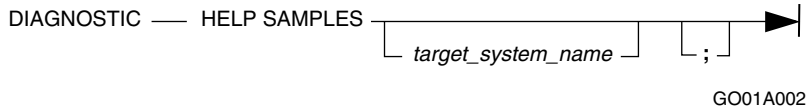
DIAGNOSTIC HELP SAMPLES does not require Target Level Emulation to be enabled, but it cannot report scope level statistics unless TLE is enabled. Instead, it reports an error message telling you that TLE is not enabled.

Invocation

Executable.

You cannot perform DIAGNOSTIC HELP SAMPLES in ANSI transaction mode or within explicit transaction boundaries in Teradata transaction mode.

Syntax



where:

Syntax element …	Specifies …
target_system_name	the name of the target system for which information is to be reported. If no target system is named, then information is reported for all the samples set in both SESSION and SYSTEM modes. If the name contains pad characters or non-standard characters, it must be demarcated by double quotation mark characters.

ANSI Compliance

DIAGNOSTIC HELP SAMPLES is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

TLE Need Not Be Enabled To Use DIAGNOSTIC HELP SAMPLES

DIAGNOSTIC HELP SAMPLES does not require Target Level Emulation to be enabled, but it cannot report scope level random AMP statistical samples unless TLE is enabled. Instead, it reports an error message telling you that TLE is not enabled.

Example 1

This statement displays all the samples set in all scope levels.

```
DIAGNOSTIC HELP SAMPLES;
```

Example 2

This statement displays all the samples set in all scope levels for the Test_1 system. The scope level for Test_1 is set to Session.

```
DIAGNOSTIC HELP SAMPLES Test_1;
*** Help information returned. One row.
*** Total elapsed time was 1 second.

SystemName TEST_1   DatabaseName EMPLOYEE
TableName  PAYROLL  Level SESSION
```

DIAGNOSTIC SET SAMPLES

Purpose

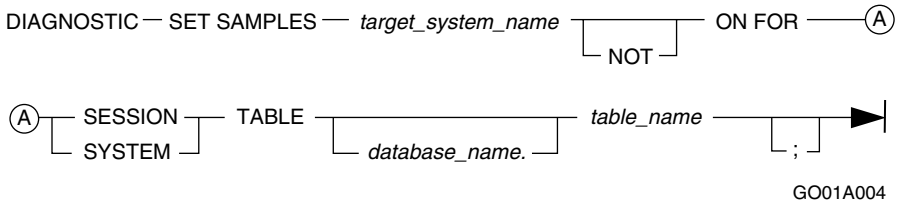
Instructs the query optimizer on the test system to use the random AMP samples from *target_system_name* for its emulations.

Invocation

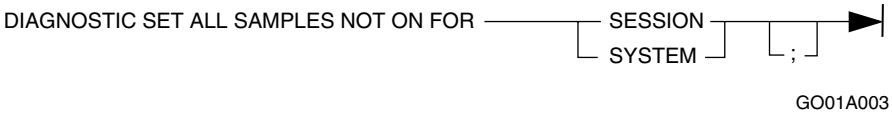
Executable.

You cannot perform DIAGNOSTIC SET SAMPLES in ANSI transaction mode or within explicit transaction boundaries in Teradata transaction mode.

General Syntax



Disable All Samples Syntax



where:

Syntax element …	Specifies …
ALL	that all random AMP statistical samples for the specified scope level are to be disabled.
target_system_name	the name of the target (production) system to be emulated. *target_system_name* is not case sensitive and can be no longer than 30 characters. If the name contains pad characters or non-standard characters, it must be demarcated by double quotation mark characters. *target_system_name* must be defined in SystemFE.Opt_RAS_Table before you can perform this statement.
NOT	to disable sample selection.

Syntax element …	Specifies …
SESSION	that sample values for the target system are in effect for the duration of the current session until one of the following events occurs: • You log off the session • You restart the database • You change the samples in effect by performing the following statement: `DIAGNOSTIC SET SAMPLES` *target_system_name* `NOT ON FOR SESSION TABLE` *table_name*
SYSTEM	that sample values for the target system are in effect for all sessions across all restarts and reboots until you perform the following statement: `DIAGNOSTIC SET SAMPLES` *target_system_name* `NOT ON FOR SYSTEM TABLE` *table_name*
database_name	the name of the database in which *table_name* is defined. If the name contains pad characters or non-standard characters, it must be demarcated by double quotation mark characters.
table_name	the name of the table for which random AMP sample statistics emulation is to be performed. If the name contains pad characters or non-standard characters, it must be demarcated by double quotation mark characters.

ANSI Compliance

DIAGNOSTIC SET SAMPLES is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have SELECT privileges on SystemFE.Opt_RAS_Table to set samples on or off.

Scope Level Precedence

Scope levels have a sequential precedence. The following table lists the precedence ratings for the various scope levels for DIAGNOSTIC SET SAMPLES.

Precedence	Option
1	SESSION
2	SYSTEM

Limits on Sample Emulation

Random AMP statistical samples can be emulated for a maximum of 128 tables at a time per scope level.

Examples

Note that these examples require that a corresponding row must exist in the SystemFE.Opt_RAS_Table with the following column values:

Column	Value
System Name	Test_1
Database Name	Employee
Table Name	Payroll

Example 1

Enable the imported random AMP statistical samples for the Payroll table in database Employee on test machine Test_1.

```
DIAGNOSTIC SET SAMPLES Test_1 ON FOR SESSION TABLE
Employee.Payroll;
```

Example 2

Disable the imported random AMP statistical samples for the Payroll table in database Employee on test machine Test_1.

```
DIAGNOSTIC SET SAMPLES Test_1 NOT ON FOR SESSION TABLE
Employee.Payroll;
```

Example 3

Disable all random AMP statistical samples for all tables enabled at the SYSTEM scope level.

```
DIAGNOSTIC SET ALL SAMPLES NOT ON FOR SYSTEM;
```

Example 4

Disable all random AMP statistical samples for all tables enabled at the SESSION scope level.

```
DIAGNOSTIC SET ALL SAMPLES NOT ON FOR SESSION;
```

DIAGNOSTIC "Validate Index"

Purpose

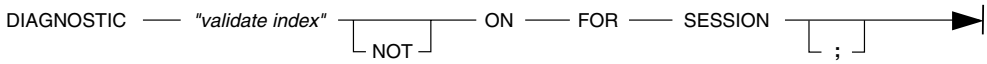
Sets the index validation mode for the session and validates the indexes proposed by the Teradata Index Wizard utility.

Invocation

Executable.

Normally invoked via the Teradata Index Wizard utility.

Syntax



TW01A002

where:

Syntax element ...	Specifies that index validation is ...
NOT ON	disabled for the session.
ON	enabled for the session.
FOR SESSION	set at the session level. No other diagnostic levels are valid.

ANSI Compliance

DIAGNOSTIC "validate index" is a Teradata extension to the ANSI SQL-99 specification.

Authorization

None.

Index Validation Workflow

The index recommendations produced by the Teradata Index Wizard are a set of SQL DDL statements drawn from the following list:

- CREATE INDEX
- DROP INDEX
- COLLECT STATISTICS (Optimizer Form)
- DROP STATISTICS (Optimizer Form)

These SQL statements must be submitted following the DIAGNOSTIC SQL command, as illustrated by the "Example" on page 3-84.

In index validation mode, the Parser creates pseudo indexes corresponding to the CREATE INDEX statements in the workload cache and collects the statistics for the COLLECT STATISTICS statement into the QCD tables.

If a DROP INDEX statement is submitted, then the corresponding index definition is considered dropped until the index validation mode is reset. The COLLECT STATISTICS and DROP STATISTICS statements must be submitted in their QCD forms (see "COLLECT STATISTICS (QCD Form)" and "DROP STATISTICS (QCD Form)" in *Teradata RDBMS SQL Reference, Volume 4*) during the index validation mode else an error is reported. Note that the data dictionary is neither updated with the index definition nor the collected statistics. Instead, the statistics are written into the QCD. Because these statements do not write definitions into the data dictionary, they are not treated as DDL when submitted for validation.

After performing the pseudo-DDL statement recommendations with DIAGNOSTIC "Validate Index," you must submit the workload SQL statements using the INSERT EXPLAIN statement (see "INSERT EXPLAIN" in *Teradata RDBMS SQL Reference, Volume 6*). The existing pseudo indexes, along with their corresponding statistics, including those collected with the COLLECT STATISTICS (Optimizer Form) statement, are used to generate the query plan. The indexes on which DROP INDEX is submitted are ignored during plan generation.

Because the recommendations are made for the workload as a whole, you should submit the recommendation statements before you submit the SQL statements with the INSERT EXPLAIN modifier. You can then verify the performance improvement with the captured query plan. After performing the workload SQL statements, you must reset the index validation mode for the session.

Example

The following example assumes that the index analysis phase of the Teradata Index Wizard has proposed the following recommendations for the lineitem table.

```
CREATE INDEX lx_pk (l_partkey) ON lineitem;
COLLECT STATISTICS FOR SAMPLE 50 INTO Wizard_QCD ON lineitem
INDEX lx_pk;
```

These DDL statements are saved in IndexRecommendations table in the specified QCD.

You then submit the following sequence of SQL statements in the order indicated to validate the recommendation:

```
DIAGNOSTIC "Validate Index" ON FOR SESSION;

CREATE INDEX lx_pk (l_partkey) ON lineitem; /* Does not update
the data dictionary */

COLLECT STATISTICS FOR SAMPLE 50 INTO Wizard_QCD ON lineitem
INDEX lx_pk; /* Does not update the data dictionary */
…
/* Perform workload SQL statements with INSERT EXPLAIN modifier */

DIAGNOSTIC "Validate Index" NOT ON FOR SESSION;
```

This procedure can also be used to perform what-if analyses on the statements.

The SQL interface allows you to simulate indexes and optionally use sampled statistics in order to gauge the behavior of SQL statements with the proposed indexes. Note that the index statistics used for this analysis are fetched from the QCD, not the data dictionary.

If the resulting plans show improvements, then you can create and drop the submitted indexes using the CREATE INDEX and DROP INDEX statements (see "CREATE INDEX" on page 1-100 and "DROP INDEX" on page 1-333).

You can collect statistics on those newly defined indexes using the COLLECT STATISTICS (Optimizer Form) and DROP STATISTICS (Optimizer Form) statements (see "COLLECT STATISTICS (Optimizer Form)" on page 3-14 and "DROP STATISTICS (Optimizer Form)" on page 3-31 for information about using these statements).

Further Information

See the chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2* for more information about the Teradata Index Wizard.

See *Teradata Index Wizard User Guide* for information about how to use the client utility.

Chapter 4:

SQL Help and Database Object Definition Tools: HELP and SHOW

Introduction

Teradata SQL provides several powerful tools for system administrators, database and security administrators, and application programmers.

These tools fall into three categories.

- Help about database object definitions:
  - "HELP COLUMN" on page 4-7
  - "HELP CONSTRAINT" on page 4-22
  - "HELP DATABASE HELP USER" on page 4-26
  - "HELP HASH INDEX" on page 4-29
  - "HELP INDEX" on page 4-31
  - "HELP JOIN INDEX" on page 4-38
  - "HELP MACRO HELP TABLE HELP VIEW" on page 4-40
  - "HELP PROCEDURE" on page 4-46
  - "HELP SESSION" on page 4-52
  - "HELP STATISTICS (Optimizer Form)" on page 4-60
  - "HELP STATISTICS (QCD Form)" on page 4-63
  - "HELP TRIGGER" on page 4-65
  - "HELP VOLATILE TABLE" on page 4-68
- Help about SQL statement and utility command syntax:
  - "HELP (Online Form)" on page 4-70
- Summaries of database object definition statement text:
  - "SHOW" on page 4-73
  - "SHOW HASH INDEX SHOW JOIN INDEX SHOW MACRO SHOW PROCEDURE SHOW TABLE SHOW TRIGGER SHOW VIEW" on page 4-79

Information about how to optimize the performance of your database from a database design perspective can be found in the following manuals.

- *Teradata RDBMS Database Design*
- *Teradata RDBMS Performance Optimization*

Experienced SQL users can also see simplified HELP and SHOW statement descriptions in *Teradata RDBMS SQL/Data Dictionary Quick Reference*.

HELP Statements

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

HELP (Online)

The HELP online statements return information about the syntax for SQL statements.

SHOW Statements

The various 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* 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, or macros for purposes of testing new software.

HELP

Purpose

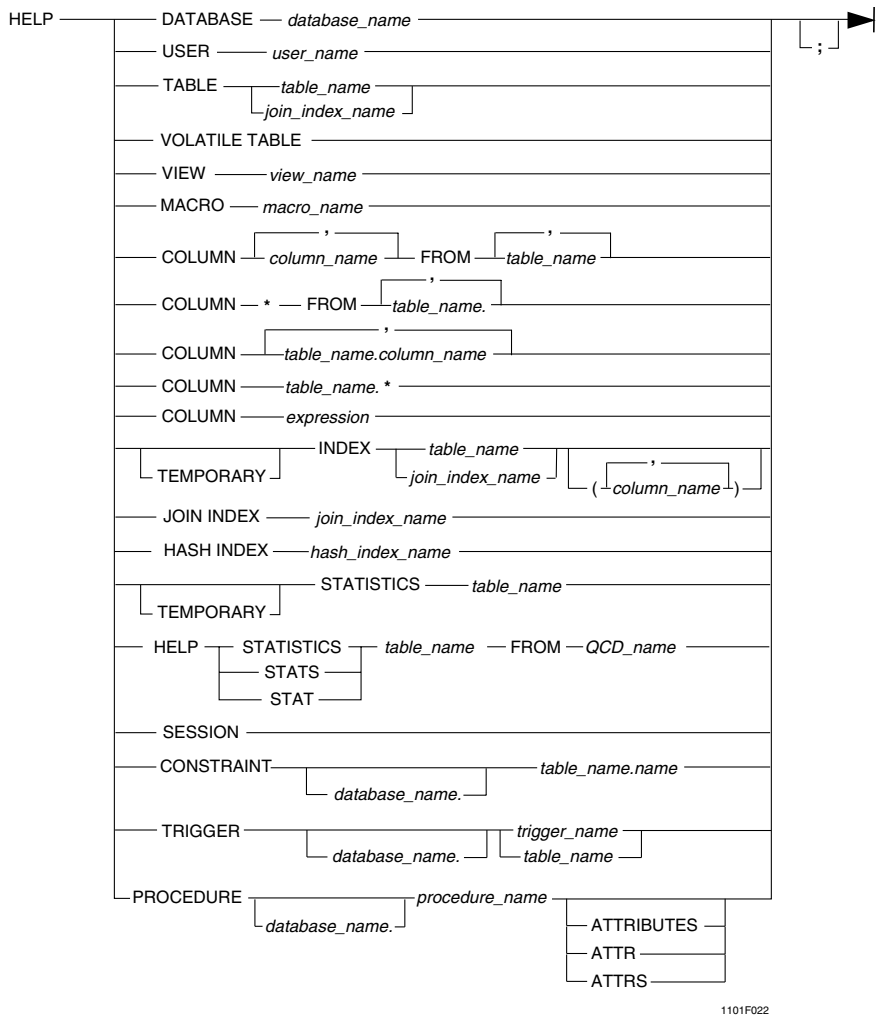
Reports information from the data dictionary, including data about the following items:

- The specified database, table, view, macro, stored procedure, constraint, trigger, hash index, join index, or secondary index.
- All or specified columns of a table or view.
- Any statistics collected for the specified table.
- The attributes of the current session.
- Attribute and format details for each parameter of a stored procedure.

Invocation

Executable.

Syntax



where:

Syntax Element…	Specifies …
database_name	the name of the database for which help is needed.
user_name	the name of the user for whom help is needed.
table_name	the name of the table for which help is needed.
join_index_name	the name of the join index for which help is needed.
volatile_table_name	the name of the volatile table for which help is needed.
view_name	the name of the view for which help is needed.
macro_name	the name of the macro for which help is needed.
column_name	the name of the column for which help is needed.
*	that help is needed for all columns in a table.
expression	an expression for which the data type is required.
TEMPORARY	that the index for which help is needed is defined on a global temporary table.
index_name	the name of the index, if it has been assigned a name.
QCD_name	the name of the QCD database in which the TableStatistics table from which statistics are to be retrieved resides.
SESSION	that help is needed for the current session.
table_name.name	a fully qualified constraint name.
trigger_name	trigger for which help is needed.
procedure_name	stored procedure for which help is needed.
ATTRIBUTES	a keyword indicating that the stored procedure creation-time attributes from the column SPObjectCodeRows in DBC.TVM are to be displayed rather than the attribute and format information for the procedure parameters.

ANSI Compliance

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

Authorization

Except for HELP USER DBC, the user requesting HELP information must either own the specified object or have at least one privilege on that object (HELP USER DBC does not require any rights on user DBC).

Usage Notes

There is no check of object type: HELP USER and HELP DATABASE have the same effect.

The HELP statement allows users to review the attributes of an object or a session without having to construct and execute a query on a Data Dictionary view.

The actual output of HELP results are too wide to be shown on a single screen and are sometimes wider than 132 characters. Interactive users can format returns from HELP with BTEQ formatting commands such as [.]SET SIDETITLES and [.]SET FOLDLINE.

Various examples of HELP queries are shown following. The contents of HELP returns are explained in the following subsections.

For Related Information

See "HELP (Online Form)" on page 4-70 for information about obtaining online help for SQL statements and client utility syntax.

Formatting HELP Results

HELP results may be formatted. For example, this BTEQ input returns session information as demonstrated by the following report.

```
.SIDETITLES
.FOLDLINE
HELP SESSION;
```

```
                     User Name  SPEC1
                  Account Name  DBC
                    Logon Date  02/05/26
                    Logon Time  09:47:09
               Current DataBase  SPEC1
                     Collation  MULTINATIONAL
                  Character Set  SWED_EBCDIC
          Transaction Semantics  ANSI
               Current DateForm  ANSIDate
            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  YYYY/MM/DD
                Radix Separator  ,
                Group Separator  .
                  Grouping Rule  6
      Currency Radix Separator  ,
      Currency Group Separator  .
        Currency Grouping Rule  6
                  Currency Name  Sweden Kronor
                       Currency  $
                    ISOCurrency  SEK
           Dual Currency Name  Sweden Kronor
                  Dual Currency  $
                Dual ISOCurrency  SEK
        Default ByteInt format  -(3)9
        Default Integer format  G-(I)9
       Default SmallInt format  G-(I)9
        Default Numeric format  G--(I)D9(F)
           Default Real format  G-9D99999999999999E-999
           Default Time format  HH:MI:SSDS(F)Z
      Default Timestamp format  YYYY-MM-DDBHH:MI:SSDS(F)Z
                  Current Role  Finance
                  Logon Account  DBC
```

HELP COLUMN

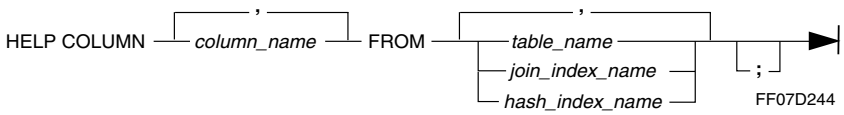
Purpose

Displays the attributes of a column, including whether it is a single-column primary or secondary index and, if so, whether it is unique.

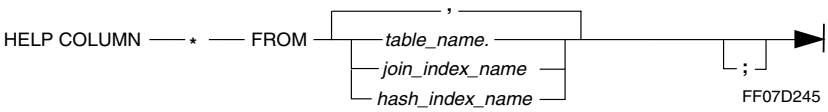
Invocation

Executable.

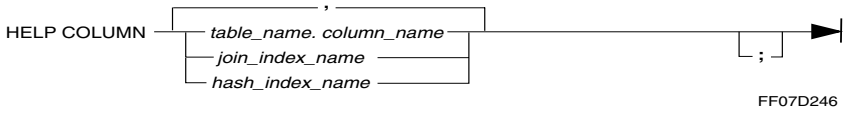
Syntax 1



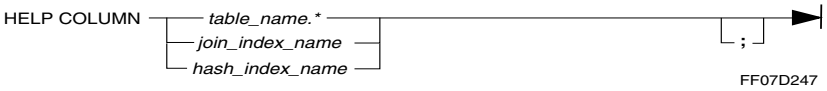
Syntax 2



Syntax 3



Syntax 4



Syntax 5



where:

Syntax Element...	Specifies...
column_name	the name of the column for which attribute information is required.
table_name	the name of the table for which attribute information is required.

Syntax Element...	Specifies...
join_index_name	the name of the join index for which attribute information is required.
hash_ index_name	the name of the hash index for which attribute information is required.
*	that attribute information for all the columns in the named table is requested.
table_name.column_name	the name of the table-qualified column for which attribute information is required.
expression	an expression for which the data type is required.

ANSI Compliance

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

Authorization

You must either own the table, join index, or hash index in which the column is defined or have at least one privilege on that table.

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”).

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 HASH INDEX SHOW JOIN INDEX SHOW MACRO SHOW PROCEDURE SHOW TABLE SHOW TRIGGER SHOW VIEW” on page 4-79) or the system view DBC.IndexConstraints (see *Teradata RDBMS Data Dictionary* for further information) to obtain the partitioning information.

Specific index information is reported as indicated by the following table:

<table>
<tr><th>Index-Related Attribute</th><th>Valid Values</th></tr>
<tr><td>Indexed?</td><td>Does the column also define a single-column index?

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.

Use HELP INDEX to see a list of all the primary and secondary indexes for a table (see “HELP INDEX” on page 4-31).

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>N</td><td>The column does not define a single-column index.

If the column is a component of a composite index, but does not define a single-column index, then the code reported is N[a].</td></tr>
<tr><td>Y</td><td>The column defines a single-column index[b].</td></tr>
</table>
</td></tr>
<tr><td>Unique?</td><td>If the column defines a single-column index, is that index unique or nonunique?

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>null</td><td>The column does not define a single-column index.</td></tr>
<tr><td>N</td><td>The index is not unique.</td></tr>
<tr><td>Y</td><td>The index is unique.</td></tr>
</table>
</td></tr>
<tr><td>Primary?</td><td>If the column defines a single-column index, is that index a primary index or a secondary index?

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>null</td><td>The column does not define a single-column index.</td></tr>
<tr><td>P</td><td>The column defines a primary index.</td></tr>
<tr><td>S</td><td>The column defines a secondary index.</td></tr>
</table>
</td></tr>
</table>

a. A column can be a member of more than one index.
b. You can define multiple indexes on the same columns as long as they are different in some respect (value-ordered vs. hash-ordered or nonunique primary index on some set of columns and a USI on the same columns).

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?

HELP COLUMN returns the information indicated in "HELP COLUMN Information and Attributes" on page 4-18.

Examples

The following pages illustrate several examples of using HELP COLUMN.

Example 1

The following shows a sample output display for HELP COLUMN on the table defined by the preceding CREATE TABLE statement.

```
CREATE TABLE table_1
 (field_1 INTEGER NOT NULL PRIMARY KEY,
  field_2 INTEGER CHECK (field_2>0) CHECK (field_2<100));

HELP COLUMN table_1.field_2;

             Column Name   f2
                    Type   I
                Nullable   Y
                  Format   -(10)9
              Max Length   4
     Decimal Total Digits   ?
Decimal Fractional Digits   ?
              Range Low     ?
             Range High     ?
              Uppercase     N
             Table/View?    T
                Indexed?    N
                 Unique?    N
                Primary?    N
                   Title    ?
        Column Constraint   CHECK (("f2">0) AND
              Char Type     ?
             IdCol Type     ?
```

In BTEQ, a ? returned in the displayed listing above indicates a null.

Example 2

The following two statements return information about the attributes of the named column(s) in the named table or view.

```
HELP COLUMN column_name
FROM table_name;

HELP COLUMN table_name.column_name;
```

Example 3

The following two statements return attribute information for all the columns of the named table or view. All column attributes are returned except any default values and comments (for these, use HELP TABLE table_name).

```
HELP COLUMN * FROM table_name;

HELP COLUMN table_name.*;
```

Example 4

The following statement requests information about particular columns in the Employee and Department tables (note that each column reference is fully qualified with its table name):

```
HELP COLUMN Employee.Name, Employee.DeptNo, Department.DeptNo
FROM Employee, Department;
```

The return from the preceding query lists each column in the order in which it was specified; therefore, the first DeptNo column listed is for the Employee table, the second is for the Department table:

```
Column Name   Type    Nullable
-----------   ----    --------
Name           CV       N
DeptNo         I2       Y
DeptNo         I2       N
```

Example 5

Consider a KanjiEBCDIC user who creates a table using all five character data types. The table definition is as follows.

```
CREATE TABLE kanji_user.table_1,
 NO FALLBACK,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL
 (
  clatin CHARACTER(5) CHARACTER SET LATIN,
  ckanji1 CHARACTER(5) CHARACTER SET KANJI1,
  cgraphic CHARACTER(5) CHARACTER SET GRAPHIC,
  csjis CHARACTER(5) CHARACTER SET KANJISJIS
  cunicode CHARACTER(5) CHARACTER SET UNICODE)
PRIMARY INDEX (clatin);
```

You submit the following HELP COLUMN statement using BTEQ with the .SIDETITLES and .FOLDLINE formatting options specified.

```
HELP COLUMN table_1.*;
```

The output, abbreviated to illustrate some relevant columns only, looks like the following.

```
             Column Name  clatin
                    Type  CF
                  Format  X(5)
              Max Length  5
                   Title  ?
       Column Constraint  ?
               Char Type  1
              IdCol Type  ?
             Column Name  ckanji1
                    Type  CF
                  Format  X(5)
              Max Length  5
                   Title  ?
       Column Constraint  ?
               Char Type  5
              IdCol Type  ?
             Column Name  cgraphic
                  Format  X(5)
                    Type  CF
              Max Length  10
                   Title  ?
       Column Constraint  ?
               Char Type  4
```

```
             IdCol Type  ?
           Column Name   csjis
                Format   X(5)
                  Type   CF
            Max Length   5
                 Title   ?
       Column Constraint ?
              Char Type  3
             IdCol Type  ?
           Column Name   cunicode
                Format   X(5)
                  Type   CF
            Max Length   10
                 Title   ?
       Column Constraint ?
              Char Type  2
             IdCol Type  ?
```

Example 6: Partitioned Primary Index Table

This example uses the following table definition:

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

HELP COLUMN generates the following report on this table. The BTEQ commands are included to show how the report format was derived. Attributes of particular interest for this example are highlighted in boldface.

```
 BTEQ -- Enter your DBC/SQL request or BTEQ command:
 .sidetitles
 .sidetitles
  BTEQ -- Enter your DBC/SQL request or BTEQ command:
 .foldline
 .foldline
  BTEQ -- Enter your DBC/SQL request or BTEQ command:
 help column * from orders;
 help column * from orders;

  *** Help information returned. 9 rows.
  *** Total elapsed time was 1 second.

                 Column Name o_orderkey
                        Type I
                    Nullable N
                      Format -(10)9
                  Max Length 4
      Decimal Total Digits ?
 Decimal Fractional Digits ?
                  Range Low ?
                 Range High ?
                   UpperCase N
                 Table/View? T
                    Indexed? Y
                     Unique? N
                    Primary? P
                       Title ?
          Column Constraint ?
                  Char Type ?
                 IdCol Type ?
                 Column Name o_custkey
                        Type I
                    Nullable Y
                      Format -(10)9
                  Max Length 4
      Decimal Total Digits ?
 Decimal Fractional Digits ?
                  Range Low ?
                 Range High ?
                   UpperCase N
                 Table/View? T
                    Indexed? Y
```

```
                  Unique? N
                 Primary? S
                    Title ?
       Column Constraint ?
                Char Type ?
               IdCol Type ?
              Column Name o_orderstatus
                    Type CF
                Nullable Y
                  Format X(1)
              Max Length 1
    Decimal Total Digits ?
Decimal Fractional Digits ?
              Range Low ?
             Range High ?
              UpperCase C
            Table/View? T
               Indexed? Na
                Unique? ?
               Primary? ?
                  Title ?
       Column Constraint ?
                Char Type 1
               IdCol Type ?
              Column Name o_totalprice
                    Type D
                Nullable N
                  Format ------------.99
              Max Length 8
    Decimal Total Digits 13
Decimal Fractional Digits 2
              Range Low ?
             Range High ?
              UpperCase N
            Table/View? T
               Indexed? Nb
                Unique? ?
               Primary? ?
                  Title ?
       Column Constraint ?
                Char Type ?
               IdCol Type ?
              Column Name o_orderdate
                    Type DA
                Nullable N
```

```
                    Format yyyy-mm-dd
                Max Length 4
     Decimal Total Digits ?
Decimal Fractional Digits ?
                 Range Low ?
                Range High ?
                 UpperCase N
               Table/View? T
                  Indexed? Y
                   Unique? N
                  Primary? S
                     Title ?
         Column Constraint ?
                 Char Type ?
                IdCol Type ?
               Column Name o_orderpriority
                      Type CF
                  Nullable Y
                    Format X(21)
                Max Length 21
     Decimal Total Digits ?
Decimal Fractional Digits ?
                 Range Low ?
                Range High ?
                 UpperCase N
               Table/View? T
                  Indexed? N
                   Unique? ?
                  Primary? ?
                     Title ?
         Column Constraint ?
                 Char Type 1
                IdCol Type ?
               Column Name o_clerk
                      Type CF
                  Nullable Y
                    Format X(16)
                Max Length 16
     Decimal Total Digits ?
Decimal Fractional Digits ?
                 Range Low ?
                Range High ?
                 UpperCase N
               Table/View? T
                  Indexed? N
```

```
                  Unique? ?
                 Primary? ?
                    Title ?
       Column Constraint ?
               Char Type 1
              IdCol Type ?
             Column Name o_shippriority
                    Type I
                 Nullable Y
                   Format -(10)9
              Max Length 4
    Decimal Total Digits ?
Decimal Fractional Digits ?
              Range Low ?
             Range High ?
              UpperCase N
            Table/View? T
               Indexed? N
                Unique? ?
               Primary? ?
                  Title ?
       Column Constraint ?
               Char Type ?
              IdCol Type ?
             Column Name o_comment
                    Type CV
                 Nullable Y
                   Format X(79)
              Max Length 79
    Decimal Total Digits ?
Decimal Fractional Digits ?
              Range Low ?
             Range High ?
              UpperCase N
            Table/View? T
               Indexed? N
                Unique? ?
               Primary? ?
                  Title ?
       Column Constraint ?
               Char Type 1
              IdCol Type ?
```

a. o_orderstatus *is* a component of a composite secondary index.
b. o_totalprice is not a component of any index.

HELP COLUMN Information and Attributes

All fields are subject to formatting using the rules described for their Export Width specification in Field Mode.

The affected fields are the following:

- Column Name
- Type
- Nullable
- Format
- Uppercase
- Table/View
- Indexed
- Unique
- Primary
- Title
- Column Constraint

Column Attributes

Field		Data Type	Nullable?	Description
COLUMN NAME		CHAR(30)	no	None.
TYPE		CHAR(2)	yes	Data type for column data. With an explicit format, DATE is imported and exported according to that format. With an implicit format created in INTEGERDATE mode, the export format is 'YY/MM/DD'. With an implicit format created in ANSIDATE mode, the export format is 'YYYY-MM-DD'. MaxLength is 4, representing the internal storage size. When DATEFORM=ANSIDATE, this field should be described externally as CHAR(10). Host data characteristics for these data types should be interpreted as the equivalent of CF, or CHAR(n) data type. MaxLength is the length of the *external* CharFix representation of the field, defined as CHAR(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 seems more appropriate for these data types.
Code	**Description**			
I1	BYTEINT			
I2	SMALLINT			
I	INTEGER			
D	DECIMAL or NUMERIC			
F	FLOAT or REAL or DOUBLE PRECISION			
DA	DATE			
AT	TIME			
TS	TIMESTAMP			
TZ	TIME WITH TIME ZONE			

Field		Data Type	Nullable?	Description
TYPE (continued)		CHAR(2)	yes	See "External Representation Examples" in *Teradata RDBMS SQL Reference, Volume 3* for each individual DateTime and Interval to determine the MaxLength values. **Decimal Total Digits:** 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 field is null for TIME and TIMESTAMP data types. **Decimal Fractional Digits:** For TIME and TIMESTAMP types and all INTERVAL types with a SECOND field, this field returns a value of 0 - 6, indicating the fractional precision of seconds. For INTERVAL types without a SECOND field, the field returns null.
SZ	TIMESTAMP WITH TIME ZONE			
YR	INTERVAL YEAR			
YM	INTERVAL YEAR TO MONTH			
MO	INTERVAL MONTH			
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			
MI	INTERVAL MINUTE			
Code	**Description**	CHAR(2)	yes	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 field. The Type Codes GF and GV are no longer returned for columns defined as CHARACTER(n) CHARACTER SET GRAPHIC.
CF	CHARACTER			
CV	CHARACTER VARYING			
BF	BYTE			
BV	BYTE VARYING			
NULLABLE		CHAR	yes	None.
Code	**Description**			
Y	can be NULL			
N	is NOT NULL			
FORMAT		CHAR (30)	yes	Display format for the data type. Data type display formats appear in "Output Format Phrases" of *Teradata RDBMS SQL Reference, Volume 3.*

Field	Data Type	Nullable?	Description
MAX LENGTH	INTEGER	yes	Maximum amount of storage (in bytes). For CHARACTER data, Max Length is the length of the column declaration in bytes. You can use the DataInfo parcel to return the description of items within the Data Field of the corresponding indicator mode Record parcels. 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 Teradata.
DECIMAL TOTAL DIGITS	SMALLINT		If type is decimal.
DECIMAL FRACTIONAL DIGITS	SMALLINT	yes	If type is decimal.
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.
UPPERCASE	CHAR	yes	If type is CHAR or VARCHAR.
Code	**Description**		
N	Not declared as UPPERCASE or CASESPECIFIC		
U	UPPERCASE		
C	CASESPECIFIC		
B	Both UPPERCASE and CASESPECIFIC		
TABLE/VIEW?	CHAR		Shows column is from table or from view.
Code	**Description**		
T	Table		
V	View		
INDEXED?	CHAR	yes	If column is single-column index. Multiple-column indexes also return N.
Code	**Description**		
Y	Used in an index		
N	Not used in an index		

Field		Data Type	Nullable?	Description
UNIQUE?		CHAR	yes	If column is single-column index. If INDEXED? attribute is N, then returns null.
Code	**Description**			
Y	Index is unique			
N	Index is not unique			
PRIMARY?		CHAR	yes	If column is single-column index. Multiple-column indexes return null result. If INDEXED? attribute is N, then returns null.
Code	**Description**			
P	Primary index			
S	Secondary index			
TITLE		VARCHAR (60)	yes	None.
COLUMN CONSTRAINT		VARCHAR (255)	yes	See example under the previous section, HELP COLUMN.
CHAR TYPE		SMALLINT	yes	Returns the type of the different character data type columns. If the data type of the column is not character, returns null.
Code	**Description**			
1	Latin character data type columns			
2	Unicode character data type columns			
3	KanjiSJIS character data type columns			
4	Graphic character data type columns			
5	Kanji1 character data type columns			
IDCOL TYPE		CHAR(2)	yes	Returns a code for the type of identity column. If the column is not an identity column, returns null.
Code	**Description**			
GA	Generated always			
GD	Generated by default			

HELP CONSTRAINT

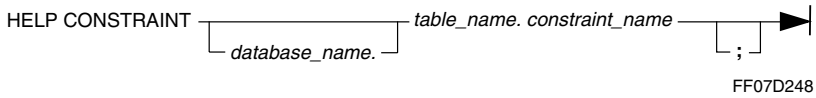
Purpose

Displays the attributes for a specific named constraint on a table. Unnamed constraint information can be obtained using the SHOW TABLE statement (see "SHOW HASH INDEX SHOW JOIN INDEX SHOW MACRO SHOW PROCEDURE SHOW TABLE SHOW TRIGGER SHOW VIEW" on page 4-79.

Invocation

Executable.

Syntax



where:

Syntax element...	Specifies...
database_name	the optional name of the database in which the table is defined.
table_name.constraint_name	the table-qualified name of a constraint for which attribute information is required.

ANSI Compliance

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

Authorization

You must either own the table in which the constraint is defined or have at least one privilege on that table.

Supported Constraint Types

There are three main types of constraints:

- Check Constraint
- Referential Constraint
- Unique Constraint

Constraint information for each of these is displayed in the following format.

Check Constraint Attributes

Attribute	Data Type	Nullable
CONSTRAINT NAME	VARCHAR(30)	no
CONSTRAINT TYPE (CHECK)	VARCHAR(15)	no
CONSTRAINT	VARCHAR(255)	no

Referential Constraint Attributes

Attribute		Data Type	Nullable
CONSTRAINT NAME		VARCHAR(30)	no
CONSTRAINT TYPE (REFERENCE)		VARCHAR(15)	no
STATE		CHARACTER(12)	no
Code	**Description**		
VALID	Valid constraint		
INVALID	Non-valid constraint		
INCONSISTENT	Inconsistent constraint		
UNRESOLVED	Unresolved constraint		
REFERENCE INDEX ID		SMALLINT	no
FOREIGN KEY COLUMNS		VARCHAR(512)	yes
PARENT KEY DATABASE NAME		VARCHAR(30)	yes
PARENT TABLE NAME		VARCHAR(30)	yes
PARENT INDEX ID (except UNRESOLVED)		SMALLINT	yes
PARENT KEY COLUMNS		VARCHAR(512)	yes

Unique Constraint Attributes

Attribute		Data Type	Nullable
CONSTRAINT NAME		VARCHAR(30)	no
CONSTRAINT TYPE		VARCHAR(15)	no
Code	**Description**		
PRIMARY INDEX	Constraint is on a primary index		
PRIMARY KEY	Constraint is on a primary key		
UNIQUE	Constraint defines uniqueness		
SECONDARY INDEX	Constraint is on a secondary index		
UNIQUE?		CHARACTER	
INDEX ID		SMALLINT	no
COLUMN NAMED		VARCHAR(512)	no

Standard Form of Display for HELP CONSTRAINT

HELP CONSTRAINT displays character constant strings, including CONSTRAINT 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.

Examples

The following example set shows how the HELP CONSTRAINT statement can be used.

Example 1

This example shows the report returned for a table with a CHECK constraint.

```
HELP CONSTRAINT table_1.check_1;


Name      Type            FK Columns

check_1   CHECK           CHECK("field_1">0
                          AND"field_2">0)
```

Example 2

The following example shows the report returned for a table with REFERENTIAL constraints.

```
HELP CONSTRAINT table_1.reference_1;
```

```
                                    Parent
                          FK        Table     PK
Name          Type        Columns   Name      Columns

reference_1   REFERENCE   field_1,  table_2   field_3,
                          field_2             field_4
```

Example 3

The following example shows the report returned for a table with a REFERENTIAL constraint on the primary key.

```
HELP CONSTRAINT table_1.primary_1;
```

```
Name          Type               Columns

primary_1     PRIMARY KEY        field_1, field_2
```

Example 4

The following example shows the report returned for a table with a UNIQUE constraint.

```
HELP CONSTRAINT table_1.unique_1;
```

```
Name          Type               Columns

unique_1      UNIQUE             field_3, field_4
```

Example 5

The following example shows how Teradata handles a HELP CONSTRAINT request that does not fit the tabled being analyzed.

```
HELP CONSTRAINT table_1.no_such_check;
```

The error message, 'The specified constraint name does not exist in table', is returned if table_1 does not have a constraint with the name '*no_such_check*'.

HELP DATABASE
HELP USER

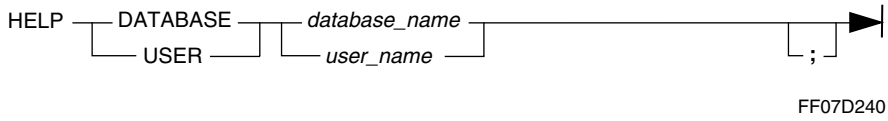
Purpose

Displays the attributes, sorted by object name, for all tables, views, join indexes, hash indexes, stored procedures, and macros in the specified database.

Invocation

Executable.

Syntax



FF07D240

where:

Syntax Element...	Specifies...
database_name	the name of the database about which information is required.
user_name	the name of the user about whom information is required.

Note that DATABASE and USER are interchangeable in this statement: either keyword returns the same result for a particular database or user. You should use the appropriate keyword for clarity.

ANSI Compliance

HELP DATABASE and HELP USER are Teradata extensions to the ANSI SQL-99 standard.

Authorization

You must either own the user or database or have at least one privilege on that user or database with one exception: HELP USER DBC does not require any rights on user DBC.

Database/User Attributes

<table>
<tr><th>Field</th><th>Data Type</th></tr>
<tr><td>NAME</td><td>CHARACTER(30)</td></tr>
<tr><td>TVM KIND</td><td>CHARACTER
<table><tr><th>Code</th><th>Description</th></tr><tr><td>G</td><td>Trigger</td></tr><tr><td>I</td><td>Join Index</td></tr><tr><td>J</td><td>Journal Table</td></tr><tr><td>M</td><td>Macro</td></tr><tr><td>N</td><td>Hash Index</td></tr><tr><td>P</td><td>Procedure</td></tr><tr><td>T</td><td>Table</td></tr><tr><td>V</td><td>View</td></tr></table></td></tr>
<tr><td>COMMENT</td><td>VARCHAR(255)</td></tr>
<tr><td>PROTECTION</td><td>CHARACTER
<table><tr><th>Code</th><th>Description</th></tr><tr><td>N</td><td>None</td></tr><tr><td>F</td><td>Fallback</td></tr></table></td></tr>
<tr><td>CREATOR NAME</td><td>CHARACTER(30)</td></tr>
<tr><td>COMMIT OPTION

Note: For temporary tables only.</td><td>CHARACTER
<table><tr><th>Code</th><th>Description</th></tr><tr><td>D</td><td>On COMMIT, delete rows.</td></tr><tr><td>P</td><td>On COMMIT, preserve rows.</td></tr></table></td></tr>
<tr><td>TRANSACTION LOG

Note: For temporary tables only.</td><td>CHARACTER
<table><tr><th>Code</th><th>Description</th></tr><tr><td>N</td><td>No log kept</td></tr><tr><td>Y</td><td>Log transactions</td></tr></table></td></tr>
</table>

Example

The following statement returns the following information about the Personnel database:

```
HELP DATABASE Personnel;

Table/View/Macro Name Kind Comment  Protection    Creator Name
--------------------- ---- -------  ----------    --------------
Department            T                      F   Administration
   .                  .                      .       .
   .                  .                      .       .
```

HELP HASH INDEX

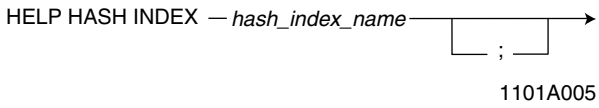
Purpose

Displays the data types of the columns defined by a particular hash index.

Invocation

Interactive.

Syntax



where:

Syntax element ...	Specifies ...
hash_index_name	the name of the hash index for which information is desired.

ANSI Compliance

HELP HASH INDEX is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must either own the table on which the hash index is defined or have at least one privilege on that table.

Hash Index Attributes

The attributes and display format for a hash index are identical to that of the HELP TABLE statement—in fact, the output of a HELP TABLE statement performed against a hash index is identical to that produced by a HELP HASH INDEX statement performed against the same hash index.

See "HELP MACRO HELP TABLE HELP VIEW" on page 4-40 for information about the attributes and display format for HELP HASH INDEX.

Example

The following example shows the output of a HELP HASH INDEX statement performed on the hash index named OrdHIdx.

```
HELP HASH INDEX OrdHIdx;

Column Name     Type Comment
--------------- ---- -----------------------------------
o_orderdate     DA   ?
```

HELP INDEX

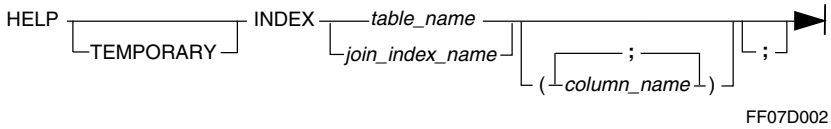
Purpose

Displays the attributes for the indexes defined for a table or join index.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies...
TEMPORARY	that the indexes in question are defined on a temporary table. This keyword is valid only when used with the *table_name* variable.
table_name	the table on which a report of all indexes (both primary and secondary) is desired.
join_index_name	the join index on which a report of all indexes (both primary and secondary) is desired. Also see "HELP JOIN INDEX" on page 4-38.
column_name	a column or column list for which an index or index list is desired. Use this syntax to exclude indexes defined on columns for which you are not interested.

ANSI Compliance

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

Authorization

You must either own the table or join index on which the index is defined or have at least one privilege on that table or join index.

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 HASH INDEX SHOW JOIN INDEX SHOW MACRO SHOW PROCEDURE SHOW TABLE SHOW TRIGGER SHOW VIEW" on page 4-79).

HELP INDEX and Hash, Join and Covering Indexes

HELP INDEX does not report information about hash or join indexes.

HELP INDEX does not report the ordering columns for a covering index or value-ordered join index. SHOW TABLE reports these columns (see "SHOW HASH INDEX SHOW JOIN INDEX SHOW MACRO SHOW PROCEDURE SHOW TABLE SHOW TRIGGER SHOW VIEW" on page 4-79).

Other Resources for Information About Indexes

You can use the system view DBC.Indices to find additional information about indexes. See "DBC.Indices" in *Teradata RDBMS Data Dictionary*.

Index Attributes

<table>
<tr><th>Attribute</th><th>Format</th><th>Description</th></tr>
<tr><td>UNIQUE?</td><td>CHARACTER</td><td>Defines whether the index is unique or nonunique.

<table><tr><th>Code</th><th>Definition</th></tr><tr><td>N</td><td>Not unique.</td></tr><tr><td>Y</td><td>Unique.</td></tr></table></td></tr>
<tr><td>PRIMARY OR SECONDARY?</td><td>CHARACTER</td><td>Defines whether the index is primary or secondary.

<table><tr><th>Code</th><th>Definition</th></tr><tr><td>P</td><td>Primary index.</td></tr><tr><td>S</td><td>Secondary index.

This code does not discriminate among USIs, regular NUSIs, or join indexes: all types of secondary index are reported as S.</td></tr></table></td></tr>
</table>

Attribute	Format	Description
COLUMN NAMES	VARCHAR(512)	Lists the names of the columns defined for the index.
INDEX ID	System-defined	
APPROXIMATE COUNT	FLOAT	Defines the approximate number of rows in the index subtable.
INDEX NAME	CHARACTER(30)	Lists the index name, if there is one. This column is set to NULL if the index is unnamed.
ORDERED OR PARTITIONED?	CHARACTER(1) CHARACTERSET LATIN	Defines whether the index is ordered or partitioned.

Code	Definition
H	Hash-ordered index.
P	Partitioned index.
V	Value-ordered index.

Example 1

The following example returns information about the specified index or indexes defined on the table columns named cname_1, cname_2, and cname_3.

```
HELP INDEX tname (cname_1, cname_2, cname_3);
```

Example 2

The response to the following statement shows that one index is defined for the Department table.

```
HELP INDEX Personnel.Department;
        Primary
        or
Unique  Secondary Column Names    Index Id Approximate Count
------  --------- --------------- -------- ----------------
Y       P         DeptNo                 1                7
```

Example 3: Partitioned Primary Index

This example uses the following table definition:

```
CREATE TABLE Orders
     (o_orderkey INTEGER NOT NULL,
      o_custkey INTEGER,
      o_orderstatus CHAR(1) CASESPECIFIC,
      o_totalprice DECIMAL(13,2) NOT NULL,
      o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      o_orderpriority CHAR(21),
      o_clerk CHAR(16),
      o_shippriority INTEGER,
      o_comment VARCHAR(79))
   PRIMARY INDEX OrdPI (o_orderkey)
PARTITION BY RANGE_N(
  o_orderdate BETWEEN DATE '1992-01-01' AND DATE '1998-12-31'
  EACH INTERVAL '1' MONTH)
   UNIQUE INDEX OrdUSI (o_orderkey)
   INDEX Ordx1 (o_orderdate)
   INDEX Ordx2 (o_orderdate) ORDER BY VALUES (o_orderdate)
   INDEX Ordx3 (o_custkey, o_orderdate) ORDER BY VALUES (o_orderdate)
   INDEX Ordx4 (o_custkey, o_orderdate) ORDER BY HASH (o_orderdate)
   INDEX Ordx5 ALL (o_custkey, o_orderstatus) ORDER BY VALUES (o_custkey)
   INDEX Ordx6 ALL (o_custkey, o_orderstatus) ORDER BY HASH (o_custkey)
   INDEX Ordx7 ALL (o_custkey) ORDER BY VALUES (o_custkey)
   INDEX Ordx8 ALL (o_custkey) ORDER BY HASH (o_custkey)
   INDEX Ordx9 (o_shippriority) ORDER BY HASH (o_shippriority)
   INDEX OrdxA ALL (o_clerk) ORDER BY HASH (o_clerk)
   INDEX OrdxB (o_orderkey, o_orderdate);
```

HELP INDEX generates the following report on this table. The BTEQ commands are included to show how the report format was derived. Attributes of particular interest for this example are highlighted in boldface.

```
BTEQ -- Enter your DBC/SQL request or BTEQ command:
.sidetitles
.sidetitles
 BTEQ -- Enter your DBC/SQL request or BTEQ command:
.foldline
.foldline
 BTEQ -- Enter your DBC/SQL request or BTEQ command:
help index orders;
help index orders;

 *** Help information returned. 13 rows.
 *** Total elapsed time was 1 second.
```

```
                  Unique? N
  Primary//or//Secondary? P
             Column Names o_orderkey
                 Index Id 1
        Approximate Count 0
               Index Name OrdPI
Ordered//or//Partitioned? P
                  Unique? Y
  Primary//or//Secondary? S
             Column Names o_orderkey
                 Index Id 4
        Approximate Count 0
               Index Name OrdUSI
Ordered//or//Partitioned? H
                  Unique? N
  Primary//or//Secondary? S
             Column Names o_orderdate
                 Index Id 8
        Approximate Count 0
               Index Name Ordx1
Ordered//or//Partitioned? H
                  Unique? N
  Primary//or//Secondary? S
             Column Names o_orderdate
                 Index Id 12
        Approximate Count 0
               Index Name Ordx2
Ordered//or//Partitioned? V
                  Unique? N
  Primary//or//Secondary? S
             Column Names o_custkey, o_orderdate
```

```
                    Index Id 16
           Approximate Count 0
                  Index Name Ordx3
Ordered//or//Partitioned? V
                     Unique? N
  Primary//or//Secondary? S
              Column Names o_custkey, o_orderdate
                    Index Id 24
           Approximate Count 0
                  Index Name Ordx4
Ordered//or//Partitioned? H
                     Unique? N
  Primary//or//Secondary? S
              Column Names o_custkey, o_orderstatus
                    Index Id 32
           Approximate Count 0
                  Index Name Ordx5
Ordered//or//Partitioned? V
                     Unique? N
  Primary//or//Secondary? S
              Column Names o_custkey, o_orderstatus
                    Index Id 40
           Approximate Count 0
                  Index Name Ordx6
Ordered//or//Partitioned? H
                     Unique? N
  Primary//or//Secondary? S
              Column Names o_custkey
                    Index Id 48
           Approximate Count 0
                  Index Name Ordx7
```

```
Ordered//or//Partitioned? V
                  Unique? N
  Primary//or//Secondary? S
             Column Names o_custkey
                 Index Id 52
        Approximate Count 0
               Index Name Ordx8
Ordered//or//Partitioned? H
                  Unique? N
  Primary//or//Secondary? S
             Column Names o_shippriority
                 Index Id 56
        Approximate Count 0
               Index Name Ordx9
Ordered//or//Partitioned? H
                  Unique? N
  Primary//or//Secondary? S
             Column Names o_clerk
                 Index Id 60
        Approximate Count 0
               Index Name OrdxA
Ordered//or//Partitioned? H
                  Unique? N
  Primary//or//Secondary? S
             Column Names o_orderkey, o_orderdate
                 Index Id 64
        Approximate Count 0
               Index Name OrdxB
Ordered//or//Partitioned? H
```

HELP JOIN INDEX

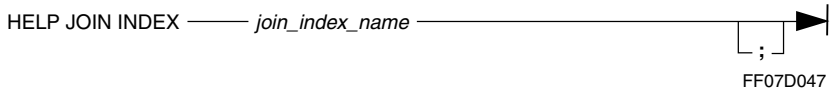
Purpose

Displays the attributes of the columns defined by a particular join index.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies...
join_index_name	the name of the join index for which information is desired.

ANSI Compliance

HELP JOIN INDEX is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must either own the tables on which the join index is defined or have at least one privilege on each of those tables.

Join Index Attributes

The attributes and display format for a join index are identical to that of the HELP TABLE statement—in fact, the output of 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 MACRO HELP TABLE HELP VIEW" on page 4-40 for information about the attributes and display format for HELP JOIN INDEX.

Example

The following example shows the output of a HELP JOIN INDEX statement performed on the join index named OrdCustIdx.

```
HELP JOIN INDEX OrdCustIdx;


Column Name     Type Comment
--------------- ---- ------------------
c_custkey       I    ?
c_name          CF   ?
o_status        CF   ?
o_date          DA   ?
o_comment       CV   ?
```

HELP MACRO
HELP TABLE
HELP VIEW

Purpose

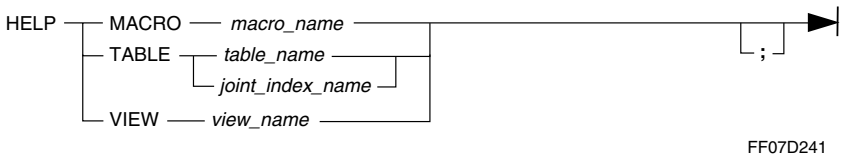
Displays the attributes for the specified join index, table, view, or macro.

You can also use this statement to report on a join index.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies the name of the ...
macro_name	macro for which help is required.
table_name	table for which help is required.
join_index_name	join index for which help is required.
view_name	view for which help is required.

ANSI Compliance

HELP MACRO, HELP TABLE, and HELP VIEW are Teradata extensions to the ANSI SQL-99 standard.

Authorization

You must either own the join index tables, macro, view, or table or have at least one privilege on that object.

Table, Hash Index, Join Index, View, or 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.

Field		Data Type	Nullable?	Comments
COLUMN NAME		CHARACTER(30)	no	
TYPE		CHARACTER(2)	yes	
Code	**Description**			
I1	BYTEINT			
I2	SMALLINT			
I	INTEGER			
D	DECIMAL NUMERIC			
F	FLOAT REAL DOUBLE PRECISION			
DA	DATE	CHARACTER(2)	yes	**MaxLength Characteristics**: MaxLength is 4, representing the internal storage size. **Host Data Characteristics**: When DATEFORM=ANSIDATE, this field should be described externally as CHAR(10). **Format Characteristics**: 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'.

Field		Data Type	Nullable?	Comments
TYPE (continued)		CHARACTER(2)	yes	**Host Data Characteristics**: Interpret as the equivalent of CF, or CHAR(n) data type. **MaxLength Characteristics**: MaxLength is the length of the *external* CharFix representation of the field, defined as CHAR(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. See "External Representation" for each individual DateTime and Interval type (see Chapter 3: "DateTime and Interval Data Types" in *Teradata RDBMS SQL Reference, Volume 3*) to determine the MaxLength values. **Decimal Total Digits**: 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 field is null for TIME and TIMESTAMP data types. **Decimal Fractional Digits**: For TIME and TIMESTAMP types and all INTERVAL types with a SECOND field, this field returns a value of 0–6, indicating the fractional precision of seconds. For INTERVAL types without a SECOND field, the field returns null.
Code	**Description**			
AT	TIME			
TS	TIMESTAMP			
TZ	TIME WITH TIME ZONE			
SZ	TIMESTAMP WITH TIME ZONE			
YR	INTERVAL YEAR			
YM	INTERVAL YEAR TO MONTH			
MO	INTERVAL MONTH			
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			
SC	INTERVAL SECOND			

Field		Data Type	Nullable?	Comments
TYPE (continued)		CHARACTER(2)	yes	
Code	**Description**			
CF	CHARACTER	CHARACTER(2)	yes	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 field.
CV	CHARACTER VARYING			
BF	BYTE			
BV	BYTE VARYING			
COMMENT		VARCHAR (255)	yes	
NULLABLE		CHARACTER	yes	
Code	**Description**			
Y	can be NULL			
N	is NOT NULL			
TITLE		VARCHAR (60)	yes	
MAX LENGTH		INTEGER	yes	Maximum amount of storage (in bytes). For CHARACTER data, Max Length is the length of the column declaration in bytes. You can use the DataInfo parcel to return the description of items within the Data Field of the corresponding indicator mode Record parcels. 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 Teradata.
DECIMAL TOTAL DIGITS		SMALLINT		If type is decimal.
DECIMAL FRACTIONAL DIGITS		SMALLINT	yes	If type is decimal.
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.

Field		Data Type	Nullable?	Comments
UPPERCASE		CHARACTER	yes	If type is CHAR or VARCHAR.
Code	**Description**			
N	not declared as UPPERCASE or CASESPECIFIC			
U	UPPERCASE			
C	CASESPECIFIC			
B	BOTH			
TABLE/VIEW		CHARACTER		If HELP TABLE or HELP VIEW
Code	**Description**			
T	table			
V	view			
DEFAULT VALUE		VARCHAR (255)	yes	
CHAR TYPE		SMALLINT	yes	Returns the type of the different character data type columns. If the data type of the column is not character, returns null.
Code	**Description**			
1	Latin character data type columns			
2	Unicode character data type columns			
3	KanjiSJIS character data type columns			
4	Graphic character data type columns			
5	Kanji1 character data type columns			
IDCOL TYPE		CHARACTER(2)	yes	Returns a code for the type of identity column.
Code	**Description**			
GA	Generated always			
GD	Generated by default			

Table and Join Index Columns

For some larger tables and join indexes, a request to display all or most of the columns will result in 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.columns.

View Columns

If a view has more than 50 columns, HELP VIEW returns an empty result. To obtain column information from oversized views, use the HELP statement in the form:

```
HELP COLUMN view_name.*
```

If you perform HELP VIEW for a view that has semantic errors in its definition, Teradata does not return an error message.

For details on defining views, see the Usage Notes of "CREATE VIEW REPLACE VIEW" on page 1-311.

Macro Parameters

If a macro contains no parameters, a message is returned to indicate that no parameters have been defined.

If you perform HELP MACRO for a macro that has semantic errors in its definition, Teradata does not return an error message.

HELP MACRO can also be used to obtain a comment that explains how to use the macro.

Example 1: HELP MACRO

The following statement returns information about the NewEmp macro.

```
HELP MACRO NewEmp;

Parameter Name  Type   Comment
--------------   ----  ---------------------------------------
Name             CV    Employee name, last name first; required
Number           I     Employee number; required
Dept             I2    Department number; required
```

Example 2: HELP TABLE

The following statement returns information about the Department table.

```
HELP TABLE Personnel.Department;

Column Name  Type   Comment
-----------  ----  ---------------------------------------
DeptNo       I2     Department number
DeptName     CV     Department name
Loc          CF     Department location
MgrNo        I      Employee number of department manager
```

HELP PROCEDURE

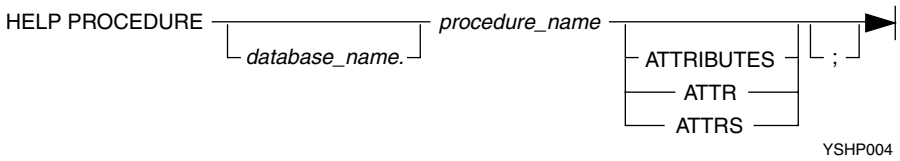
Purpose

Displays the attribute and format parameters for each parameter of the procedure or just the creation time attributes for the specified procedure.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies...
database_name	the name of the database in which the procedure is stored. If no database is specified, then the current default database is assumed.
procedure_name	the name of the procedure for which the attributes and format of each parameter in the procedure is desired.
ATTRIBUTES ATTR ATTRS	a keyword indicating that the stored procedure creation-time attributes from the column SPObjectCodeRows in DBC.TVM are to be displayed rather than the attribute and format information for the procedure parameters.

ANSI Compliance

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

Authorization

You must have one of the following access privileges to perform HELP PROCEDURE:

- At least one procedure-specific privilege on the database containing the procedure
- At least one privilege on the specified stored procedure

Procedure Parameter Attributes

The following attribute and format information for each parameter in a stored procedure are returned.

If the specified stored procedure has no parameters, the “Empty help” message is returned.

Attribute		Format
PARAMETER NAME		CHARACTER(30)
DATA TYPE		CHARACTER(2)
Code	**Description**	
I1	BYTEINT	
I2	SMALLINT	
I	INTEGER	
D	DECIMAL/NUMERIC	
F	FLOAT/REAL/DOUBLE PRECISION	
DA	DATE	
AT	TIME	
TS	TIMESTAMP	
TZ	TIME WITH TIME ZONE	
YR	INTERVAL YEAR	
YM	INTERVAL YEAR TO MONTH	
MO	INTERVAL MONTH	
DY	INTERVAL DAY	
DH	INTERVAL DAY TO HOUR	
DM	INTERVAL DAY TO MINUTE	
HR	INTERVAL HOUR	
HM	INTERVAL HOUR TO MINUTE	
HS	INTERVAL HOUR TO SECOND	
SC	INTERVAL SECOND	
CF	CHARACTER	
CV	CHARACTER VARYING	
BF	BYTE	
BV	BYTE VARYING	

Attribute		Format
COMMENT		CHARACTER(255)
NULLABLE		CHARACTER
Code	**Description**	
Y	Can be NULL	
N	Is NOT NULL	
FORMAT		CHARACTER(30)
TITLE		CHARACTER(60)
MAX LENGTH		INTEGER
DECIMAL TOTAL DIGITS		SMALLINT
DECIMAL FRACTIONAL DIGITS		SMALLINT
UPPERCASE		CHARACTER
Code	**Description**	
N	Not declared as UPPERCASE or CASESPECIFIC	
U	UPPERCASE	
C	CASESPECIFIC	
B	Both UPPERCASE and CASESPECIFIC	
TVM KIND		CHARACTER
Code	**Description**	
P	Procedure	
DEFAULT VALUE		CHARACTER(255)
CHARACTER TYPE		SMALLINT
Code	**Description**	
1	Latin character data type columns	
2	Unicode character data type columns	
3	KanjiSJIS character data type columns	
4	Graphic character data type columns	
5	Kanji1 character data type columns	

Attribute		Format
PARAMETER TYPE		CHARACTER
Code	**Description**	
I	IN	
B	INOUT	
O	OUT	

Procedure Attributes

Attribute		Format
TRANSACTION SEMANTICS		CHARACTER(21)
Value	**Description**	
ANSI	ANSI transaction semantics	
Teradata	Teradata transaction semantics	
PRINT MODE		CHARACTER(1)
Code	**Description**	
Y	WITH PRINT	
N	WITH NOPRINT	
PLATFORM		CHARACTER(30)
Value	**Description**	
UNIX MP-RAS	NCR UNIX operating system	
Windows	Microsoft Windows operating system	
CHARACTER SET		CHARACTER(30)
Code	**Description**	
ASCII	ASCII character set	
EBCDIC	EBCDIC character set	
KanjiEUC	Kanji Extended UNIX Code character set	
KanjiSJIS	Kanji Shift Japanese Industrial Standard Code (DOS Kanji) character set	
KanjiEBCDIC	Kanji EBCDIC character set	

<table>
<tr><th colspan="2">Attribute</th><th>Format</th></tr>
<tr><td colspan="2">DEFAULT CHARACTER DATA TYPE</td><td rowspan="6">CHARACTER(30)</td></tr>
<tr><td>Code</td><td>Description</td></tr>
<tr><td>LATIN</td><td>Teradata Latin character type</td></tr>
<tr><td>UNICODE</td><td>Unicode character type</td></tr>
<tr><td>KANJI1</td><td>Noncanonical Kanji character type</td></tr>
<tr><td>KANJISJIS</td><td>Canonical Kanji: Shift Japanese Industrial Standard character type</td></tr>
<tr><td>GRAPHIC</td><td>IBM DB2 Graphic character type</td></tr>
<tr><td colspan="2">COLLATION</td><td rowspan="3">CHARACTER(30)</td></tr>
<tr><td>Code</td><td>Description</td></tr>
<tr><td>• ASCII
• EBCDIC
• Charset_Coll
• Multinational
• JIS_Coll</td><td>ASCII character set is collated first, followed by the others in the sequence specified.</td></tr>
<tr><td colspan="2">STORED PROCEDURE TEXT</td><td rowspan="3">CHARACTER(1)</td></tr>
<tr><td>Code</td><td>Description</td></tr>
<tr><td>Y</td><td>Statements defining the stored procedure are stored in the procedure object.</td></tr>
<tr><td>N</td><td>Statements defining the stored procedure are not stored.</td></tr>
<tr><td colspan="2">VERSION NUMBER</td><td rowspan="5">BYTEINT</td></tr>
<tr><td>Code</td><td>Description</td></tr>
<tr><td>00</td><td>Procedure created in release 4.0</td></tr>
<tr><td>01</td><td>Procedure created in release 4.1</td></tr>
<tr><td>02</td><td>Procedure created in release 4.2</td></tr>
<tr><td>03</td><td>Procedure created in release 5.0</td></tr>
<tr><td colspan="2">DEFAULT DATABASE

The default database name for procedures created in release 4.0 is NULL.</td><td>CHARACTER(30)</td></tr>
</table>

Attribute		Format
WARNING OPTION		CHARACTER(1)
Code	**Description**	
Y	All compilation warnings generated during stored procedure compilation or recompilation are returned to the user. Y is the default warning option.	
N	All compilation warnings are suppressed.	

HELP SESSION

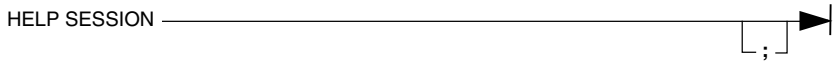
Purpose

Displays attribute information for the user of the current session.

Invocation

Executable.

Syntax



FF07D242

ANSI Compliance

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

Authorization

You must either own the session or the user or database that is using that session.

Session Attributes

Attribute	Description
USER NAME	Name of the owner of the session.
ACCOUNT NAME	Account name currently in effect for the session (see "SET SESSION" on page 1-384).
LOGON DATE	Date of logon to Teradata, in the format YY/MM/DD.
LOGON TIME	Time of logon to Teradata, in the format HH:MI:SS.
CURRENT DATABASE	Database currently defined as the default for the session (see "DATABASE" on page 1-325 and "CREATE USER" on page 1-296).

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>COLLATION</td><td>Sort sequence currently defined for the session (see “SET SESSION” on page 1-384 and “CREATE USER” on page 1-296).
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>ASCII</td><td>Order it would appear if converted for an ASCII session, and a binary sort performed.</td></tr>
<tr><td>EBCDIC</td><td>Order it would appear if converted for an EBCDIC session and a binary sort performed.</td></tr>
<tr><td>MULTINATIONAL</td><td>Two-level collation based on the Unicode collation standard.
For backward compatibility, the following are true:
• MULTINATIONAL collation of KANJI1 data is single level.
• Single byte character collation is redefineable by the system administrator.</td></tr>
<tr><td>CHARSET_COLL</td><td>Order it would appear if converted to the current client character set and then sorted in binary order.</td></tr>
<tr><td>JIS_COLL</td><td>Japanese Industrial Standards (JIS) in this order:
1 JIS X 0201-defined characters in standard order
2 JIS X 0208-defined characters in standard order
3 JIS X 0212-defined characters in standard order
4 KanjiEBCDIC-defined characters not defined in JIS X 0201, JIS X 0208, or JIS X 0212 in standard order
5 All remaining characters in Unicode standard order</td></tr>
</table></td></tr>
<tr><td>CHARACTER SET</td><td>Character set currently defined for the session.</td></tr>
<tr><td>TRANSACTION SEMANTICS</td><td>Whether the session is in ANSI or Teradata transaction mode.
Not nullable.
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>ANSI</td><td>Session is in ANSI mode.</td></tr>
<tr><td>Teradata</td><td>Session is in Teradata mode.</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>CURRENT DATEFORM</td><td>Current mode for export and import of DATE values.

For information about date formats, see Chapter 3: “DateTime and Interval Data Types” in Teradata RDBMS SQL Reference, Volume 3.

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>IntegerDate</td><td>Uses four-byte integer values.</td></tr>
<tr><td>ANSIDate</td><td>Uses Char(10) ANSI data format.</td></tr>
</table>
</td></tr>
<tr><td>SESSION TIME ZONE</td><td>Current Time Zone value in use by the session, displayed in INTERVAL HOUR TO MINUTE format.

For example, -08:00.</td></tr>
<tr><td>DEFAULT CHARACTER TYPE</td><td>Default character type for the user.</td></tr>
<tr><td>EXPORT WIDTH</td><td>Export width value for each character data type for the session (see “Teradata CHARACTER Strings and Client Physical Bytes” in the chapter “Character Data Types,” in Teradata RDBMS SQL Reference, Volume 3).

There are 5 export width attributes.

Fields one through four cover the following respective types.
• Latin
• Unicode
• Unicode Adj.
• KanjiSJIS
• Graphic

Field five reports the adjustment value for Unicode character data type with IBM mixed single byte/multibyte character sessions when the expected defaults are in force.

No export width is reported for Kanji1 character data type because the export width for such data is always one.</td></tr>
<tr><td>DEFAULT DATE FORMAT</td><td>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.

For information about date formats, see Chapter 3: “DateTime and Interval Data Types” in Teradata RDBMS SQL Reference, Volume 3.

For information about changing the default date output format in the SDF and the definition of date formatting characters, see Chapter 8: “Output Format Phrases” in Teradata RDBMS SQL Reference, Volume 3.</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>NON-MONETARY NUMERIC OUTPUT</td><td>Settings of attributes that apply to the output of non-monetary numeric data. Values are set in the Specification for Data Formatting (SDF) file by the DBA. The attributes correspond to specific formatting characters that can appear in the output FORMAT phrase of a non-monetary numeric type.

The following table lists the 3 monetary numeric output attributes:

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>RADIX SEPARATOR</td><td>Character that separates the integer and fractional part of non-monetary numeric data. The value is copied into the output string of numeric data whenever a D appears in the corresponding output FORMAT phrase.</td></tr>
<tr><td>GROUP SEPARATOR</td><td>Character that separates groups of digits in the integer part of non-monetary strings. The value of GROUPING RULE determines when the value of GROUP SEPARATOR is copied into the output string of numeric data.</td></tr>
<tr><td>GROUPING RULE</td><td>The number of digits to group together before inserting the value of GROUP SEPARATOR in the output of non-monetary numeric data. The grouping rule applies to the output of numeric data that uses a FORMAT phrase containing a G formatting character.</td></tr>
</table>

For information about changing the default non-monetary numeric output format in the SDF and the definition of non-monetary numeric formatting characters, see Chapter 8: “Output Format Phrases” in Teradata RDBMS SQL Reference, Volume 3.</td></tr>
<tr><td>MONETARY NUMERIC OUTPUT</td><td>Settings of attributes that apply to the output of monetary numeric data. Values are set in the Specification for Data Formatting (SDF) file by the database administrator.

The attributes correspond to specific formatting characters that can appear in the output FORMAT phrase of a monetary numeric type.

The following table lists the 9 monetary numeric output attributes:

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>CURRENCY RADIX SEPARATOR</td><td>Character that separates the integer and fractional part of monetary numeric data. The value is copied into the output string of numeric data whenever a D appears in the corresponding output FORMAT phrase.</td></tr>
<tr><td>CURRENCY GROUP SEPARATOR</td><td>Character that separates groups of digits in the integer part of monetary strings. The value of CURRENCY GROUPING RULE determines when the value of CURRENCY GROUP SEPARATOR is copied into the output string of numeric monetary data.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>MONETARY
NUMERIC OUTPUT

(continued)</td><td>
<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>CURRENCY
GROUPING RULE</td><td>The number of digits to group together before inserting the value of CURRENCY GROUP SEPARATOR in the output of monetary numeric data. The grouping rule applies to the output of numeric data that uses a FORMAT phrase containing a G formatting character.</td></tr>
<tr><td>CURRENCY NAME</td><td>Local currency as a completely spelled-out currency name. The value is copied into the output string of monetary numeric data whenever an N appears in the corresponding output FORMAT phrase.</td></tr>
<tr><td>CURRENCY</td><td>Local currency symbol. The value is copied into the output string of monetary numeric data whenever an L appears in the corresponding output FORMAT phrase.</td></tr>
<tr><td>ISOCURRENCY</td><td>Local currency as an uppercase, three-character code from ISO 4217. The value is copied into the output string of monetary numeric data whenever a C appears in the corresponding output FORMAT phrase.</td></tr>
<tr><td>DUAL CURRENCY
NAME</td><td>Dual currency as a completely spelled-out currency name. The value is copied into the output string of monetary numeric data whenever an A appears in the corresponding output FORMAT phrase.</td></tr>
<tr><td>DUAL CURRENCY</td><td>Dual currency symbol. The value is copied into the output string of monetary numeric data whenever an O appears in the corresponding output FORMAT phrase.</td></tr>
<tr><td>DUAL
ISOCURRENCY</td><td>Dual currency as an uppercase, three-character code from ISO 4217. The value is copied into the output string of monetary numeric data whenever a U appears in the corresponding output FORMAT phrase.</td></tr>
</table>
For information about changing the default monetary numeric output format in the SDF and the definition of monetary numeric formatting characters, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>DEFAULT DATA TYPE OUTPUT FORMATS</td><td>Default output formats of numeric, date, and time data types. Values are set in the Specification for Data Formatting (SDF) file by the database administrator.

The following table lists the 7 default data type output format attributes:

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>DEFAULT BYTEINT FORMAT</td><td>Default output format applied to BYTEINT data types.</td></tr>
<tr><td>DEFAULT INTEGER FORMAT</td><td>Default output format applied to INTEGER data types.</td></tr>
<tr><td>DEFAULT SMALLINT FORMAT</td><td>Default output format applied to SMALLINT data types.</td></tr>
<tr><td>DEFAULT NUMERIC FORMAT</td><td>Default output format applied to NUMERIC and DECIMAL data types.</td></tr>
<tr><td>DEFAULT REAL FORMAT</td><td>Default output format applied to REAL, DOUBLE PRECISION, and FLOAT data types.</td></tr>
<tr><td>DEFAULT TIME FORMAT</td><td>Default output format applied to TIME and TIME WITH TIME ZONE data types.</td></tr>
<tr><td>DEFAULT TIMESTAMP FORMAT</td><td>Default output format applied to TIMESTAMP and TIMESTAMP WITH TIME ZONE data types.</td></tr>
</table>

For information about changing the default data type output format in the SDF and the definition of formatting characters, see Chapter 8: “Output Format Phrases” in Teradata RDBMS SQL Reference, Volume 3.</td></tr>
<tr><td>CURRENT ROLE</td><td>Current role for the session.</td></tr>
<tr><td>LOGON ACCOUNT</td><td>Account name used to log on to Teradata.</td></tr>
</table>

Example 1

The following statement shows the attributes of the session for the current user.

```
HELP SESSION;
```

```
                     User Name     SPEC2
                  Account Name     DBC
                    Logon Date     02/08/31
                    Logon Time     10:58:56
               Current DataBase    DBC
                     Collation     ASCII
                Character Set      ASCII
           Transaction Semantics   Teradata
                Current DateForm   IntegerDate
              Session Time Zone    -08:00
         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   -9.99999999999999E-999
             Default Time format   HH:MI:SS.S(F)Z
       Default Timestamp format    YYYY-MM-DDBHH:MI:SS.S(F)Z
                   Current Role    Finance
                  Logon Account    DBC
```

Example 2

A KanjiEBCDIC user issues a HELP SESSION statement having an export width of the expected defaults. The help session was entered using BTEQ with the .SIDETITLES and .FOLDLINE formatting options.

The following information is reported.

```
                User Name  KANJI_USER
             Account Name  KANJI_USER
               Logon Date  02/05/23
               Logon Time  11:43:09
           Current Database  KANJI_USER
                 Collation  CHARSET_COLL
              Character Set  KANJIEBCDIC5035_0I
     Transaction Semantics  Teradata
          Current DateForm  ANSI
        Session Time Zone  -09:00
    Default Character Type  UNICODE
             Export Latin  1
           Export Unicode  2
   Export Unicode Adjusted  2
          Export KanjiSJIS  1
            Export Graphic  2
       Default Date Format  YYYY\u5E74MM\u6708DD\u65E5
           Radix Separator  .
           Group Separator  ,
             Grouping Rule  3
  Currency Radix Separator  .
  Currency Group Separator  ,
    Currency Grouping Rule  3
             Currency Name  Yen
                  Currency  \u00A5
               ISOCurrency  JPY
        Dual Currency Name  Yen
             Dual Currency  \u00A5
          Dual ISOCurrency  JPY
    Default ByteInt format  -(3)9
    Default Integer format  G-(I)9
  Default SmallInt format  G-(I)9
    Default Numeric format  G--(I)D9(F)
       Default Real format  G-9D99999999999999E-999
       Default Time format  HH\u6642MI\u5206SSDS(F)\u79D2Z
  Default Timestamp format  YYYY\u5E74MM\u6708DD\u65E5BHH\u6642MI\u5206SSDS(F)\u79D2Z
              Current Role  Finance
             Logon Account  KANJI_USER
```

HELP STATISTICS (Optimizer Form)

Purpose

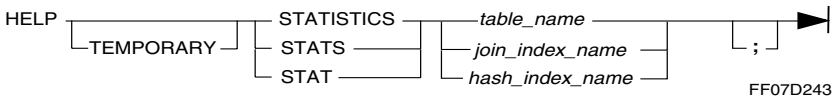
Displays attributes for the statistics that have been collected in the data dictionary for the specified table, join index, or hash index at the time they were collected. The report also displays the number of unique values.

You can view the complete table, join index, or hash index statistics using the STATS module of Teradata Manager.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies...
TEMPORARY	that statistics are desired for a temporary table.
table_name	the name of the table for which statistics are required.
join_index_name	the name of the join index for which statistics are required.
hash_index_name	the name of the hash index for which statistics are required.

ANSI Compliance

HELP STATISTICS (Optimizer Form) is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have one of the following privileges to perform the standard form of HELP STATISTICS:

- At least one privilege on *table_name*, *join_index_name*, or *hash_index_name*.
- Own *table_name*, *join_index_name*, or *hash_index_name*.

Support for Global Temporary and Volatile Tables

If you specify the keyword TEMPORARY then the statistics definitions for the local instance in the current session are displayed; else, the statistics definitions for the base temporary table are displayed.

Because no statistics are collected on volatile tables, HELP STATISTICS returns an error if you ask for statistics for a volatile table.

How HELP STATISTICS Behaves When No Statistics Exist

If no statistics have been collected for the table, a warning message is returned.

HELP STATISTICS information can be useful in determining whether statistics for the table should be updated or dropped.

Statistics Attributes

The data types for the table attributes reported by HELP STATISTICS are as follows:

Attribute	Format
Collection Date	CHARACTER(8)
Collection Time	CHARACTER(8)
Unique Values	VARCHAR(20)
Column Names	VARCHAR(512)

Example

If statistics have been collected for the Employee table, you can use the following statement to obtain a list of the columns and indexes of the Employee table that currently have statistics:

```
HELP STATISTICS Personnel.Employee;
```

The result shows that statistics were last collected for YrsExp on 02/06/17, when there were 17 unique values for the column:

```
Date     Time     Unique Values        Column Names
-------- -------- -------------------- ----------------
02/06/17 13:12:45                   17 YrsExp
```

You can view the table statistics for these columns and indexes using the STATS module of Teradata Manager.

For More Information

For information about collecting and dropping optimizer statistics, see the following:

- "COLLECT STATISTICS (Optimizer Form)" on page 3-14
- "DROP STATISTICS (Optimizer Form)" on page 3-31

HELP STATISTICS (QCD Form)

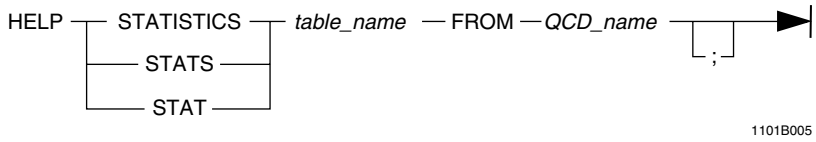
Purpose

Displays attributes for the statistics that have been collected in the TableStatistics table of the specified QCD database for the specified table at the time they were collected. The report also displays the number of unique values.

Invocation

Executable.

Syntax



where:

Syntax element …	Specifies …
table_name	the name of the table for which statistics are required.
FROM *QCD_name*	that the statistics to be retrieved are to be found in the TableStatistics table of the specified QCD database.

ANSI Compliance

HELP STATISTICS (QCD Form) is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have one of the following privileges to perform the QCD form of HELP STATISTICS:

- At least one privilege on *table_name.*
- Own *table_name* and have SELECT privilege on the TableStatistics table in *QCD_name*.

How HELP STATISTICS Behaves When No Statistics Exist

If no statistics have been collected for the table, a warning message is returned.

HELP STATISTICS information can be useful in determining whether statistics for the table should be updated or dropped.

Statistics Attributes

The data types for the table attributes reported by HELP STATISTICS are as follows:

Attribute	Format
Collection Date	CHARACTER(8)
Collection Time	CHARACTER(8)
Unique Values	VARCHAR(20)
Column Names	VARCHAR(512)

Example

The report produced by HELP STATISTICS (QCD Form) is identical to that produced by HELP STATISTICS (Optimizer Form).

Assume you collected statistics on the Employee table at the same time using COLLECT STATISTICS (Optimizer Form) and COLLECT STATISTICS (QCD Form) with a 99 percent sample percentage[1], then the following statement would report the columns and indexes of the Employee table that currently have sampled statistics in the TableStatistics table of the MyQCD database:

```
HELP STATISTICS Personnel.Employee FROM MyQCD;
```

The result shows that statistics were last collected for YrsExp on 02/06/17, when there were 17 unique values for the column:

```
Date     Time     Unique Values        Column Names
-------- -------- -------------------- ----------------
02/06/17 13:12:45                   17 YrsExp
```

For More Information

For information about collecting and dropping QCD statistics, see the following:

- “COLLECT STATISTICS (QCD Form)” on page 3-27
- “DROP STATISTICS (QCD Form)” on page 3-37

1 The report does not reflect sample size. Sample size is mentioned only to ensure that the comparison between the reports is accurate. A significantly smaller sample size could produce somewhat different statistics.

HELP TRIGGER

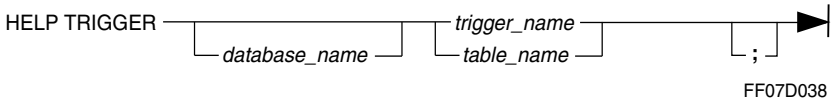
Purpose

Displays the attributes for the specified trigger.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies...
database_name	the optional name of the home database for the trigger.
trigger_name	the name of the trigger for which information is being requested.
table_name	that all triggers for the named table are to be displayed.

ANSI Compliance

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

Authorization

You must either own the specified trigger or have at least one privilege on it.

Trigger Attributes

Attribute	Description
DATABASE NAME	Name of the database in which the trigger is defined.
TABLE NAME	Name of the table on which a trigger is defined.
TRIGGER NAME	Name of the trigger.

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>TRIGGER ACTION TIME</td><td>Action time for the trigger.
This defines when the trigger fires.
<table><tr><th>Code</th><th>Description</th></tr><tr><td>A</td><td>After the trigger event.</td></tr><tr><td>B</td><td>Before the trigger event.</td></tr><tr><td>I</td><td>Instead of the trigger event.</td></tr></table></td></tr>
<tr><td>TRIGGER EVENT</td><td>Type of SQL DML statement that causes the trigger to fire.
<table><tr><th>Code</th><th>Description</th></tr><tr><td>D</td><td>Delete</td></tr><tr><td>I</td><td>Insert</td></tr><tr><td>U</td><td>Update</td></tr></table></td></tr>
<tr><td>TRIGGER KIND</td><td>the trigger type as defined by the FOR EACH clause.
<table><tr><th>Code</th><th>Description</th></tr><tr><td>R</td><td>Row</td></tr><tr><td>S</td><td>Statement</td></tr></table></td></tr>
<tr><td>DECIMAL ORDER VALUE</td><td>Default value used to specify trigger firing order when multiple triggers defined on the same table have the same action time, event, and FOR EACH clause.</td></tr>
<tr><td>ENABLED/DISABLED</td><td>Whether the trigger is activated or not.
<table><tr><th>Code</th><th>Description</th></tr><tr><td>Y</td><td>Enabled</td></tr><tr><td>N</td><td>Disabled</td></tr></table></td></tr>
<tr><td>COMMENT</td><td>Optional text commenting on the named trigger.</td></tr>
</table>

Example

The following example reports the information for a trigger that updates a table key.

```
HELP TRIGGER UpdateForKey;
```

```
                    Name       UpdateForKey
              ActionTime       B
                   Event       U
                    Kind       R
      Decimal Order Value       32, 767
                 Enabled       Y
                 Comment       ?
```

HELP VOLATILE TABLE

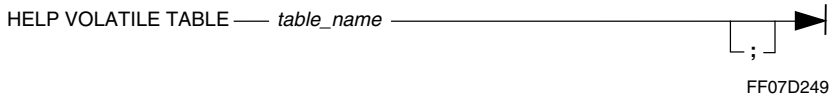
Purpose

Displays the attributes for the requested volatile table.

Invocation

Executable.

Syntax



where:

Syntax Element ...	Specifies ...
table_name	Name of the volatile table.

ANSI Compliance

HELP VOLATILE TABLE is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must either own the volatile table or have at least one privilege on it.

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 HASH INDEX SHOW JOIN INDEX SHOW MACRO SHOW PROCEDURE SHOW TABLE SHOW TRIGGER SHOW VIEW" on page 4-79).

- Protection
- Commit Option
- Transaction Log Option

Attribute	Data Type	Nullable?
TABLE NAME	CHARACTER(30)	no
TABLE ID	CHARACTER(12)	no

Example

Suppose you need to know the Table ID of the volatile table Sales_temp. To determine this information, you enter the following statement.

```
HELP VOLATILE TABLE Sales_temp;
```

The result shows that the Table ID for this table is `40C00E000000`.

```
Table Name       Sales_temp

  Table Id       40C00E000000
```

HELP (Online Form)

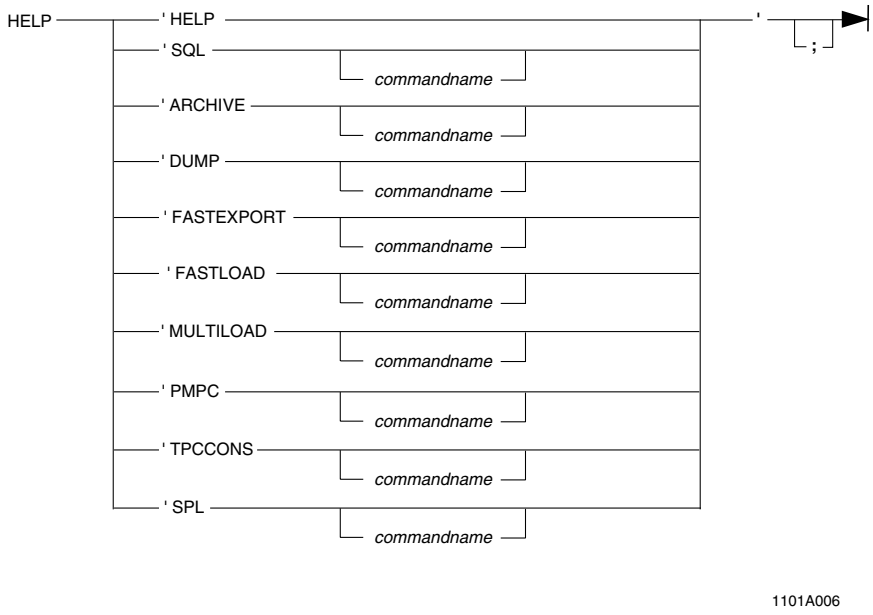
Purpose

Displays help on any SQL or client utility syntax topic.

Invocation

Executable.

Syntax



where:

Syntax Element ...	Specifies ...
HELP	HELP 'HELP' displays information about using the online help statement.
SQL Archive Dump FastExport FastLoad MultiLoad PMPC TPCCONS SPL	SQL or one of the client utilities. These are the topics about which HELP is requested. If these topics are not followed by a specific statement or utility name, a list of commands is displayed for that topic.
command_name	An optional specific SQL statement or utility command name.

ANSI Compliance

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

Authorization

None required.

Online Help Distinct From SQL HELP Statements

The online HELP statement facility is distinct from the SQL HELP statements, which provide information about various attributes of the database.

See also "HELP" on page 4-3.

Notation

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.

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]
```

Topics

Teradata provides online help for selected topical areas in addition to individual SQL statements. The online HELP topic keywords are:

- Archive
- Dump
- FastExport
- FastLoad
- Help
- MultiLoad
- PMPC
- SPL (stored procedures)
- SQL
- TPCCONS

HELP ‘topic with optional CommandName’ can be typed in upper or lower case.

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 is comprised of 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 in order to view the whole HELP message.

Examples

This statement...	Displays...
HELP ‘sql’;	a list of SQL statements.
HELP ‘multiload’;	a list of MultiLoad commands.
HELP ‘sql update’;	the syntax for the UPDATE statement.
HELP ‘spl’;	a list of all stored procedure SQL DDL and control statements.
HELP ‘spl if’;	the syntax, semantics, and usage of the stored procedure IF control statement.

SHOW

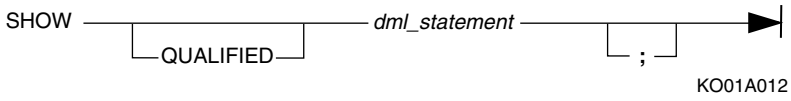
Purpose

Displays the DDL for all database objects referenced by a specified DML statement.

Invocation

Executable.

Syntax



where:

Syntax Element...	Specifies...
QUALIFIED	that if any view DDL is returned, it is to be qualified by the appropriate database and table names. This qualified text is taken from the DBC.TVM.CreateText table. All qualified objects in the report are enclosed within QUOTATION MARK characters. If you do not specify QUALIFIED, then any returned view DDL is taken from the DBC.TVM.RequestText table and is *not* qualified by the names of the appropriate database and table names.
dml_statement	an SQL DML statement for which the full text of the DDL for all the database objects it references is to be returned.

ANSI Compliance

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

Authorization

You must have either of the following privileges:

- Necessary privileges to perform a DML statement
- SELECT on dictionary table TVM

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. Views
3. Macros
4. Triggers
5. Join indexes

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 statement.
- The modifiers LOCKING and USING cannot precede the statement.
- *DML_statement* is limited to the following types of SQL DML statements:
  - DELETE
  - INSERT
  - INSERT/SELECT
  - 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.

Examples

These examples show the difference between requesting an unqualified SHOW DML report and a qualified SHOW DML report.

Example 1

The following example reports the DDL definitions supporting a simple SELECT statement involving one view and one table. Both objects were created in database DAFDB, so qualification is unnecessary.

```
SHOW
SELECT * FROM View_1, Table_2
WHERE View1.column_1=Table2.column_3;
```

The report looks like this.

```
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
------------------------------------------------------------
CREATE SET TABLE DAFDB.Table_2
  NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
   (
    column_1 INTEGER,
    column_2 INTEGER
   )
 PRIMARY INDEX(column_1);

CREATE SET TABLE DAFDB.Table_1
  NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
   (
    column_1 INTEGER
   )
 PRIMARY INDEX (column_1);

CREATE VIEW View_1 AS
  SELECT column_1 FROM Table_1;
```

Example 2

Consider the following scenario: you create a view using two other views, each having the same name, but created in different databases.

The database names are US and International. The session is in Teradata mode.

Here is the DDL used to create the objects used for this example:

```
DATABASE us;

CREATE TABLE table_1
 (eno INTEGER);

CREATE VIEW view_1 AS
 SELECT *
 FROM table_1;


DATABASE international;

CREATE TABLE table_1
 (zno INTEGER);

CREATE VIEW view_1 AS
 SELECT *
 FROM table_1;

CREATE VIEW view_3 AS
 SELECT *
 FROM international.view_1, us.view_1;
```

If you perform a SHOW statement on a query that selects data from view_3, the report it returns will be ambiguous with respect to the two view_1 elements of view_3 unless you specify the QUALIFIED option.

For example, suppose you perform SHOW on the following query.

```
SHOW SELECT * FROM view_3;
```

The report looks like this.

```
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
------------------------------------------------------------
CREATE SET TABLE international.table_1, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL
(
zno INTEGER)
PRIMARY INDEX (a);
```

```
   *** Text of DDL statement returned.
   ------------------------------------------------------------
   CREATE SET TABLE us.table_1, NO FALLBACK,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL
   (
   eno INTEGER)
   PRIMARY INDEX (eno);


   *** Text of DDL statement returned.
   ------------------------------------------------------------
   CREATE VIEW view_1 AS SELECT * FROM table_1;


   *** Text of DDL statement returned.
   ------------------------------------------------------------
   CREATE VIEW view_1 AS SELECT * FROM table_1;


   *** Text of DDL statement returned.
------------------------------------------------------------
   CREATE VIEW view_3 AS SELECT * FROM international.view_1,us.view_1;
```

Notice that none of the views is qualified by the name of its containing database.

Example 3

Use the QUALIFIED option to produce a report that resolves the ambiguous dependencies.

Consider the same scenario used in "Example 2" on page 4-76, but this time specify that the report must be qualified.

```
SHOW QUALIFIED SELECT * FROM view_3;
```

The report looks like this.

```
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
------------------------------------------------------------
CREATE SET TABLE international.table_1, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL
(
zno INTEGER)
PRIMARY INDEX (a);
```

```
   *** Text of DDL statement returned.
   ------------------------------------------------------------
   CREATE SET TABLE us.table_1, NO FALLBACK,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL
   (
   eno INTEGER)
   PRIMARY INDEX (eno);


   *** Text of DDL statement returned.
   ------------------------------------------------------------
   CREATE VIEW "international"."view_1" AS SELECT * FROM
   "international"."table_1";


   *** Text of DDL statement returned.
   ------------------------------------------------------------
   CREATE VIEW "us"."view_1" AS SELECT * FROM "us"."table_1";


   *** Text of DDL statement returned.
------------------------------------------------------------

   CREATE VIEW "international"."view_3" AS SELECT * FROM
   "international"."view_1"."zno", "us"."view_1"."eno";
```

Each of the view definitions in this report is fully qualified, removing the ambiguity seen in “Example 2” on page 4-76.

SHOW HASH INDEX
SHOW JOIN INDEX
SHOW MACRO
SHOW PROCEDURE
SHOW TABLE
SHOW TRIGGER
SHOW VIEW

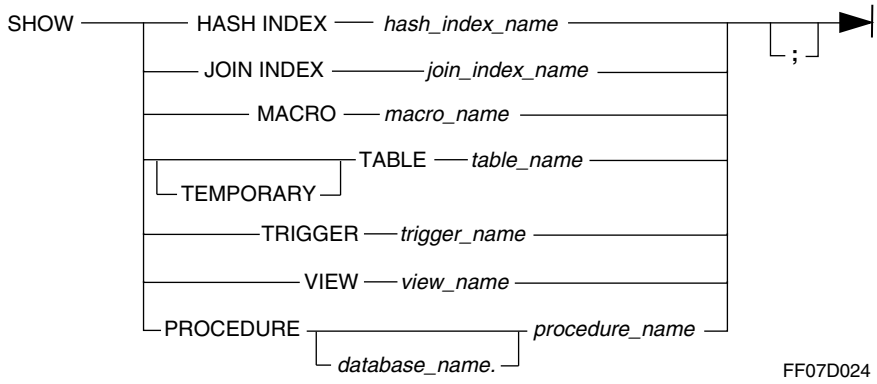
Purpose

Displays the data definition statement most recently used to create, replace, or modify the specified hash index, join index, macro, stored procedure, table, trigger, or view.

Invocation

Executable.

General Syntax

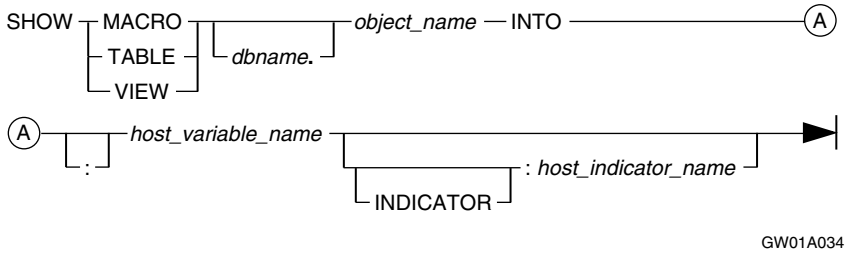


where:

Syntax Element...	Specifies...
hash_index_name	the name of the hash index whose most recent DDL statement is to be shown.
join_index_name	the name of the join index whose most recent DDL statement is to be shown.
macro_name	the name of the macro whose most recent DDL statement is to be shown. There is an upper limit of 12500 characters that SHOW MACRO can display.

Syntax Element...	Specifies...
TEMPORARY	that the table for which the most recent DDL statement is to be shown is a global temporary table.
table_name	the name of the table whose most recent DDL statement is to be shown. SHOW displays a standardized CREATE TABLE statement that would create the named table. If the table has been modified using the ALTER TABLE or CREATE INDEX statements, modifications are reflected in the CREATE TABLE statement displayed. SHOW TABLE displays all index and constraint information for the table. There is an upper limit of 31744 characters that SHOW TABLE can display. Because a standard display form is used, the result might not be identical to the text used to create the table. Its form is such that if the table were dropped and created using the SHOW TABLE output, the result would be a table with structure identical to the one shown.
trigger_name	the name of the trigger whose most recent DDL statement is to be shown. There is an upper limit of 12500 characters that SHOW TRIGGER can display.
view_name	the name of the view whose most recent DDL statement is to be shown. There is an upper limit of 12500 characters that SHOW VIEW can display.
database_name	the name of the database in which the stored procedure is defined.
procedure_name	the name of the stored procedure whose source text is to be shown. SHOW PROCEDURE displays stored procedure text as it was sent to the database. You must type line breaks in the procedure using the Enter or Return key to properly display the text.

Embedded SQL Syntax



where:

Syntax element...	Is the name of...
database_name	the database in which the macro, table or view is to be found.
object_name	the hash index, join index, macro, table or view definition to be viewed.
host_variable_name	a host variable conforming to VARCHAR(20480), into which the DDL statement is stored by Teradata SQL. The preceding colon is optional.
host_indicator_name	an indicator variable.

ANSI Compliance

SHOW HASH INDEX/JOIN INDEX/MACRO/PROCEDURE/TABLE/TRIGGER/VIEW are Teradata extensions to the ANSI SQL-99 standard.

Authorization

To use SHOW HASH INDEX, JOIN INDEX, MACRO, PROCEDURE, TABLE, or VIEW, a user must either have:

- a privilege on the hash index, join index, macro, stored procedure, table, or view, or a privilege on the database containing it
- SELECT privilege on table RDBMS.TVM (used by database administrators only)

DDL Security and SHOW

Because even database administrators cannot see or alter contents of objects for which they do not have rights, data is always secure.

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 MULTISET (duplicate rows if no uniqueness defined).
- SHOW TABLE displays fallback and journal status of the table.
- Named constraints become either table level constraints or are mapped to unique indexes. The display reflects the implementation.

  For example, the case shown in "Example 3: Column Level Named Constraints" on page 4-84, the named constraints on field_1 and field_2 are shown as table level constraints in the display, after the list of fields.

The named constraints on field_3 and field_4 become unique indexes. Since a PRIMARY KEY and no PRIMARY INDEX was declared in the CREATE TABLE, the PRIMARY KEY is implemented in the form of a UNIQUE PRIMARY INDEX, and is so listed. The UNIQUE constraint is implemented in the form of a UNIQUE INDEX and is so listed.

- 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.

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, the SHOW TABLE command results in the currently defined data block size or percent freespace (or both) being listed in the response.

If the data block size or freespace value is not listed, the default value is in effect. See the description of "ALTER TABLE" on page 1-12 or "CREATE TABLE" on page 1-182.

Support for Global Temporary and Volatile Tables

When you specify the keyword TEMPORARY, the table definition for the local materialized table in the current session is displayed; else the definition for the base temporary table is displayed.

The keyword VOLATILE is reported when the requested table is a volatile table.

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 network-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 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, Teradata does not return an error message.

Examples

The following examples illustrate the use of SHOW HASH INDEX, JOIN INDEX, MACRO, TABLE, or VIEW.

Example 1

The following statement displays the most recently executed definition of the NewEmp macro:

```
SHOW MACRO NewEmp;
```

The Teradata RDBMS displays:

```
REPLACE MACRO NewEmp( name CHAR(20) NOT NULL,
              street CHAR(30),
              city CHAR(20),
              number INTEGER,
              dept SMALLINT DEFAULT 999)
AS (INSERT INTO Employee (Name, Street, City,
              Empno, DeptNo)
      VALUES (:name, :street, :city, :number, :dept);
     UPDATE Department
      SET EmpCount = EmpCount + 1
     WHERE  DeptNo = :dept;);
```

Example 2

If a SHOW TABLE statement is entered for a table that has been modified, the original CREATE TABLE statement, including all current modifications, is displayed.

For example, consider the following statements used to create and subsequently modify the EmpBonus table:

```
CREATE TABLE Personnel.EmpBonus
 (EmpNo
   SMALLINT
   FORMAT '9(5)'
   CHECK (EmpNo BETWEEN 10001 AND 32001)
   NOT NULL,
  BonusNo
   SMALLINT
   FORMAT 'Z9'
   CHECK (BonusNo BETWEEN 0 and 99)
   NOT NULL,
  BonusAmt
   DECIMAL (6,2)
   CHECK (BonusAmt BETWEEN 1.00 AND 5000.00)
  )
 UNIQUE PRIMARY INDEX (EmpNo);
```

```
ALTER TABLE EmpBonus
   ADD DeptNo SMALLINT;
```

When you perform SHOW TABLE for the EmpBonus table, for example,

```
SHOW TABLE EmpBonus;
```

the following CREATE TABLE statement is returned:

```
CREATE SET TABLE Personnel.EmpBonus, FALLBACK
 (EmpNo SMALLINT FORMAT '9(5)' CHECK (EmpNo BETWEEN 10001
   AND 32001)NOT NULL,
 BonusNo SMALLINT FORMAT 'Z9' CHECK (BonusNo BETWEEN 0 AND 99)
NOT NULL,
 BonusAmt DECIMAL (6,2) CHECK (BonusAmt BETWEEN 1.00 AND
5000.00),
 DeptNo SMALLINT)
 UNIQUE PRIMARY INDEX (EmpNo);
```

The DeptNo column is included in the CREATE TABLE statement. In addition, default values are displayed (for example, FALLBACK).

Example 3: Column Level Named Constraints

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

```
CREATE TABLE good_1
( field_1 INTEGER NOT NULL
    CONSTRAINT primary_1 PRIMARY KEY,
  field_2 INTEGER NOT NULL
    CONSTRAINT unique_1 UNIQUE,
  field_3 INTEGER
    CONSTRAINT check_1
    CHECK (field_3 > 0),
  field_4 INTEGER
    CONSTRAINT reference_1
    REFERENCES parent_1);
```

Now perform the following SHOW TABLE statement on good_1.

```
SHOW TABLE good_1;
```

The statement produces the following DDL report:

```
CREATE SET TABLE Personnel.good_1, NO FALLBACK,
   NO BEFORE JOURNAL
   NO AFTER JOURNAL
   (
   field_1 INTEGER NOT NULL,
   field_2 INTEGER NOT NULL,
   field_3 INTEGER,
   field_4 INTEGER,
   CONSTRAINT check_1 CHECK (field_3 >0),
   CONSTRAINT reference_1 FOREIGN KEY (field_4)
      REFERENCES Personnel.PARENT_1 (f2))
UNIQUE PRIMARY INDEX primary_1 (field_1)
UNIQUE INDEX unique_1 (field_2);
```

Example 4: REFERENCES Constraint

Table T1 is created with a referential constraint which specifies that Foreign Key columns (f1) reference the unique primary index columns in table Good_1.

```
CREATE TABLE T1
(f1 integer NOT NULL
f2 integer)
PRIMARY INDEX (f1)
FOREIGN KEY (f1) REFERENCES Good_1;
SHOW TABLE T1;
```

results in display:

```
CREATE SET TABLE T1, NO FALLBACK,
   NO BEFORE JOURNAL, NO AFTER JOURNAL
(f1 integer NOT NULL,
f2 integer)
PRIMARY INDEX (f1)
FOREIGN KEY (f1) REFERENCES Personnel.Good_1(field_1);
```

Example 5: SHOW VIEW

The following SHOW VIEW example begins with the creation of the view, then progresses to the SHOW VIEW statement and finally presents the result of the report request.

```
CREATE VIEW staff_info
 (number, name, position, department, sex, dob) AS
SELECT employee.empno, name, jobtitle, deptno, sex, dob
FROM Employee
WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
WITH CHECK OPTION;
```

When the following SHOW VIEW statement is entered, the immediately following view definition is returned.

```
SHOW VIEW staff_info;

CREATE VIEW staff_info
 (number, name, position, department, sex, dob) AS
SELECT employee.empno, name, jobtitle, deptno, sex, dob
FROM Employee
WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
WITH CHECK OPTION;
```

Example 6: SHOW TABLE

Consider a user whose default character type is Latin. The user creates the following table.

```
CREATE TABLE kanji_user.table_1,
  NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (
   clatin CHARACTER(5),
   ckanji1 CHARACTER(5) CHARACTER SET KANJI1,
   cgraphic CHARACTER(5) CHARACTER SET GRAPHIC,
   ckanjisjis CHARACTER(5) CHARACTER SET KANJISJIS,
   cunicode CHARACTER(5))
  PRIMARY INDEX (clatin);
```

A SHOW TABLE statement against this table produces the following report, as expected.

```
CREATE TABLE kanji_user.table_1,
  NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (
   clatin CHARACTER(5) CHARACTER SET LATIN,
   ckanji1 CHARACTER(5) CHARACTER SET KANJI1,
   cgraphic CHARACTER(5) CHARACTER SET GRAPHIC,
   ckanjisjis CHARACTER(5) CHARACTER SET KANJISJIS,
   cunicode CHARACTER(5) CHARACTER SET LATIN)
  PRIMARY INDEX (clatin);
```

Example 7: SHOW JOIN INDEX

The following example illustrates a SHOW JOIN INDEX statement run against the join index named OrdCustIdx.

```
SHOW JOIN INDEX OrdCustIdx;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.

--------------------------------------------------
CREATE JOIN INDEX LOUISB.OrdCustIdx AS
SELECT LOUISB.Customer.c_custkey,
LOUISB.Customer.c_name, LOUISB.Orders.o_status,
LOUISB.Orders.o_date, LOUISB.Orders.o_comment
FROM LOUISB.Orders, LOUISB.Customer
WHERE LOUISB.Orders.o_custkey =
LOUISB.Customer.c_custkey
PRIMARY INDEX (c_custkey)
INDEX (o_date);
```

Example 8: SHOW HASH INDEX

The following example illustrates a SHOW HASH INDEX statement run against the hash index named OrdHIdx created by the following CREATE HASH INDEX statement.

```
CREATE HASH INDEX OrdHIdx (o_orderdate) ON Orders;

SHOW HASH INDEX OrdHIdx;
 *** Text of DDL statement returned.
 *** Total elapsed time was 1 second.
------------------------------------------------------------
CREATE HASH INDEX PLS.OrdHIdx, NO FALLBACK PROTECTION
(PLS.Orders.o_ordertable) ON PLS.Orders;
```

The report differs from the text of the CREATE HASH INDEX DDL in several ways.

- All names are qualified
- The fallback protection option is displayed

Note that when you do not specify a BY or ORDER clause in the CREATE HASH INDEX DDL, the default values for those clauses are not reported.

Example 9: SHOW PROCEDURE

The following example shows the creation of a stored procedure `print1` and the output of the SHOW PROCEDURE statement:

```
CREATE PROCEDURE print1()
BEGIN DECLARE var1 INTEGER;
   SET var1 = 1;
   PRINT 'value of var1 =', var1;
END;
```

When you specify SHOW PROCEDURE for `print1` in Field mode, the following output containing the stored procedure definition returns to the requestor.

```
SHOW PROCEDURE print1

   SPLText
   ---------------
   CREATE PROCEDURE print1 ()
   BEGIN DECLARE var1 INTEGER;
      SET var1 = 1;
      PRINT 'value of var1 =', var1;
   END;
```

The stored procedure definition is returned as it was entered. The format of the procedure in the response is the same as specified by the creator when the stored procedure was created.

Example 10:

Appendix A:

Notation Conventions

This appendix describes the notation conventions used in this book.

Introduction

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

- Syntax diagrams, used to describe SQL syntax form, including options. See "Syntax Diagram Conventions" on page A-2.
- Square braces in the text, used to represent options. The indicated parentheses are required when you specify options.

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

  The values for *n* and *m* are integers in all cases.
- Japanese character code shorthand notation, used to represent unprintable Japanese characters. See "Character Shorthand Notation Used In This Book" on page A-6.

Symbols from the predicate calculus are also used occasionally to describe logical operations. See "Predicate Calculus Notation Used in This Book" on page A-8.

Syntax Diagram Conventions

Notation Conventions

The following table defines the notation used in this section:

<table>
<tr><th>Item</th><th>Definition / Comments</th></tr>
<tr><td>Letter</td><td>An uppercase or lowercase alphabetic character ranging from A through Z.</td></tr>
<tr><td>Number</td><td>A digit ranging from 0 through 9.

Do not use commas when entering a number with more than three digits.</td></tr>
<tr><td>Word</td><td>Variables and reserved words.

<table>
<tr><th>IF a word is shown in . . .</th><th>THEN it represents . . .</th></tr>
<tr><td>UPPERCASE LETTERS</td><td>a keyword.

Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase.

If a keyword is shown in uppercase, you may enter it in uppercase or mixed case.</td></tr>
<tr><td>lowercase letters</td><td>a keyword that you must enter in lowercase, such as a UNIX command.</td></tr>
<tr><td>lowercase italic letters</td><td>a variable such as a column or table name.

You must substitute a proper value.</td></tr>
<tr><td>lowercase bold letters</td><td>a variable that is defined immediately following the diagram that contains it.</td></tr>
<tr><td>UNDERLINED LETTERS</td><td>the default value.

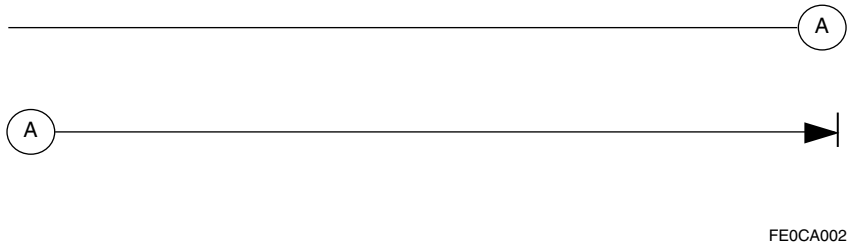
This applies both to uppercase and to lowercase words.</td></tr>
</table>
</td></tr>
<tr><td>Spaces</td><td>Use one space between items, such as keywords or variables.</td></tr>
<tr><td>Punctuation</td><td>Enter all punctuation exactly as it appears in the diagram.</td></tr>
</table>

Paths

The main path along the syntax diagram begins at the left, and proceeds, left to right, to the vertical bar, which marks the end of the diagram. Paths that do not have an arrow or a vertical bar only show portions of the syntax.

The only part of a path that reads from right to left is a loop.

Paths that are too long for one line use continuation links. Continuation links are small circles with letters indicating the beginning and end of a link:



When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue.

Required Items

Required items appear on the main path:

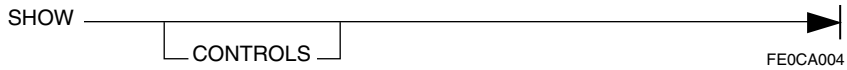


If you can choose from more than one item, the choices appear vertically, in a stack. The first item appears on the main path:

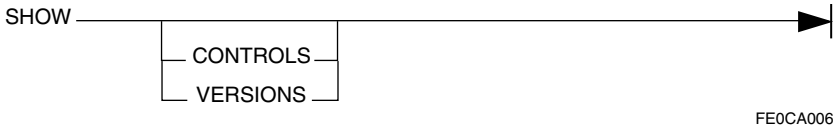


Optional Items

Optional items appear below the main path:



If choosing one of the items is optional, all the choices appear below the main path:



You can choose one of the options, or you can disregard all of the options.

Abbreviations

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

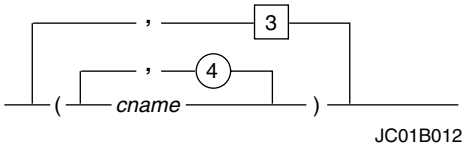


In the above syntax, the following formats are valid:

- SHOW CONTROLS
- SHOW CONTROL

Loops

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



The following rules apply to loops:

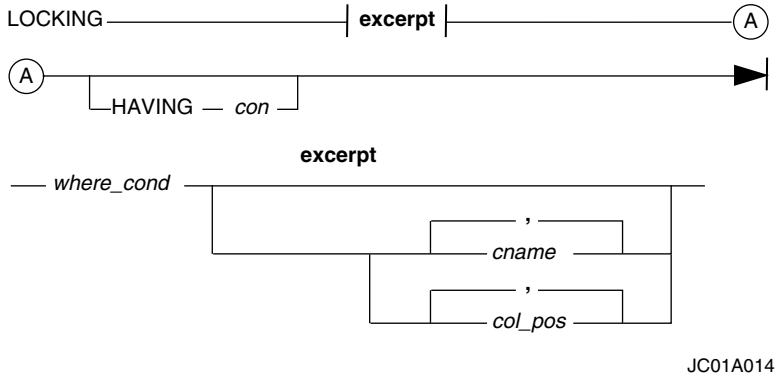
IF ...	THEN ...
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may enter *cname* a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must enter at least 3 groups of column names.
a separator character is required between entries	the character appears on the return path. If the diagram does not show a separator character, use one blank space. In the example, the separator character is a comma.

IF ...	THEN ...
a delimiter character is required around entries	the beginning and end characters appear outside the return path. Generally, a space is not needed between delimiter characters and entries. In the example, the delimiter characters are the left and right parentheses.

Excerpts

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

The named phrase appears immediately after the complete diagram, as illustrated by the following example.



Character Shorthand Notation Used In This Book

Introduction

This book uses the UNICODE naming convention for characters. For example, the lowercase character ‘a’ is more formally specified as either LATIN SMALL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the book, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings.

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in Appendix B: “Japanese Character Sets," in *Teradata RDBMS International Character Set Support*.

Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a..z A..Z 0..9	Any	Any single byte Latin letter or digit.
a..z A..Z 0..9	Unicode compatibility zone	Any full-width Latin letter or digit.
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Used to indicate transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Used to indicate transition from multibyte to single byte KanjiEBCDIC.

Symbol	Encoding	Meaning
T	Any	Any multibyte character. Its encoding depends on the current character set. For KanjiEUC, code set 3 characters are sometimes shown preceded by "ss_3".
I	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by "ss_2", forming an individual multibyte character.
Δ	Any	Represents the graphic pad character.
Δ	Any	Represents either a single or multibyte pad character, depending on context.
ss_2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss_3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string "TEST", where each letter is intended to be a full-width character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set

```
LMN<TEST>QRS
```

is represented as:

```
D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2
```

Pad Characters

The following table lists the pad characters for the various character data types.

Character Data Type	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20

Predicate Calculus Notation Used in This Book

Relational databases are based on the theory of relations as developed in set theory. Predicate calculus is often the most unambiguous way to express certain relational concepts.

Occasionally this book uses the following predicate calculus notation to explain concepts.

This symbol ...	Represents this phrase ...
iff	If and only if
∀	For all
∃	There exists

Index

Symbols

*
  CREATE VIEW option 1–313
  used in macros 1–147

Numerics

2PC
  DATABASE statement not used 1–326
  disallow SET SESSION COLLATION statement 1–391

A

Access privileges 2–2
  automatic 2–5, 2–6
  creators 2–4
  explicit 2–5, 2–6
  granting to others 2–8
  implicit 2–5, 2–6
  inherited 2–6, 2–7
  owners 2–4
  system-granted 2–8
  types 2–5
  users 2–4
Access rights 2–2
  GRANT statement 2–22
  INDEX 2–30
  REFERENCES 2–29
ACCOUNT, SET SESSION option 1–387
Acctid, specifying 1–305
ALTER PROCEDURE statement 1–3
ALTER TABLE statement 1–12
  locks and concurrency 1–24
  modifying a primary index 1–43
  not valid with volatile tables 1–24
  revalidating table headers and partitioning 1–49
ALTER TRIGGER statement 1–65
AnalysisLog table 3–62
ANSIDATE dateform
  session 1–398
  user 1–302, 1–370
Attributes
  changing with ALTER TABLE 1–37
  column, changing 1–29
Authorizations 2–2
Automatic access privileges 2–5, 2–6

B

BEGIN LOGGING statement 1–67
  rules for using 1–74
BEGIN QUERY LOGGING statement 3–4
Between constraints 1–238
BETWEEN, supported in CREATE TABLE statement 1–238

C

Character data type, expansion rules 1–320
CHARSET_COLL 1–393
CHECK constraint rules 1–237
Check constraints 1–237
Collation
  CREATE USER option 1–305
  defining 1–392
  MODIFY USER option 1–375
COLLECT DEMOGRAPHICS statement 3–11
COLLECT STATISTICS (Optimizer Form)
  statement 3–14
  collecting statistics on a partitioned primary index 3–21
  collecting statistics on multiple non-indexed columns 3–22
  creating from CREATE INDEX statement 3–16
  reducing the cost of collecting statistics by using sampling 3–19
  rules for using 3–20
COLLECT STATISTICS (QCD Form) statement 3–27
Column attributes, changing 1–29
Column character type, default 1–200
Column level rights
  granting 2–30

Columns
adding with ALTER TABLE 1–26
defined as primary key or unique nulls 1–247
dropping with ALTER TABLE 1–25
indexed, dropping 1–25
nullable, as primary key/index 1–247
COMMENT statement 1–76
COMPRESS
ALTER TABLE option 1–30
CREATE TABLE option 1–191
Concurrency, with CREATE INDEX 1–103
Constraint names 1–224
Constraints
between 1–238
check 1–237
primary key 1–225
REFERENCES 1–226
unique 1–225
CREATE DATABASE statement 1–80
CREATE HASH INDEX statement 1–86
CREATE INDEX statement 1–100
CREATE JOIN INDEX statement 1–108
CREATE MACRO statement 1–142
rules for using parameters 1–148
CREATE PROCEDURE statement 1–154
CREATE PROFILE statement 1–173
CREATE ROLE statement 1–180
CREATE TABLE statement 1–182
AS clause 1–249
column definition clause 1–212
copy table syntax 1–184
examples 1–257
index definition clause 1–239
table kind clause 1–201
table options clause 1–208
temporary/volatile table preservation clause 1–248
CREATE TRIGGER statement 1–276
privileges 1–280
CREATE USER statement 1–296
maxima 1–305
CREATE VIEW statement 1–311, 1–316
including data definitions 1–319
rules 1–317
Creator
definition for access privileges 2–4
Current role
setting 1–382

D

Data control language statements. See DCL statements
Data definition language statements. See DDL statements
Data definitions, included in CREATE VIEW 1–319
Data type
changing with ALTER TABLE 1–37
procedure for changing 1–39
Database
compared with user 2–3
definition 2–3
DATABASE statement 1–325
DATABLOCKSIZE, ALTER TABLE option 1–27
DATEFORM
ANSIDATE 1–302, 1–370, 1–398
INTEGERDATE 1–302, 1–370, 1–398
session 1–398
user 1–302, 1–370
DCL statements, defined 2–1
DDL statements, defined 1–1
DEFAULT DATABASE, MODIFY USER option 1–375
Defaults, changing 2–70
DELETE DATABASE statement, SQL 1–327
DELETE USER statement 1–327
Derived table
described 1–314
DIAGNOSTIC "Validate Index" statement 3–83
DIAGNOSTIC DUMP COSTS statement 3–66
authorization requirements 3–66
using to extract cost information 3–67
DIAGNOSTIC DUMP SAMPLES statement 3–76
authorization requirements 3–77
DIAGNOSTIC HELP COSTS statement 3–68
authorization requirements 3–68
using to extract cost information 3–68
DIAGNOSTIC HELP SAMPLES statement 3–78
authorization requirements 3–78
DIAGNOSTIC SET COSTS statement 3–71
authorization requirements 3–72
performance precedence 3–73
scope level precedence 3–73
scope levels 3–72
using to extract cost information 3–71
DIAGNOSTIC SET SAMPLES statement 3–80
authorization requirements 3–81
scope level precedence 3–81
DROP DATABASE statement 1–330
privileges 1–330, 1–345
DROP HASH INDEX statement 1–332

DROP INDEX statement 1–333
DROP JOIN INDEX statement 1–337
DROP MACRO statement 1–338
  privileges 1–340
DROP PROCEDURE statement 1–338
DROP PROFILE statement 1–341
DROP ROLE statement 1–343
DROP STATISTICS (QCD Form) statement 3–37
DROP STATISTICS statement 3–31
  named indexes, rules 3–35
DROP TABLE statement 1–338
  privileges 1–340
DROP TRIGGER statement
  privileges 1–340
DROP USER statement 1–345
  privileges 1–330, 1–345
DROP VIEW statement 1–338
  privileges 1–340
DUMP EXPLAIN statement 3–40
Duplicate rows
  MULTISET table kind option 1–201
  SET table kind option 1–201

E

END LOGGING statement 1–347
  permissions required 1–350
END QUERY LOGGING statement 3–44
Error table, created by ALTER TABLE 1–36
Explicit access privileges 2–5, 2–6

F

FALLBACK
  ALTER TABLE option, overriding 1–24
  ALTER TABLE statement and 1–24
  option described 1–208
FOR EACH ROW, rules 1–285
FOR EACH STATEMENT, rules 1–286
Foreign key
  adding with ALTER TABLE 1–36
  applicable rules 1–226
  maintaining integrity of 1–36
FREESPACE, description 1–211

G

GIVE statement 2–11
  compared to GRANT statement 2–29
GRANT LOGON statement 2–43
GRANT MONITOR statement 2–15
  cautions WITH GRANT option 2–18
GRANT statement 2–14, 2–22
  compared to GIVE 2–29
  granting role membership 2–19
  options 2–28
Granting access privileges to others 2–8

H

Hash index
  restrictions
    Archive and Restore 1–95
    ROLLBACK/ROLLFORWARD 1–95
    triggers 1–95
HAVING
  CREATE VIEW option 1–316
HELP (OnLine) statement 4–70
HELP COLUMN statement 4–7, 4–52
HELP CONSTRAINT statement 4–22
HELP DATABASE statement 4–26
HELP HASH INDEX statement 4–29
HELP INDEX statement 4–31
HELP JOIN INDEX statement 4–38
HELP MACRO statement 4–40
HELP PROCEDURE statement 4–46
HELP statement 4–3
HELP STATISTICS (Optimizer Form) statement 4–60
HELP STATISTICS (QCD Form) statement 4–63
HELP TABLE statement 4–40
HELP TRIGGER statement 4–65
HELP USER statement 4–26
HELP VIEW statement 4–40
HELP VOLATILE statement 4–68

I

Identity column
  surrogate keys 1–212
Implicit access privileges 2–5, 2–6
Inconsistent table, and referential integrity 1–37
INDEX
  access rights 2–30
  privilege 2–30

Index Wizard utility 3–11, 3–46, 3–55, 3–56, 3–57, 3–58, 3–59, 3–62, 3–83
INDEX, CREATE TABLE option
compared to unique primary key 1–226
Index. See primary index, secondary index, hash index, join index
Indexed columns, dropping 1–25
Inherited access privileges 2–6, 2–7
INITIATE INDEX ANALYSIS statement 3–46
INSERT EXPLAIN statement 3–12
INTEGERDATE dateform
session 1–398
user 1–302, 1–370

J

Japanese character code notation, how to read A–6
Japanese characters, in comments 1–78
Join index
hash distribution of rows 1–129
not supported with triggers 1–288
procedure for defining 1–130
restrictions
Archive and Restore 1–130
number of selected per table 1–118
ROLLBACK/ROLLFORWARD 1–131
triggers 1–131
sparse 1–135, 1–136
JOURNAL
ALTER TABLE option, overriding 1–24
option described 1–208
Journals
ALTER TABLE options 1–28
LOCAL 1–83
not deleted with DELETE DATABASE/USER 1–328

K

Keys
primary 1–225
surrogate 1–225

L

LOCKING FOR READ, CREATE VIEW option, overriding 1–319
LOCKING, CREATE VIEW option 1–319
Locks
CREATE TABLE statement 1–200
in view definition 1–319
MODIFY USER 1–376
overriding 1–319
with CREATE INDEX 1–103
with DELETE DATABASE/USER 1–328
Logon, revoking permission 2–70

M

Macros
tactical queries 1–148
MODIFY DATABASE statement 1–352
privileges 1–355
Modify primary index clause, ALTER TABLE statement 1–41
MODIFY PROFILE statement 1–359
MODIFY USER statement 1–364
locks 1–376
privileges 1–371
rules 1–372
MultiLoad
restrictions with hash index 1–94
restrictions with join index 1–130
MULTINATIONAL
CREATE USER option 1–305
described 1–392
MULTISET
rules for assigning 1–201
table kind option 1–201

N

Names, constraint 1–224
NOT NULL, column attribute 1–29
NULL, column attribute 1–29

O

Online help 4–70
Operators, in views 1–319
Optimizer
reasons to collect statistics 3–18
Owner
definition for access privileges 2–4

P

Partitioned primary index 1–46
  ALTER TABLE statement 1–21
  Archive/Restore restrictions 1–245
  description 1–240
  FastLoad restrictions 1–245
  MODIFY PRIMARY INDEX option 1–21
  MultiLoad restrictions 1–245
  restrictions 1–242
Password, requirements for (GRANT LOGON) 2–45
Primary index
  described 1–240
  nonpartitioned 1–240
  partitioned 1–240
  procedure for redefining without using ALTER TABLE 1–41
  rules for 1–246
  rules for changing partitioning expression 1–46
  rules for partitioning expressions 1–240
  rules for redefining with modify primary index option 1–43
Primary key 1–225, 1–246
Privileges 2–2
  column-level 2–23, 2–29
  CREATE DATABASE 1–83
  CREATE TABLE statement 1–199
  CREATE TRIGGER 1–280
  CREATE VIEW statement 1–316
  DROP DATABASE 1–330, 1–345
  DROP INDEX 1–335
  DROP MACRO/TABLE/TRIGGER/VIEW 1–340
  DROP USER 1–330, 1–345
  duration of 2–67
  establishing, with GRANT statement 2–14
  GIVE statement 2–11
  GRANT statement 2–22
  INDEX 2–30
  logon, revoking 2–70
  MODIFY DATABASE 1–355
  MODIFY USER option 1–371
  MONITOR-related, GRANT MONITOR statement 2–15
  non-revokable 2–52
  REFERENCES 2–29
  REPLACE TRIGGER 1–281
  revokable 2–52
  SHOW statements 4–73, 4–81
  when effective 2–29

Q

QCD database 3–11, 3–12, 3–13, 3–28, 3–38, 3–40, 3–41, 3–42, 3–58, 3–62
  AnalysisLog table 3–62
  COLLECT STATISTICS (QCD Form) 3–27
  DROP STATISTICS (QCD Form) 3–37
  TableStatistics table 3–28, 3–38

R

REFERENCES
  access rights 2–29
References constraint 1–226
REFERENCES, CREATE TABLE option
  applicable rules 1–227
REFERENCES, privilege 2–29
Referential integrity, adding or dropping constraint 1–34
RENAME MACRO statement 1–380
RENAME PROCEDURE statement 1–380
RENAME statement 1–380
RENAME TABLE statement 1–380
RENAME TRIGGER statement 1–380
RENAME VIEW statement 1–380
REPLACE MACRO statement 1–142
REPLACE PROCEDURE statement 1–154
REPLACE TRIGGER statement 1–276
  privileges 1–281
Reporting space used 1–306
RESTART INDEX ANALYSIS statement 3–62
Revalidate primary index clause, ALTER TABLE statement 1–49
REVOKE LOGON statement 2–70
REVOKE MONITOR statement 2–49
REVOKE statement 2–47, 2–57
  revoking role membership 2–54
Roles
  active 2–20
  active role 1–382
  creating 1–180
  current role 1–382
  dropping 1–343
  granting role membership 2–19
  revoking role membership 2–54
  setting 1–382
  usage 2–20
Row triggers 1–284

Rules
assigning SET/MULTISET 1–201
character data type expansion 1–320
CHECK constraint 1–237
constraint names 1–224
CREATE VIEW 1–317
FOR EACH ROW 1–285
FOR EACH STATEMENT 1–286
foreign key 1–226
FREESPACE 1–211
GRANT statement 2–28
macro expansion 1–149
MODIFY USER 1–372
named indexes, COLLECT/DROP STATISTICS 3–35
OUTER JOIN 1–114
parameters in macros 1–148
primary key 1–246
privilege statements, REVOKE statement 2–61
REFERENCES, CREATE TABLE option 1–227
time zone displacement 1–401

S

Secondary index, described 1–245
SET ROLE statement 1–382
SET SESSION ACCOUNT statement 1–387
SET SESSION COLLATION statement 1–391
SET SESSION DATABASE statement 1–397
SET SESSION DATEFORM statement 1–398
SET SESSION statement 1–384
SET table kind option 1–201
SET TIME ZONE statement 1–400
SET, rules for assigning 1–201
SHOW HASH INDEX statement 4–79
SHOW JOIN INDEX statement 4–79
SHOW MACRO statement 4–79
SHOW PROCEDURE statement 4–79
SHOW statement 4–73
SHOW TABLE statement 4–79
SHOW TRIGGER statement 4–79
SHOW VIEW statement 4–79
Single sign-on
creating a user 1–307, 1–373
granting logon to a user 2–46
Space, reporting space used 1–306
Sparse join index 1–135, 1–136
SQL development tools 4–1
SQL statements
ALTER PROCEDURE 1–3
ALTER TABLE 1–12
ALTER TRIGGER 1–65
BEGIN LOGGING 1–67
BEGIN QUERY LOGGING 3–4
COLLECT DEMOGRAPHICS 3–11
COLLECT STATISTICS (Optimizer Form) 3–14
COLLECT STATISTICS (QCD Form) 3–27
COMMENT 1–76
CREATE DATABASE 1–80
CREATE HASH INDEX 1–86
CREATE INDEX 1–100
CREATE JOIN INDEX 1–108
CREATE MACRO 1–142
CREATE PROCEDURE 1–154
CREATE TABLE 1–182
CREATE TRIGGER 1–276
CREATE USER 1–296
CREATE VIEW 1–311
DATABASE 1–325
DELETE DATABASE 1–327
DELETE USER 1–327
DIAGNOSTIC "Validate Index" 3–83
DIAGNOSTIC DUMP COSTS 3–66
DIAGNOSTIC DUMP SAMPLES 3–76
DIAGNOSTIC HELP COSTS 3–68
DIAGNOSTIC HELP SAMPLES 3–78
DIAGNOSTIC SET COSTS 3–71
DIAGNOSTIC SET SAMPLES 3–80
DROP DATABASE 1–330
DROP HASH INDEX 1–332
DROP INDEX 1–333
DROP JOIN INDEX 1–337
DROP MACRO 1–338
DROP PROCEDURE 1–338
DROP STATISTICS 3–31
DROP STATISTICS (QCD Form) 3–37
DROP TABLE 1–338
DROP USER 1–345
DROP VIEW 1–338
DUMP EXPLAIN 3–40
END LOGGING 1–347
END QUERY LOGGING 3–44
GIVE 2–11
GRANT 2–14, 2–22
GRANT LOGON 2–43
GRANT MONITOR 2–15
HELP 4–3
HELP (OnLine) 4–70
HELP COLUMN 4–7, 4–52
HELP CONSTRAINT 4–22
HELP DATABASE 4–26

HELP HASH INDEX 4–29
HELP INDEX 4–31
HELP JOIN INDEX 4–38
HELP MACRO 4–40
HELP PROCEDURE 4–46
HELP STATISTICS (Optimizer Form) 4–60
HELP STATISTICS (QCD Form) 4–63
HELP TABLE 4–40
HELP TRIGGER 4–65
HELP USER 4–26
HELP VIEW 4–40
HELP VOLATILE TABLE 4–68
INITIATE INDEX ANALYSIS 3–46
INSERT EXPLAIN 3–12
MODIFY DATABASE 1–352
MODIFY USER 1–364
RENAME 1–380
REPLACE MACRO 1–142
REPLACE PROCEDURE 1–154
REPLACE TRIGGER 1–276
RESTART INDEX ANALYSIS 3–62
REVOKE 2–47, 2–57
REVOKE LOGON 2–70
REVOKE MONITOR 2–49
SET SESSION 1–384
SET SESSION ACCOUNT 1–387
SET SESSION COLLATION 1–391
SET SESSION DATABASE 1–397
SET SESSION DATEFORM 1–398
SET TIME ZONE 1–400
SHOW 4–73
SHOW HASH INDEX 4–79
SHOW JOIN INDEX 4–79
SHOW MACRO 4–79
SHOW PROCEDURE 4–79
SHOW TABLE 4–79
SHOW TRIGGER 4–79
SHOW VIEW 4–79
Statement triggers 1–284
Stored procedures
ALTER PROCEDURE 1–3
CREATE PROCEDURE 1–154
replace 1–154
Surrogate key 1–225
identity columns 1–212
Syntax, how to read A–1
System-granted access privileges 2–8

T

Table
derived, description of 1–314
gobal 1–202
temporary, materializing 1–202
temporary, referenced in views 1–317
volatile 1–202
volatile, referenced in views 1–317
Table kind clause, CREATE TABLE statement 1–201
TableStatistics table 3–28, 3–38
Tactical queries
macros 1–148
triggers 1–289
Teradata Index Wizard utility 3–11, 3–46, 3–55, 3–57, 3–58, 3–59, 3–62, 3–83
Teradata Index wizard utility 3–56
Teradata Statistics Wizard 1–119
Teradata statistics wizard
optimizer support 3–18
QCD support 3–29
Time zone, SET TIME ZONE statement 1–400
Triggered events 1–281
Triggering actions 1–281
Triggers
not supported by load utilities 1–287
orphaned 1–329
restrictions with hash index 1–95
restrictions with join index 1–131
row and statement triggers described 1–284
tactical queries 1–289

U

Unicode, notation A–6
Unique constraint 1–225
UNIQUE PRIMARY KEY, CREATE TABLE option
compared to INDEX 1–226
Updatable cursor, not supported for triggers 1–288
User
compared with database 2–3
definition 2–3
definition for access privileges 2–4
UTC, Coordinated Universal Time 1–401

V

Volatile table, ALTER TABLE not valid with 1–24

W

WHERE
CREATE VIEW option 1–316
WITH CHECK OPTION
CREATE VIEW option 1–318