



Teradata[®] RDBMS

SQL Reference - Volume 1

Fundamentals

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, CICS/PLEX, CICS/VIEW, 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 © 1996-2003, 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
February 2003	Replaced restricted word list in Appendix B with a cross-reference to Appendix A in <i>Teradata RDBMS Release Summary</i> .
December 2002	• Moved material on the following topics to <i>Teradata RDBMS SQL Reference, Volume 6</i>: – Result code variables – Host variables – Multisession asynchronous programming • Added overview of partitioned primary index • Added overview of roles and profiles • Moved international character set support chapter and appendixes to a new book, <i>Teradata RDBMS International Character Set Support</i> • Added Appendix C, “Teradata RDBMS Maxima”
June 2001	• Added material on hash index • Added material on UTF8 support • Reorganized material on restricted words • Performed DR and RFC enhancements
September 2000	• Usability enhancements
June 2000	• General reorganization and rewriting of material to enhance usability • Revised and reorganized description of international character support • Addition and extensive rewriting and reorganization of embedded SQL material formerly found in <i>Teradata Application Programming with C, COBOL, and PL/I</i>. • Updated reserved word list • Updated ANSI-Teradata SQL similarities and differences list

About This Book

Purpose

Teradata RDBMS SQL Reference, Volume 1 describes basic SQL data handling, SQL data definition, control, and manipulation, and the SQL lexicon.

This preface describes the organization of *Teradata RDBMS SQL Reference, Volume 1* 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, security administrators, application programmers, NCR field engineers, end users, and other technical personnel responsible for designing, maintaining, and using the Teradata RDBMS will find this manual useful.

How This Book Is Organized

This book has three chapters and three appendixes.

- [Chapter 1: “SQL Data Handling Fundamentals”](#) describes the fundamentals of data manipulation, SQL requests, and other basic information about using the Teradata relational database management system.
- [Chapter 2: “SQL Data Definition, Control, and Manipulation”](#) is an overview of the families of SQL statements and binding types.
- [Chapter 3: “SQL Lexicon”](#) describes SQL words and object names, delimiters and separators, constants, and operators.
- [Appendix A: “Notation Conventions”](#) explains how to read syntax diagrams, character shorthand notation, and predicate calculus notation.
- [Appendix B: “Restricted Words for V2R5.0”](#) describes the following types of restricted SQL words for the V2R5.0 release:
 - Teradata reserved words, non-reserved words, and words reserved for future use
 - ANSI SQL-99 keywords
- [Appendix C: “Teradata RDBMS Maxima”](#) describes the limits on various system, database, and session capabilities.

Prerequisites

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

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

Experienced SQL users can also see simplified statement, data type, function, and expression descriptions in *Teradata RDBMS SQL/Data Dictionary Quick Reference*.

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	Non-Partitioned Primary Index
NUPI	Non-Unique Primary Index
NUSI	Non-Unique Secondary Index
OLAP	On-Line Analytical Processing
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
SDF	Specification for Data Formatting

SQL	Structured Query Language
UCS-2	Universal Coded Character Set containing 2 bytes
UPI	Unique Primary Index
USI	Unique Secondary Index
vproc	Virtual Process

Technical Information on the Web

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

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

Contents

Preface

About This Book	ii
List of Acronyms	iv
Technical Information on the Web.....	vi

Chapter 1:

SQL Data Handling Fundamentals

The ANSI SQL Standard	1-2
Terminology Differences Among ANSI SQL-92, ANSI SQL-99, and Teradata	1-6
SQL Flagger	1-8
Basic SQL Syntax and Lexicon	1-9
Invoking SQL Statements.....	1-11
SQL Requests	1-13
Null Statements or Comments	1-15
SQL Transaction Processing	1-16
SQL Transaction Processing in ANSI Mode.....	1-17
SQL Transaction Processing in Teradata Mode.....	1-18
Multistatement Requests.....	1-19
Dynamic and Static SQL.....	1-22
Dynamic SQL in Stored Procedures	1-24
Using SELECT With Dynamic SQL	1-26
Standard Form for Data in the Teradata RDBMS.....	1-28
Structure of an SQL Statement	1-30
Unqualified Object Names	1-31
Default Database	1-33
Manipulating Nulls	1-35
Session Parameters.....	1-41
Session Management	1-46
SQL Return Codes	1-49
SQL Statement Responses	1-52
SQL Success Response	1-53
SQL Warning.....	1-54
SQL Failure Response.....	1-55
SQL Error Response (ANSI Mode Only)	1-56

Chapter 2:**SQL Data Definition, Control, and Manipulation**

SQL Functional Families and Binding Styles	2-2
SQL Data Definition Language (DDL)	2-4
Tables	2-6
Global Temporary Tables and Volatile Tables	2-8
Global Temporary Tables	2-9
Volatile Tables	2-15
Primary Keys and Foreign Keys	2-18
Keys Versus Indexes	2-19
Indexes	2-20
Primary Indexes	2-26
Secondary Indexes	2-31
Full Table Scans	2-38
Join Indexes	2-39
Hash Indexes	2-44
Table Access Summary	2-48
Views	2-50
Triggers	2-52
Altering Table Structure and Definition	2-54
Dropping Tables, Views, Triggers, Procedures, and Indexes	2-56
Referential Integrity	2-57
Profiles	2-64
Roles	2-66
SQL Data Control Language (DCL)	2-68
SQL Data Manipulation Language (DML)	2-69
Query Processing	2-73
Subqueries	2-84
Teradata SQL Macros	2-86
Embedded SQL	2-89
Stored Procedures	2-90

Chapter 3:**SQL Lexicon**

SQL Lexicon Characters	3-2
Names	3-3
Name Validation For Japanese Language Support	3-6
Metadata Translation and Storage	3-14
Metadata Comparisons	3-15
Finding the Internal Hexadecimal Representation for Metadata	3-18

Specifying Names in a Logon String	3-20
Keywords	3-22
Delimiters	3-23
Literals.....	3-25
Numeric Literals.....	3-26
DateTime Literals	3-28
Interval Literals.....	3-29
Character Data Literals.....	3-30
Graphic Literals	3-31
Hexadecimal Literals	3-32
Built-In Functions Used As Literals.....	3-33
NULL as a Literal	3-34
SQL Operators	3-35
Lexical Separators	3-37
Statement Separator (Teradata).....	3-39
Request Terminator (Teradata)	3-41

Appendix A:
Notation Conventions

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

Appendix B:
Restricted Words for V2R5.0

Appendix C:
Teradata RDBMS Maxima

System Maxima.....	C-2
Database Maxima	C-3
Session Maxima	C-5

Index	Index-1
--------------------	---------



Chapter 1:

SQL Data Handling Fundamentals

This chapter describes basic data handling and database objects in the Teradata Relational Database Management System.

Topics include:

- The ANSI SQL standard
- Terminology differences among ANSI SQL-92, ANSI SQL-99, and Teradata
- SQL Flagger
- Basic SQL syntax and lexicon
- Invoking SQL statements
- SQL requests, including request processing
- Multistatement requests
- Dynamic and static SQL
- Standard form for data in the Teradata RDBMS
- Structure of an SQL statement
- Unqualified object names
- Default database
- Manipulating nulls
- Session parameters
- Session management
- SQL return codes

The ANSI SQL Standard

Formal Name for the Standard

The American National Standards Institute (ANSI) SQL standard, formally titled *International Standard ISO/IEC 9075:1992, Database Language SQL* (plus two later updates, both titled *Technical Corrigenda*), defines a version of Structured Query Language that all vendors of relational database management systems support to a greater or lesser degree.

Some of the common colloquial names for the standard are the following:

- SQL2
- SQL-92
- SQL/92

All of these expressions refer to the same standard.

Teradata SQL also supports many features defined in the new standard for core ANSI SQL, commonly referred to as SQL-99.

What Is the Motivation Behind an SQL Standard?

Teradata, like most vendors of relational database management systems, had its own dialect of the SQL language for many years prior to the development of the SQL-92 standard.

You might ask several questions like the following:

- Why should there be an industry-wide SQL standard?
- Why should any vendor with an entrenched user base consider modifying its SQL dialect to conform with the ANSI SQL-92 standard?

Why an SQL Standard?

National and international standards abound in the computer industry. As anyone who has worked in the industry for any length of time knows, standardization offers both advantages and disadvantages both to users and to vendors.

The principal advantages of having an SQL standard are the following:

- Open systems
The overwhelming trend in computer technology has been toward open systems with publicly defined standards to facilitate third party and end user access and development using the standardized products.
The ANSI SQL-92 standard provides an open definition for the SQL language.

- Less training for transfer and new employees
A programmer trained in ANSI-standard SQL can move from one SQL programming environment to another with no need to learn a new SQL dialect. When a core dialect of the language is the lingua franca for SQL programmers, the need for retraining is significantly reduced.
- Application portability
When there is a standardized public definition for a programming language, users can rest assured that any applications they develop to the specifications of that standard are portable to any environment that supports the same standard.
This is an extremely important budgetary consideration for any large scale end user application development project.
- Definition and manipulation of heterogeneous databases is facilitated
Many user data centers support multiple merchant databases across different platforms. A standard language for communicating with relational databases, irrespective of the vendor offering the database management software, is an important factor in reducing the overhead of maintaining such an environment.
- Intersystem communication is facilitated
It is common for an enterprise to exchange applications and data among different merchant databases.
Common examples of this are listed below.
 - Two-phase commit transactions where rows are written to multiple databases simultaneously.
 - Bulk data import and export between different vendor databases.These operations are made much cleaner and simpler when there is no need to translate data types, database definitions, and other component definitions between source and target databases.

Teradata Compliance With the ANSI Standard

Conformance to a standard presents problems for any vendor that produces an evolved product and supports a large user base.

Teradata had, in its historical development, produced any number of innovative SQL language elements that do not conform to the ANSI SQL standard, a standard that did not exist when those features were conceived. The existing Teradata user base had invested substantial time, effort, and capital into developing applications using that Teradata SQL dialect.

At the same time, new customers demand that vendors conform to open standards for everything from chip sets to operating systems to application programming interfaces.

Meeting these divergent requirements presents a challenge that Teradata SQL solves by following the multipronged policy outlined below.

WHEN ...	THEN ...
a new feature or feature enhancement is added to Teradata SQL	that feature conforms to the ANSI SQL-99 standard.
the difference between the Teradata SQL dialect and the ANSI SQL-99 standard for a language feature is slight	the ANSI SQL-99 is added to the Teradata feature as an option.
the difference between the Teradata SQL dialect and the ANSI SQL-99 standard for a language feature is significant	both syntaxes are offered and the user has the choice of operating in either Teradata or ANSI mode or of turning off SQL Flagger. The mode can be defined in the following ways: • Persistently Use the SessionMode field of the DBS Control Record to define persistent session mode characteristics. • For a session Use the BTEQ command .SET SESSION TRANSACTION to control transaction semantics for the session. Use the BTEQ command .SET SESSION SQLFLAG to control use of the SQL Flagger for the session. Use the SQL statement SET SESSION DATEFORM to control how data typed as DATE is handled.

WHEN ...	THEN ...	
a new feature or feature enhancement is added to Teradata SQL and that feature is not defined by the ANSI SQL-92 standard	that feature is designed using the following criteria set:	
	IF other vendors ...	THEN Teradata designs the new feature ...
	offer a similar feature or feature extension	to broadly comply with other solutions, but consolidates the best ideas from all and, where necessary, creates its own, cleaner solution.
	do not offer a similar feature or feature extension	- as cleanly and generically as possible with an eye toward creating a language element that will not be subject to major revisions in order to comply with future updates to the ANSI SQL-99 standard - in a way that offers the most power to users without violating any of the basic tenets of the ANSI SQL-99 standard.

Terminology Differences Among ANSI SQL-92, ANSI SQL-99, and Teradata

The ANSI SQL-92, SQL-99, and PSM-96 standards and Teradata occasionally use different terminology.

Terms

The following table lists the more important variances:

ANSI	Teradata
Assertion	Constraint
Base table	Table ¹
Binding style	Not defined, but implicitly includes the following: • Interactive SQL • Embedded SQL • ODBC • CLIv2
Authorization ID	User ID
Catalog	Dictionary
CLI	ODBC ²
Direct SQL	Interactive SQL
Domain	Not defined
Module	Not defined
Persistent stored module	Stored procedure
Schema	User Database
SQL database	Relational database
Viewed table	View
Not defined	Explicit transaction ³
Not defined	CLIv2 ⁴
Not defined	Macro ⁵

Notes

Note	Description
1	In the ANSI SQL standard, the term table has the following definitions: • A base table • A viewed table (view) • A derived table
2	ANSI CLI is not exactly equivalent to ODBC, but the ANSI standard is heavily based on the ODBC definition.
3	ANSI transactions are always implicit, beginning with an executable SQL statement and ending with either a COMMIT or a ROLLBACK statement.
4	Teradata CLIV2 is an implementation-defined binding style.
5	The function of Teradata macros is similar to that of ANSI persistent stored modules without having the loop and branch capabilities stored modules offer.

SQL Flagger

Function

The SQL Flagger, when enabled, reports any use of non-entry-level ANSI SQL features or other non-standard SQL. The SQL Flagger always permits statements flagged as non-entry-level or noncompliant ANSI SQL to execute. Its task is not to enforce the standard, but rather to return a warning message to the requestor noting the noncompliance.

The analysis includes syntax checking as well as some dictionary lookup, particularly the implicit assignment and comparison of different data types (where ANSI requires use of the CAST function to convert the types explicitly) as well as some semantic checks.

Enabling and Disabling the SQL Flagger

Flagging is enabled by a client application before a session is logged on and generally is used only to check for ANSI compliance in code that must be portable across multiple vendor environments.

The SQL Flagger is disabled by default. You can enable or disable it using any of the following procedures, depending on your application.

FOR this software ...	USE these commands or options ...	TO turn the SQL Flagger ...
BTEQ	.[SET] SESSION SQLFLAG ENTRY	to entry-level ANSI
	.[SET] SESSION SQLFLAG NONE	off
	See <i>Teradata BTEQ Reference</i> for more detail on using BTEQ commands.	
Preprocessor2	SQLFLAGGER(ENTRY)	to entry-level ANSI
	SQLFLAGGER(NONE)	off
	See <i>Teradata RDBMS Application Programming with C, COBOL and PL/I</i> for more detail on setting Preprocessor options.	
CLI	set lang_conformance = '2' set lang_conformance to '2'	to entry-level ANSI
	set lang_conformance = 'N'	off
	See <i>Teradata Call-Level Interface Version2</i> for more detail on setting the conformance field.	

Basic SQL Syntax and Lexicon

Basic SQL Statement Syntax

A typical SQL statement consists of a statement keyword, one or more column names, a database name, a table name, and one or more optional clauses introduced by keywords.

For example, in the following single-statement request, the statement keyword is SELECT:

```
SELECT deptno, name, salary
FROM personnel.employee
WHERE deptno IN(100, 500)
ORDER BY deptno, name ;
```

The select list for this statement is made up of the names:

- Deptno, name, and salary (the column names)
- Personnel (the database name)
- Employee (the table name)

The search condition, or WHERE clause, is introduced by the keyword WHERE.

```
WHERE deptno IN(100, 500)
```

The sort order, or ORDER BY, clause is introduced by the keywords ORDER BY.

```
ORDER BY deptno, name
```

Basic SQL Statement Lexicon

The essential lexicon for SQL statements is described in the sections that follow.

Statement Punctuation

SQL statements are punctuated using the following syntax elements.

Syntax Element	Name	Function in an SQL Statement
.	FULL STOP	• Separates database names from table, trigger, and stored procedure names (for example, <i>personnel.employee</i>). • Separates table names from a particular column name (for example, <i>employee.deptno</i>).
,	COMMA	Separates and distinguishes column names in the select list, or column names or parameters in an optional clause.

Syntax Element	Name	Function in an SQL Statement
'	APOSTROPHE	Delimits the boundaries of character string constants.
(	LEFT PARENTHESIS	Groups expressions or defines the limits of a phrase.
)	RIGHT PARENTHESIS	
;	SEMICOLON	Separates statements in multi-statement requests and terminates requests submitted via utilities such as BTEQ.
"	QUOTATION MARKS	Defines the extent of nonstandard names.
:	COLON	Prefixes reference parameters or client system variables.

To include an APOSTROPHE character or show possession in a title, double the APOSTROPHE characters.

Case Sensitivity

See the following topics in the chapter "Character Data Types" in *Teradata RDBMS SQL Reference, Volume 3*.

- "Defining Case Sensitivity for Table Columns"
- "CASESPECIFIC Phrase"
- "UPPERCASE Phrase"

and the topic "Case Sensitivity of Character String Literals" in the chapter "Data Literals" in *Teradata RDBMS SQL Reference, Volume 3*.

See the following topics in the chapter "Character String Functions" in *Teradata RDBMS SQL Reference, Volume 5*.

- "LOWER Function"
- "UPPER Functions"

Invoking SQL Statements

Introduction

One of the primary issues that motivated the development of relational database management systems was the perceived need to create database management systems that could be queried not just by predetermined, hard-coded requests but also interactively by well-formulated *ad hoc* queries.

SQL addresses this issue by offering four ways to invoke an executable statement:

- Interactively from a terminal
- Embedded within an application program
- Dynamically performed from within an embedded application
- Embedded within a stored procedure

Executable SQL Statements

An executable SQL statement is one that performs an action. The action can be on data or on a transaction or some other entity at a higher level than raw data.

Some examples of executable SQL statements are the following:

- SELECT
- CREATE TABLE
- COMMIT
- CONNECT
- PREPARE

Most, but not all, executable SQL statements can be performed interactively from a terminal using an SQL query manager like BTEQ or QueryMan.

Types of executable SQL commands that cannot be performed interactively are the following:

- Cursor control and declaration statements
- Dynamic SQL control statements
- Stored procedure control statements and condition handlers
- Connection control statements
- Special forms of SQL statements such as SELECT INTO

These statements can only be used within an embedded SQL or stored procedure application.

Nonexecutable SQL Statements

A nonexecutable SQL statement is one that declares an SQL statement, object, or host or local variable to the preprocessor or stored procedure compiler. Nonexecutable SQL statements are not processed during program execution.

Some examples of nonexecutable SQL statements are:

- DECLARE CURSOR
- BEGIN DECLARE SECTION
- EXEC SQL

These statements can only be used within an embedded SQL application.

In stored procedures, DECLARE is the only nonexecutable control statement. DECLARE is used for local variable declaration within a stored procedure.

SQL Requests

Introduction

A *request* to the Teradata RDBMS consists of one or more SQL statements and can span any number of input lines. Teradata RDBMS can receive and perform SQL *statements* that are:

- Embedded in a client application program that is written in a procedural language.
- Embedded in a stored procedure.
- Entered interactively through the BTEQ or QueryMan interfaces.
- Submitted in a BTEQ script as a batch job.
- Submitted through other supported methods (such as Cliv2, ODBC, and JDBC)

Single Statement Requests

A single statement request consists of a statement keyword followed by one or more expressions, other keywords, clauses, and phrases. A single statement request is treated as a solitary unit of work.

Single Statement Syntax



Multistatement Requests

A *multistatement request* consists of two or more statements separated by SEMICOLON characters.

Multistatement requests are non-ANSI standard.

For more information, see [“Multistatement Requests” on page 1-19](#).

Multistatement Syntax



ANSI Mode

If an error is found in a request, then that request is aborted, but *not* the entire transaction.

Teradata Mode

A multistatement request is treated as an implicit transaction. That is, if an error is found in any statement in the request, then the entire transaction is aborted.

Abort processing proceeds as described in the following table.

Stage	Process
1	Backs out any changes made to the database as a result of any preceding statements.
2	Deletes any associated spooled output.
3	Releases any associated locks.
4	Bypasses any remaining statements in the transaction.

Complete Requests

A request is considered *complete* when either an End of Text character or the request terminator is encountered. The request terminator is a SEMICOLON character. It is the last nonpad character on an input line.

A request terminator is optional except when the request is embedded in an SQL macro or when it is entered through BTEQ.

Null Statements or Comments

A null statement is a statement that has no content except for pad characters or SQL comments.

Simple Null Statement or Comment Syntax



where:

Syntax element ...	Specifies ...
--	the text that follows is an SQL comment. This is the ANSI-compliant, simple form of indicating an SQL comment.

A request terminator is optional except when the request is embedded in an SQL macro or when it is entered through BTEQ.

Bracketed Null Statement or Comment Syntax



where:

Syntax element ...	Specifies ...
/*	the text that follows up to the */ character sequence is an SQL comment. This is the ANSI-compliant, bracketed form of indicating the beginning of an SQL comment.
*/	the SQL comment terminator. This is the ANSI-compliant, bracketed form of indicating the end of an SQL comment.

The loop indicates a multistatement comment or null statement sequence.

A request terminator is optional except when the request is embedded in an SQL macro or when it is entered through BTEQ.

SQL Transaction Processing

Introduction

A *transaction* is a logical unit of work where the statements nested within the transaction either execute successfully as a group or do not execute.

Transaction Processing Mode

You can perform transaction processing in either of the following modes:

- ANSI
- Teradata

In ANSI mode, transaction processing adheres to the rules defined by the ANSI SQL specification. In Teradata mode, transaction processing follows the rules defined by Teradata over years of evolution.

To set the transaction processing mode, use the:

- SessionMode field of the DBS Control Record
- BTEQ command .SET SESSION TRANSACTION
- Preprocessor2 TRANSACT() option

For More Information

The next few pages highlight some of the differences between transaction processing in ANSI mode and transaction processing in Teradata mode.

For detailed information on statement and transaction processing, see *Teradata RDBMS SQL Reference, Volume 2*.

SQL Transaction Processing in ANSI Mode

Introduction

Transactions are always implicit in ANSI mode.

A transaction initiates when one of the following happens:

- The first SQL statement in a session executes
- The first statement following the close of a transaction executes

The COMMIT or ROLLBACK/ABORT statements close a transaction.

If a transaction includes a DDL statement, it must be the last statement in the transaction.

Note that DATABASE and SET SESSION are DDL statements. See “Rollback Processing” in Chapter 6 – “Transaction Processing” of *Teradata RDBMS SQL Reference, Volume 2*.

If a session terminates with an open transaction, then any effects of that transaction are rolled back.

Two-Phase Commit (2PC)

Sessions in ANSI mode do not support 2PC. If an attempt is made to use the 2PC protocol in ANSI mode, the Logon process aborts and an error returns to the requestor.

SQL Transaction Processing in Teradata Mode

Introduction

A Teradata SQL transaction can be a single Teradata SQL statement, or a sequence of Teradata SQL statements, treated as a single unit of work.

Each request is processed as one of the following transaction types:

- Implicit
- Explicit
- Two-phase commit (2PC)

Implicit Transactions

An implicit transaction is a request that does not include the `BEGIN TRANSACTION` and `END TRANSACTION` statements. The implicit transaction starts and completes all within the SQL request: it is self-contained.

An implicit transaction can be one of the following:

- A single DML statement that affects one or more rows of one or more tables
- A macro containing one or more statements
- A request containing multiple statements separated by `SEMICOLON` characters. Each `SEMICOLON` character can appear anywhere in the input line. The Parser interprets a `SEMICOLON` character at the end of an input line as a transaction terminator.

DDL statements are not valid in an implicit multistatement transaction.

Explicit Transactions

In Teradata mode, an explicit transaction contains one or more statements enclosed by `BEGIN TRANSACTION` and `END TRANSACTION` statements. The first `BEGIN TRANSACTION` initiates a transaction and the last `END TRANSACTION` terminates the transaction.

When multiple statements are included in an explicit transaction, you can only specify a DDL statement if it is the last statement in the series.

Two-Phase Commit (2PC) Rules

Two-phase commit (2PC) protocol is supported in Teradata mode.

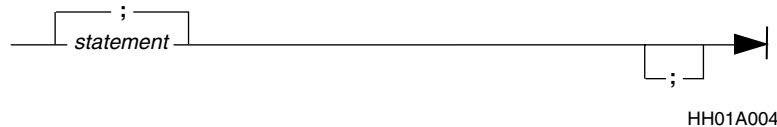
- A 2PC transaction contains one or more DML statements that affect multiple databases and are coordinated externally using the 2PC protocol.
- A DDL statement is not valid in a two-phase commit transaction.

Multistatement Requests

Definition

An atomic request containing more than one SQL statement, each terminated by a SEMICOLON character.

Syntax



ANSI Compliance

Multistatement requests are non-ANSI SQL-99 standard.

Rules

The Teradata RDBMS imposes restrictions on the use of multistatement requests:

- Only one USING row descriptor is permitted per request, so only one USING row descriptor can be used per multistatement request.
This rule applies to interactive SQL only. Embedded SQL and stored procedures do not permit the USING row descriptor
- A multistatement request cannot include a DDL statement.
- Stored procedures do not support multistatement requests. Each statement is treated as a separate request.

Power of Multistatement Requests

The multistatement request is application-independent. It improves performance for a variety of applications that can package more than one SQL statement at a time. BTEQ, CLI, and the SQL preprocessor all support multistatement requests.

Multistatement requests improve system performance by reducing processing overhead. By performing a series of statements as one request, performance for the client, the Parser, and the Database Manager are all enhanced.

Because of this reduced overhead, using multistatement requests also decreases response time. A multistatement request that contains 10 SQL statements could be as much as 10 times more efficient than the 10 statements entered separately (depending on the types of statements submitted).

Multistatement Requests Treated as Transaction

In a multistatement request, treated as a single unit of work, either all statements in the request complete successfully, or the entire request is aborted.

In ANSI mode, the request is rolled back if aborted. In Teradata mode, any updates to the database up to that point for the transaction are rolled back.

Parallel Step Processing

Teradata can perform some requests in parallel (see [“Parallel Steps” on page 1-21](#)). This capability applies both to implicit transactions, such as macros and multistatement requests, and to Teradata-style transactions explicitly defined by BEGIN/END TRANSACTION statements.

Statements in a multistatement request are broken down by the Parser into one or more steps that direct the execution performed by the AMPs. It is these steps, not the actual statements, that are executed in parallel.

A handshaking protocol between the PE and the AMP allows the AMP to determine when the PE can dispatch the next parallel step.

Up to twenty parallel steps can be processed per request if channels are not required, such as a request with an equality constraint based on a primary index value. Up to ten channels can be used for parallel processing when a request is not constrained to a primary index value.

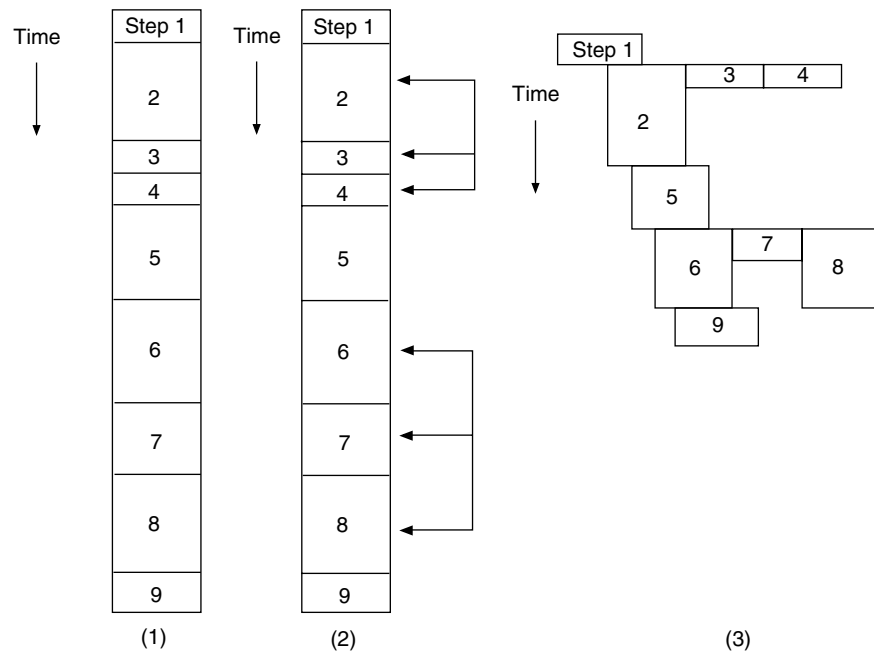
For example, if an INSERT step and a DELETE step are allowed to be run in parallel, the AMP informs the PE that the DELETE step has progressed to the point where the INSERT step will not impact it adversely. This handshaking protocol also reduces the chance of a deadlock.

[“Parallel Steps” on page 1-21](#) illustrates the following process:

Stage	Process
1	The statements in a multistatement request are broken down into a series of steps.
2	The Optimizer determines which steps in the series can be executed in parallel.
3	The steps are processed.

Each step undergoes some preliminary processing before it is executed, such as placing locks on the objects involved. These preliminary processes are not performed in parallel.

Parallel Steps



FF02A001

Dynamic and Static SQL

Definition: Dynamic SQL

Dynamic SQL is a method of invoking an SQL statement by compiling and performing it at runtime from within an embedded SQL application program or a stored procedure.

The specification of data to be manipulated by the statement is also determined at runtime.

Definition: Static SQL

Static SQL is, by default, any method of invoking an SQL statement that is not dynamic.

ANSI Compliance

Dynamic SQL is ANSI SQL-99-compliant.

The ANSI SQL standard does not define the expression *static SQL*, but relational database management commonly uses it as a comparison with the ANSI-defined expression dynamic SQL.

Ad Hoc and Hard-Coded Invocation of SQL Statements

Perhaps the best way to think of dynamic SQL is to contrast it with ad hoc SQL statements created and executed from a terminal and with preprogrammed SQL statements created by an application programmer and executed by an application program.

In the case of the ad hoc query, everything legal is available to the requester: choice of SQL statements and clauses, variables and their names, databases, tables, and columns to manipulate, and literals.

In the case of the application programmer, the choices are made in advance and hard-coded into the source code of the application. Once the program is compiled, nothing can be changed short of editing and recompiling the application.

Dynamic Invocation of SQL Statements

Dynamic SQL offers a compromise between these two extremes. By choosing to code dynamic SQL statements in the application, the programmer has the flexibility to allow an end user to select not only the variables to be manipulated at run time, but also the SQL statement to be executed.

As you might expect, the flexibility that dynamic SQL offers a user is offset by more work and increased attention to detail on the part of the application

programmer, who needs to set up additional dynamic SQL statements and manipulate information in the SQLDA to ensure a correct result.

This is done by first preparing, or compiling, an SQL text string containing placeholder tokens at run time and then executing the prepared statement, allowing the application to prompt the user for values to be substituted for the placeholders.

SQL Statements to Set Up and Invoke Dynamic SQL

The embedded SQL statements for preparing and executing an SQL statement dynamically are PREPARE, EXECUTE, and EXECUTE IMMEDIATE.

For details, see “PREPARE,” “EXECUTE (Dynamic SQL Form,” and “EXECUTE IMMEDIATE,” in Chapter 3: “SQL Data Manipulation Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 6*.

EXECUTE IMMEDIATE is a special form that combines PREPARE and EXECUTE into one statement. EXECUTE IMMEDIATE can only be used in the case where there are no input host variables.

This description applies directly to all executable SQL statements except SELECT, which requires additional handling.

Note that SELECT INTO *cannot* be invoked dynamically.

Further Information

For more information on ...	See ...
examples of dynamic SQL code in C, COBOL, and PL/I	<i>Teradata RDBMS Application Programming with C, COBOL and PL/I.</i>

Dynamic SQL in Stored Procedures

Overview

The way stored procedures support dynamic SQL statements is different from the way embedded SQL does.

Use the following statement to set up and invoke dynamic SQL in a stored procedure:

```
CALL DBC.SysExecSQL(string_expression)
```

where *string_expression* is any valid string expression that builds an SQL statement.

The string expression is composed of string literals, status variables, local variables, input (IN and INOUT) parameters, and for-loop aliases. Dynamic SQL statements are not validated at compile time.

The following example illustrates the usage of dynamic SQL within stored procedure source text:

```
CREATE PROCEDURE new_sales_table( my_table VARCHAR(30),  
                                my_database VARCHAR(30))  
BEGIN  
  DECLARE sales_columns VARCHAR(128)  
  DEFAULT '(item INTEGER, price DECIMAL(8,2), sold  
                                INTEGER)';  
  CALL DBC.SysExecSQL('CREATE TABLE ' || :my_database ||  
                    '.' || :my_table || :sales_columns);  
END;
```

Any number of calls to SysExecSQL can be made in a stored procedure. With each call, only a single SQL statement may be specified in the string expression for dynamic SQL.

Because the request text of dynamic SQL statements can vary from execution to execution, dynamic SQL provides more usability and conciseness to the stored procedure definition.

Restrictions

Dynamic SQL statements can be specified in a stored procedure only when the creator is the same as the immediate "owner" of the stored procedure.

The following SQL statements *cannot* be specified as dynamic SQL in stored procedures:

- CALL
- DATABASE
- HELP
- SELECT
- SET SESSION ACCOUNT
- SET SESSION DATEFORM
- SHOW
- CREATE PROCEDURE
- EXPLAIN
- REPLACE PROCEDURE
- SELECT - INTO
- SET SESSION COLLATION
- SET TIME ZONE

Further Information

For rules and usage examples of dynamic SQL statements in stored procedures, see “SQL Stored Procedures” in *Teradata RDBMS SQL Reference, Volume 6*.

Using SELECT With Dynamic SQL

Unlike other executable SQL statements, SELECT returns information beyond statement responses and return codes to the requester.

DESCRIBE Statement

Because the requesting application needs to know how much (if any) data will be returned by a dynamically prepared SELECT, you must use an additional SQL statement, DESCRIBE, to make the application aware of the demographics of the data to be returned by the SELECT statement (see “DESCRIBE” in Chapter 3: “SQL Data Manipulation Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 6*).

DESCRIBE writes this information to the SQLDA declared for the SELECT statement as follows.

THIS information ...	IS written to this field of SQLDA ...
number of values to be returned	SQLN
column name or label of n^{th} value	SQLVAR (n^{th} row in the SQLVAR(n) array)
column data type of n^{th} value	
column length of n^{th} value	

General Procedure

The following table describes the general procedure an application must follow to set up, execute, and retrieve the results of a SELECT statement invoked as dynamic SQL.

Step	Action
1	Declare a dynamic cursor for the SELECT in the form: <code>DECLARE cursor_name CURSOR FOR sql_statement_name</code>
2	Declare the SQLDA, preferably using an INCLUDE SQLDA statement.
3	Build and PREPARE the SELECT statement.

Step	Action						
4	Issue a DESCRIBE statement in the form: DESCRIBE <i>sql_statement_name</i> INTO SQLDA DESCRIBE performs the following actions.						
	<table><tr><th>Step</th><th>Action</th></tr><tr><td>1</td><td>Interrogate the database for the demographics of the expected results.</td></tr><tr><td>2</td><td>Write the addresses of the target variables to receive those results to the SQLDA. This step is bypassed if any of the following occurs: • The request does not return any data. • An INTO clause was present in the PREPARE statement. • The statement returns known columns and the INTO clause is used on the corresponding FETCH statement. • The application code defines the SQLDA. </td></tr></table>	Step	Action	1	Interrogate the database for the demographics of the expected results.	2	Write the addresses of the target variables to receive those results to the SQLDA. This step is bypassed if any of the following occurs: • The request does not return any data. • An INTO clause was present in the PREPARE statement. • The statement returns known columns and the INTO clause is used on the corresponding FETCH statement. • The application code defines the SQLDA.
	Step	Action					
	1	Interrogate the database for the demographics of the expected results.					
2	Write the addresses of the target variables to receive those results to the SQLDA. This step is bypassed if any of the following occurs: • The request does not return any data. • An INTO clause was present in the PREPARE statement. • The statement returns known columns and the INTO clause is used on the corresponding FETCH statement. • The application code defines the SQLDA.						
5	Allocate storage for target variables to receive the returned data based on the demographics reported by DESCRIBE.						
6	Retrieve the result rows using the following SQL cursor control statements: • OPEN <i>cursor_name</i> • FETCH <i>cursor_name</i> USING DESCRIPTOR SQLDA • CLOSE <i>cursor_name</i>						

Note that in Step 6, results tables are examined one row at a time using the selection cursor. This is because client programming languages do not support data in terms of sets, but only as individual records.

Standard Form for Data in the Teradata RDBMS

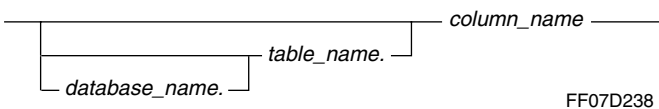
Introduction

Data in the Teradata RDBMS is presented to a user according to the relational model, which models data as two dimensional tables with rows and columns. Each row of a table is composed one or more fields identified by column name. Each field contains a data item (or a null) having a single data type.

Definition: Database

A database consists of a collection of related tables. The rows of a table can be considered an unordered set.

Syntax for Referencing a Column



FF07D238

where:

Syntax element ...	Specifies ...
<i>database_name</i>	a qualifying name for the database in which the table and column being referenced is stored. Depending on the ambiguity of the reference, <i>database_name</i> might or might not be required. See “Unqualified Object Names” on page 1-31.
<i>table_name</i>	a qualifying name for the table in which the column being referenced is stored. Depending on the ambiguity of the reference, <i>table_name</i> might or might not be required. See “Unqualified Object Names” on page 1-31.
<i>column_name</i>	one of the following: • The name of the column being referenced • The alias of the column being referenced • The keyword PARTITION See “Column Alias” on page 1-29.

Definition: Fully Qualified Column Name

A fully qualified name consists of a database name, a table name, and a column name.

For example, a fully qualified reference for the Name column in the Employee table of the Personnel database is:

```
Personnel.Employee.Name
```

Column Alias

In addition to referring to a column by name, an SQL query can reference a column by an alias, or correlation name. Column aliases are used for join indexes when two columns have the same name. However, an alias can be used for any column when a pseudonym is more descriptive or easier to use.

You can specify a column alias with or without the keyword AS on the first reference to the column in the query. The following example creates and uses aliases for the first two columns.

```
SELECT departnumber AS d, employeename e, salary
FROM personnel.employee
WHERE d IN(100, 500)
ORDER BY d, e ;
```

Alias names must meet the same requirements as names of other database objects. See [“Names” on page 3-3](#).

The scope of alias names is confined to the query.

Referencing All Columns in a Table

An asterisk references all columns in a row simultaneously, for example, the following SELECT statement references all columns in the Employee table. A list of those fully qualified column names follows the query.

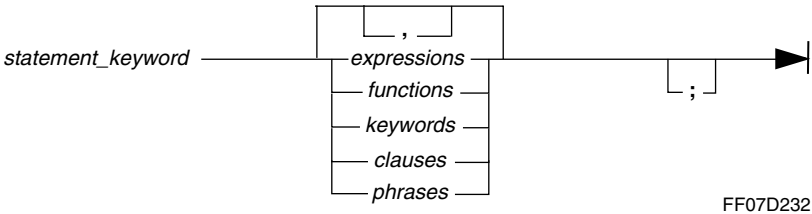
```
SELECT * FROM Employee;

Personnel.Employee.EmpNo
Personnel.Employee.Name
Personnel.Employee.DeptNo
Personnel.Employee.JobTitle
Personnel.Employee.Salary
Personnel.Employee.YrsExp
Personnel.Employee.DOB
Personnel.Employee.Sex
Personnel.Employee.Race
Personnel.Employee.MStat
Personnel.Employee.EdLev
Personnel.Employee.HCap
```

Structure of an SQL Statement

Syntax

The following diagram indicates the basic structure of an SQL statement.



FF07D232

where:

This syntax element ...	Specifies ...
<i>statement_keyword</i>	the name of the statement and introduces the primary clause.
<i>expressions</i>	constants, name references, or operations using names and constants.
<i>functions</i>	the name of a function and its arguments, if any.
<i>keywords</i>	special values introducing clauses or phrases or representing special objects, such as NULL or ROWID. Most keywords are reserved words and cannot be used in names.
<i>clauses</i>	subordinate statement qualifiers.
<i>phrases</i>	data attribute phrases.
<i>;</i>	the Teradata statement separator and request terminator. The semicolon separates statements in a multistatement request and terminates a request when it is the last non-blank character on an input line in BTEQ. Note that the request terminator is required for a request defined in the body of a macro. For a discussion of macros and their use, see “Teradata SQL Macros” on page 2-86 .

Unqualified Object Names

Definition

An unqualified object name is a table, column, trigger, macro, or stored procedure reference that is not fully qualified. For example, the WHERE clause in the following statement uses “DeptNo” as an unqualified column name:

```
SELECT *  
FROM Personnel.Employee  
WHERE DeptNo = 100 ;
```

Unqualified Column Names

You can omit database and table name qualifiers when you reference columns as long as the reference is not ambiguous.

For example, the WHERE clause in the following statement:

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

can be written as:

```
WHERE DeptNo = 100 ;
```

because the database name and table name can be derived from the Personnel.Employee reference in the FROM clause.

Omitting Database Names

When you omit the database name qualifier, Teradata looks in the following databases to find the unqualified table, view, trigger, or macro name:

- The *default database*, which is established by a DATABASE, CREATE USER, MODIFY USER, CREATE PROFILE, or MODIFY PROFILE statement
- Other databases, if any, referenced by the SQL statement
- The login user database for a volatile table, if the unqualified object name is a table name

The search must find the table name in only one of those databases. An ambiguous name error message results if the name exists in more than one of those databases.

For example, if your login user database has no volatile tables named Employee and you have established Personnel as your default database, you can omit the Personnel database name qualifier from the preceding sample query.

Rules for Name Resolution

The following rules govern name resolution.

- Name resolution is performed statement-by-statement.
- When an INSERT statement contains a subquery, names are resolved in the subquery first.
- Names in a view are resolved when the view is created.
- Names in a macro data *manipulation* statement are resolved when the macro is created.
- Names in a macro data *definition* statement are resolved when the macro is performed using the default database of the user submitting the EXECUTE statement.

Therefore, you should fully qualify all names in a macro data definition statement, unless you specifically intend for the user's default to be used.

- Names in stored procedure statements are resolved when the procedure is created. All unqualified object names acquire the current default database name.
- An ambiguous unqualified name returns an error to the requestor.

Further Information

FOR More Information on ...	SEE ...
default databases	"Default Database" on page 1-33
the DATABASE statement	Chapter 1: "SQL Data Definition Language Statement Syntax" in <i>Teradata RDBMS SQL Reference, Volume 4</i> .
the CREATE USER statement	
the MODIFY USER statement	

Default Database

Definition

The default database is a Teradata extension to SQL that defines a database that Teradata uses to look for unqualified table, view, trigger, or macro names in SQL statements.

The default database is not the only database that Teradata uses to find an unqualified table, view, trigger, or macro name in an SQL statement, however; Teradata also looks for the name in:

- Other databases, if any, referenced by the SQL statement
- The login user database for a volatile table, if the unqualified object name is a table name

If the unqualified object name exists in more than one of the databases in which Teradata looks, the SQL statement produces an ambiguous name error.

Establishing a Permanent Default Database

You can establish a permanent default database that is invoked each time you log on.

TO ...	USE one of the following SQL Data Definition Statements ...
define a permanent default database	• CREATE USER, with a DEFAULT DATABASE clause. • CREATE USER, with a PROFILE clause that specifies a profile that defines the default database.
change your permanent default database definition	• MODIFY USER, with a DEFAULT DATABASE clause.
add a default database when one had not been established previously	• MODIFY USER, with a PROFILE clause. • MODIFY PROFILE, with a DEFAULT DATABASE clause.

For example, the following statement automatically establishes Personnel as the default database for Marks at the next logon:

```
MODIFY USER marks AS  
DEFAULT DATABASE = personnel ;
```

After you assign a default database, Teradata uses that database as one of the databases to look for all unqualified object references.

To obtain information from a table, view, trigger, or macro in another database, fully qualify the table reference by specifying the database name, a FULLSTOP character, and the table name.

Establishing a Default Database for a Session

You can establish a default database for the current session that Teradata uses to look for unqualified table, view, trigger, or macro names in SQL statements.

TO ...	USE ...
establish a default database for a session	the DATABASE statement.

For example, after entering the following SQL statement:

```
DATABASE personnel ;
```

you can enter a SELECT statement as follows:

```
SELECT deptno (TITLE 'Org'), name  
FROM employee ;
```

which has the same results as:

```
SELECT deptno (TITLE 'Org'), name  
FROM personnel.employee;
```

To establish a default database, you must have some privilege on a database, macro, stored procedure, table, user, or view in that database. Once defined, the default database remains in effect until the end of a session or until it is replaced by a subsequent DATABASE statement.

Further Information

FOR More Information on ...	SEE ...
the DATABASE statement	Chapter 1: “SQL Data Definition Language Statement Syntax” in <i>Teradata RDBMS SQL Reference, Volume 4</i> .
the CREATE USER statement	
the MODIFY USER statement	
fully-qualified names	“Standard Form for Data in the Teradata RDBMS” on page 1-28.
	“Unqualified Object Names” on page 1-31.
using profiles to define a default database	“Profiles” on page 2-64.

Manipulating Nulls

Introduction

A null represents any of three things:

- an empty field
- an unknown value
- an unknowable value

Nulls are neither values nor do they signify values; they represent the *absence* of value. A null is a place holder indicating that no value is present.

You *cannot* solve for the value of a null because, by definition, it *has* no value. For example, the expression `NULL = NULL` has no meaning and therefore can never be true. A query that specifies the predicate `WHERE NULL = NULL` is not valid because it can never be true. The meaning of the comparison it specifies is not only unknown, but unknowable.

These properties make the use and interpretation of nulls in SQL problematic. The following sections outline the behavior of nulls for various SQL operations to help you to understand how to use them in data manipulation statements and to interpret the results those statements effect.

NULL Literals

See [“NULL as a Literal” on page 3-34](#) for information on how to use the NULL keyword as a literal.

Nulls and Predicates

Nulls are not valid as predicate conditions with the exception of CASE (see [“Nulls and CASE Expressions” on page 1-37](#)).

Nulls and Arithmetic Operators and Functions

If an operand of any arithmetic operator or function is null, then the result of the operation or function is null with the following exceptions:

WHEN the expression is ...	THEN the result is ...
<code>5 + NULL</code>	null
<code>LOG (NULL)</code>	null
<code>NULLIFZERO (NULL)</code>	null
<code>ZEROIFNULL (NULL)</code>	0
<code>COALESCE (NULL, 6)</code>	6

Nulls and Comparison Operators

If either operand of a comparison operator is NULL, then the result is unknown and an error is returned to the requestor. The following examples indicate this behavior.

```
5 = NULL
5 <> NULL
NULL = NULL
NULL <> NULL
5 = NULL + 5
```

Note that if the argument of the NOT operator is unknown, the result is also unknown. This translates to FALSE as a final boolean result.

Instead of using comparison operators, use the IS NULL operator to search for fields that contain nulls and the IS NOT NULL operator to search for fields that do not contain nulls. For details, see [“Searching for Nulls” on page 1-36](#) and [“Excluding Nulls” on page 1-36](#).

Using IS NULL is different from using the comparison operator =. When you use an operator like =, you specify a comparison between values or value expressions, whereas when you use the IS NULL operator, you specify an existence condition.

Searching for Nulls

TO ...	USE ...
search for fields that contain nulls	the operator IS NULL.

The IS NULL operator tests row data for the presence of nulls. For example, to search for the names of all employees who have a null in the deptno column, you could enter the statement:

```
SELECT name
FROM employee
WHERE deptno IS NULL ;
```

This query produces the names of all employees with a null in the deptno field.

Excluding Nulls

TO ...	USE ...
exclude nulls from the results of a query	the operator IS NOT NULL.

For example, to search for the names of all employees with a value other than null in the jobtitle column, enter the statement.

```
SELECT name
FROM employee
WHERE jobtitle IS NOT NULL ;
```

Searching for Nulls and Non-Nulls Together

To search for nulls and non-nulls in the same statement, the search condition for nulls must be separate from any other search conditions.

For example, to select the names of all employees with the job title of Vice Pres, Manager, or null, enter the following SELECT statement.

```
SELECT name, jobtitle
FROM employee
WHERE jobtitle IN ('Manager' , 'Vice Pres')
OR jobtitle IS NULL ;
```

Including NULL in the IN list has no effect because NULL is never equal to NULL or to any value.

Nulls and DateTime and Interval Data

The general rule for managing nulls with DateTime and Interval data environment is that, for individual definitions, the rules are identical to those for handling numeric and character string values.

WHEN any component of this type of expression is null ...	THEN the result is ...
Value	null.
Conditional	FALSE.
CASE	as it would be for any other CASE expression.

A DateTime or Interval value is either atomically null or it is not null. For example, you cannot have an interval of YEAR TO MONTH in which YEAR is null and MONTH is not.

Nulls and CASE Expressions

The ANSI definition for CASE and its related expressions COALESCE and NULLIF specifies that they can return a null. Because of this, their behavior is an exception to the rules for all other predicates and expressions (see [“Introduction” on page 1-35](#)).

- If no ELSE clause is specified in a CASE expression and the evaluation falls through all the WHEN clauses, the result is NULL.
- Nulls and null expressions are valid as the CASE test expression in a valued CASE expression. The following example is valid:

```
SELECT CASE NULL
        WHEN 10
        THEN 'TEN'
END ;
```

The preceding example always returns NULL because no ELSE clause is specified, and the evaluation falls through the WHEN clause because NULL is not equal to any value or to NULL.

- Comparing NULL to any value or to NULL is always FALSE. When testing for NULL, it is best to use a searched CASE expression using the IS NULL or IS NOT NULL operators in the WHEN condition.

The following example is valid:

```
SELECT CASE
      WHEN column_1 IS NULL
      THEN 'NULL'
    END
FROM table_1;
```

Often, Teradata can detect when an expression that always evaluates to NULL is compared to some other expression or NULL, and gives an error that recommends using IS NULL or IS NOT NULL instead.

The following example is not legal:

```
SELECT CASE column_1
      WHEN NULL
      THEN 'NULL'
    END
```

- Nulls and null expressions are valid as THEN clause conditions. The following example is valid.

```
SELECT CASE
      WHEN column_1 = 10
      THEN NULL
    END
FROM table_1
```

Note that, unlike the previous examples, the NULL in the THEN clause is an SQL keyword and not the value of a character constant.

For details on the rules for nulls in CASE, NULLIF, and COALESCE expressions, see Chapter 1: “CASE Expression” in *Teradata RDBMS SQL Reference, Volume 5*.

Null Sorts as the Lowest Value in a Collation

When you use an ORDER BY clause to sort records, Teradata sorts null as the lowest value. Sorting nulls can vary from RDBMS to RDBMS. Other systems may sort null as the highest value.

If any row has a null in the column being grouped, then all rows having a null are placed into one group.

Teradata Replaces Nulls With Values on Return to Client in Record Mode

When the Teradata RDBMS returns information to a client system in record mode, nulls must be replaced with some value for the underlying column because client system languages do not recognize nulls.

The following table shows the values returned for various column data types.

Data Type	Substitute Value Returned for Null
INTEGER SMALLINT BYTEINT FLOAT DECIMAL REAL DOUBLE PRECISION NUMERIC	0
CHARACTER(n) DATE (ANSI) TIME TIMESTAMP INTERVAL	Pad character (or <i>n</i> pad characters for CHARACTER(<i>n</i>), where <i>n</i> > 1)
BYTE[(n)]	0 bytes if <i>n</i> omitted else <i>n</i> binary zero bytes
VARBYTE(n)	0-length byte string
VARCHARACTER(n)	0-length character string
DATE (Teradata)	0

The substitute values returned for nulls are not, by themselves, distinguishable from valid non-null values. Data from CLI is normally accessed in IndicData mode, in which additional identifying information that flags nulls is returned to the client.

BTEQ uses the identifying information, for example, to determine whether the values it receives are values or just aliases for nulls so it can properly report the results. Note that BTEQ displays nulls as ?, which are not by themselves distinguishable from a CHAR or VARCHAR value of '?'.

Nulls and Aggregate Functions

With the important exception of COUNT(*), aggregate functions ignore nulls in their arguments. This treatment of nulls is very different from the way arithmetic operators and functions treat them.

This behavior can result in apparent nontransitive anomalies. For example, if there are nulls in either column A or column B (or both), then the following expression is virtually always true.

$$\text{SUM}(A) + (\text{SUM } B) <> \text{SUM } (A+B)$$

In other words, for the case of SUM, the result is never a simple iterated addition if there are nulls in the data being summed.

The only exception to this is the case in which the values for columns A and B are *both* null in the same rows, because in those cases the entire row is

disregarded in the aggregation. This is a trivial case that does not violate the general rule.

The same is true, the necessary changes being made, for all the aggregate functions except COUNT(*).

If this property of nulls presents a problem, you can always do either of the following workarounds, each of which produces the desired result of the aggregate computation $SUM(A) + SUM(B) = SUM(A+B)$.

- Always define NUMERIC columns as NOT NULL DEFAULT 0.
- Use the ZEROIFNULL function within the aggregate function to convert any nulls to zeros for the computation, for example

`SUM(ZEROIFNULL(x) + ZEROIFNULL(y))`

which produces the same result as this:

`SUM(ZEROIFNULL(x) + ZEROIFNULL(y))`.

COUNT(*) *does* include nulls in its result. For details, see “COUNT Function” in Chapter 6: “Aggregate Functions” of *Teradata RDBMS SQL Reference, Volume 5*.

Session Parameters

Introduction

The following session parameters can be controlled with keywords or predefined system variables:

Parameter	Valid Keywords or System Variables
SQL Flagger	ON
	OFF
Transaction Mode	ANSI (COMMIT)
	Teradata (BTET)
Session Collation	ASCII
	EBCDIC
	MULTINATIONAL
	HOST
	CHARSET_COLL
	JIS_COLL
Account and Priority	Dynamic account and reprioritization. You can define performance groups or use one of the following predefined performance groups: • \$R • \$H • \$M • \$L
Date Form	ANSIDATE
	INTEGERDATE
Character Set	Indicates the character set being used by the client. You can view site-installed client character sets from DBC.CharSets or DBC.CharTranslations. The following client character sets are available at all sites: • ASCII • EBCDIC • UTF8 For more information on character sets, see <i>Teradata RDBMS International Character Set Support</i>.

Parameter	Valid Keywords or System Variables
Express Logon (for network-attached clients)	ENABLE
	DISABLE

SQL Flagger

When enabled, the SQL Flagger assists SQL programmers by notifying them of the use of non-ANSI and non-entry level ANSI SQL syntax. Intermediate-level ANSI SQL syntax is flagged as non-entry level ANSI.

Enabling the SQL Flagger can be done regardless of whether you are in ANSI or Teradata mode.

To set the SQL Flagger on or off for interactive SQL, use the `.SET SESSION` command in BTEQ.

To set this level of flagging ...	Set the flag variable to this value ...
None	SQLFLAG NONE
Entry level	SQLFLAG ENTRY
Intermediate level	SQLFLAG INTERMEDIATE

For more detail on using the SQL Flagger, see [“SQL Flagger” on page 1-8](#).

To set the SQL Flagger on or off for embedded SQL, use the `SQLCHECK` or `-sc` and `SQLFLAGGER` or `-sf` options when you invoke the preprocessor.

If you are using SQL in other application programs, see the reference manual for that application for instructions on enabling the SQL Flagger.

Transaction Mode

You can run transactions in either Teradata or ANSI mode and these modes can be set or changed.

To set the transaction mode, use the `.SET SESSION` command in BTEQ.

To set this type of transaction mode ...	Set the variable to this value ...
Teradata	TRANSACTION BTET
ANSI	TRANSACTION ANSI

For more detail on transaction semantics, see *“Transaction Processing” in Teradata RDBMS SQL Reference, Volume 2*.

If you are using SQL in other application programs, see the reference manual for that application for instructions on setting or changing the transaction mode.

Session Collation

Collation of character data is an important and complex task. The Teradata RDBMS provides several named collations. The MULTINATIONAL and CHARSET_COLL collations allow the system administrator to provide collation sequences tailored to the needs of the site.

The collation for the session is determined at logon from the defined default collation for the user. You can change your collation any number of times during the session using the SET SESSION COLLATION statement, but you cannot change your default logon in this way.

Your default collation is assigned via the COLLATION option of the CREATE USER or MODIFY USER statement. This has no effect on any current session, only new logons.

Each named collation can be CASESPECIFIC or NOT CASESPECIFIC. NOT CASESPECIFIC collates lowercase data as if it were converted to uppercase before the named collation is applied.

Collation Name	Description
ASCII	Character data is collated in the order it would appear if converted for an ASCII session, and a binary sort performed
EBCDIC	Character data is collated in the order it would appear if converted for an EBCDIC session, and a binary sort performed.
MULTINATIONAL	The default MULTINATIONAL collation is a two-level collation based on the Unicode collation standard. Your system administrator can redefine this collation to any two-level collation of characters in the LATIN repertoire. 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. This definition is not compatible with MULTINATIONAL collation of non-KANJI1 data. CHARSET_COLL collation is usually a better solution for KANJI1 data. See “ORDER BY Clause” in Chapter 1 – “The SELECT Statement” of <i>Teradata RDBMS SQL Reference, Volume 6</i>. For information on setting up the MULTINATIONAL collation sequence, see “Collation Sequences” in <i>Teradata RDBMS International Character Set Support</i>.
HOST	The default. HOST collation defaults are as follows: - EBCDIC collation for channel-connected systems. - ASCII collation for all others.

Collation Name	Description
CHARSET_COLL	Character data is collated in the order it would appear if converted to the current client character set and then sorted in binary order. CHARSET_COLL collation is a system administrator-defined collation.
JIS_COLL	Character data is collated based on the Japanese Industrial Standards (JIS). JIS characters collate in the following 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 <i>not</i> defined in JIS X 0201, JIS X 0208, or JIS X 0212 in standard order 5 All remaining characters in Unicode standard order

For details, see “SET SESSION COLLATION” in Chapter 1: “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4* for more information.

Account and Priority

You can dynamically downgrade or upgrade the performance group priority for your account.

Priorities can be downgraded or upgraded at either the session or the request level. For more information, see “SET SESSION ACCOUNT” in Chapter 1: “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

Note that changing the performance group for your account changes the account name for accounting purposes because a performance group is part of an account name.

Date Form

You can change the format in which DATE data is imported or exported in your current session.

DATE data can be set to be treated either using the ANSI date format (DATEFORM=ANSIDATE) or using the old Teradata data format (DATEFORM=INTEGERDATE).

For details, see “SET SESSION DATEFORM” in Chapter 1: “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

Character Set

To modify the client character set, use one of the following:

- From BTEQ, use the BTEQ [.] SET SESSION CHARSET '*name*' command.
- In a CLIv2 application, call CHARSET *name*.

where the '*name*' or *name* value, if other than ASCII, EBCDIC, or UTF8, must be a name assigned to the translation codes that define an available character set.

If not explicitly requested, the session default is the character set associated with the logon client. This is either the standard client default, or the character set assigned to the client by the database administrator.

Express Logon

Express Logon improves the logon response time for network-attached client and is especially useful in the OLTP environment where sessions are short-lived.

Enable or disable this mode from the Gateway Global Utility, from the XGTWGLOBAL interface:

In this mode ...	Use this command to enable or disable Express Logon ...
Terminal	ENABLE EXLOGON DISABLE EXLOGON
Window	EXLOGON button (via the LOGON dialog box)

The feature can be enabled or disabled for a particular host group, or for all host groups. For more detail on this feature, see *Teradata RDBMS Utilities*.

For channel-attached clients, see [“Session Pools” on page 1-46](#).

HELP SESSION

The HELP SESSION statement identifies the transaction mode, character set, and collation sequence in effect for the current session. See “HELP SESSION” in the chapter “SQL Help and Database Object Definition Tools: HELP and SHOW” in *Teradata RDBMS SQL Reference, Volume 4* for details.

Session Management

Introduction

Sessions are managed by dividing the session control functions into a series of single small steps that are executed in sequence to implement multi-threaded tasking. This provides concurrent processing of multiple logon and logoff events, which can be any combination of individual users, and one or more concurrent sessions established by one or more users and applications.

Each session is logged on and off via calls to CLIV2 routines or through ODBC or JDBC, which offer a one-step logon-connect function.

Once connected and active, a session can be viewed as a work stream consisting of a series of requests between the client and the Teradata server.

Session Pools

You can establish session pools for channel-connected, but not network-connected, applications. Session pools are collections of sessions that are logged on to the Teradata RDBMS in advance (generally at the time of TDP initialization) for use by applications that require a 'fast path' logon. This capability is particularly advantageous for transaction processing in which interaction with the Teradata RDBMS consists of many single, short transactions.

TDP identifies each session with a unique session number. Teradata identifies a session with a session number, the username of the initiating user, and the logical host identification number of the connection (LAN or mainframe channel) associated with the controlling TDP or mTDP.

Session Reserve

Use the ENABLE SESSION RESERVE command from an OS/390 or VM client to reserve session capacity in the event of a PE failure. To release reserved session capacity, use the DISABLE SESSION RESERVE command.

See *Teradata for Client MVS Installation Guide* and *Teradata for Client VM Installation Guide* for further information.

Session Control

The major functions of session control are session logon and logoff.

Upon receiving a session request, the logon function verifies authorization and returns a yes or no response to the client.

The logoff function terminates any ongoing activity and deletes the session context.

Requests

A request consists of the following components:

- One or more Teradata SQL statements
- Control information

In a session, requests are identified by request numbers that are unique to the controlling TDP.

An application calls CLI routines to build a request and send it to the TDP. Once a session sends a request to the Teradata server, it must receive a response before it can send another request.

If any operation specified by an initiating request fails, the request is backed out, along with any change that was made to the database. In this case, a failure response is returned to the application.

Requests and responses are composed of one or more concatenated parcels.

Parcels

Parcels contain the following components:

- Teradata SQL statements
- Data records
- Operational information
- Error information

An application refers to parcels by including them as arguments in calls to CLI routines. The CLI routines use the parcels to construct a complete request in the output buffer of the address space for an application. When the request completes, the concatenated parcels are moved to the TDP address space.

Teradata processes parcels in order of arrival. If one statement of a request fails, subsequent statements in that request are not processed, and any effects of preceding statements of the request, for ANSI mode, or of the transaction, for Teradata mode, are backed out of the database.

The response to a request is returned to the TDP in parcels. The TDP sends the parcels to the application address space, where it constructs the response in the input buffer.

Requests and responses are transmitted in the form of messages.

Messages

A message consists of the following components:

- Message header
- Request number that identifies the message by request and session
- Data

Message data is either a request to the Teradata RDBMS or a response from the Teradata RDBMS.

Both the TDP and the Teradata RDBMS contain message subsystems that coordinate message traffic. Messages are sent to logical addresses, mailboxes, that are defined for both the client and the Teradata server.

Communication typically occurs in message pairs. The recipient of a message returns a response message to the originator. The message subsystem assembles messages into packets.

Packets

Packets are physical subdivisions of a message that is transmitted in blocks. A packet can contain a part of, or an entire message.

SQL Return Codes

Definition

SQL return codes are signed numeric messages that provide information about the status of a completed executable SQL DML statement.

Status Variables for Receiving SQL Return Codes

ANSI SQL defines two status variables for receiving return codes:

- SQLSTATE
- SQLCODE

SQLCODE is not ANSI SQL-99-compliant. The ANSI SQL-92 standard explicitly deprecates SQLCODE, and the ANSI SQL-99 standard does not define SQLCODE. The ANSI SQL committee recommends that new applications use SQLSTATE in place of SQLCODE.

Teradata defines a third status variable for receiving the number of rows affected by an SQL statement in a stored procedure:

- ACTIVITY_COUNT

Teradata SQL defines a non-ANSI SQL Communications Area (SQLCA) that also has a field named SQLCODE for receiving return codes.

For information on ...	See the ...
• SQLSTATE • SQLCODE • ACTIVITY_COUNT	“Result Code Variables” chapter in <i>Teradata RDBMS SQL Reference, Volume 6</i>
SQLCA	“SQL Communications Area (SQLCA)” appendix in <i>Teradata RDBMS SQL Reference, Volume 6</i>

Exception and Completion Conditions

ANSI SQL defines two categories of conditions that issue return codes:

- Exception conditions
- Completion conditions

Exception Conditions

An exception condition indicates a statement failure.

A statement that raises an exception condition does nothing more than return that exception condition to the application.

There are as many exception condition return codes as there are specific exception conditions.

For more information about exception conditions, see [“SQL Failure Response” on page 1-55](#) and [“SQL Error Response \(ANSI Mode Only\)” on page 1-56](#).

See *Teradata RDBMS Messages* for a complete list of exception condition codes.

Completion Conditions

A completion condition indicates statement success.

There are three categories of completion conditions:

- Successful completion
- Warnings
- No data found

For more information, see [“SQL Statement Responses” on page 1-52](#), [“SQL Success Response” on page 1-53](#), and [“SQL Warning” on page 1-54](#).

A statement that raises a completion condition can take further action such as querying the database and returning results to the requesting application, updating the database, initiating an SQL transaction, and so on.

FOR this type of completion condition ...	The value for this return code is ...	
	SQLSTATE	SQLCODE
Success	'00000'	0
Warning	'01901'	901
	'01800' to '01841'	901
	'01004'	902
No data found	'02000'	100

Return Codes for Stored Procedures

The return code values are different in the case of SQL control statements in stored procedures.

The return codes for stored procedures are as follows:

FOR this type of condition ...	The value for this return code is ...	
	SQLSTATE	SQLCODE
Successful completion	'00000'	0
Warning	SQLSTATE value corresponding to the warning code.	the RDBMS warning code.
No data found or any other Exception	SQLSTATE value corresponding to the error code.	the RDBMS error code.

How an Application Uses SQL Return Codes

An application program or stored procedure tests the status of a completed executable SQL statement to determine its status.

IF the statement raises this type of condition ...	THEN the application or condition handler takes the following remedial action ...
Successful completion	none.
Warning	the statement execution continues. If a warning condition handler is defined in the application, the handler executes.
No data found or any other exception	whatever appropriate action is required by the exception. If an EXIT handler has been defined for the exception, the statement execution terminates. If a CONTINUE handler has been defined, execution continues after the remedial action.

SQL Statement Responses

Response Types

The Teradata RDBMS responds to an SQL request with one of the following condition responses:

- Success response, with optional warning
- Failure response
- Error response (ANSI mode only)

Depending on the type of statement, the RDBMS also responds with one or more rows of data.

Multistatement Responses

A response to a request that contains more than one statement, such as a macro, is not returned to the client until all statements in the request are successfully executed.

How a Response Is Returned to the User

The manner in which the response is returned depends on the interface that is being used.

For example, if an application is using a language preprocessor, then the activity count, warning code, error code, and fields from a selected row are returned directly to the program through its appropriately declared variables.

If the application is a stored procedure, then the activity count is returned directly in the `ACTIVITY_COUNT` status variable.

If you are using BTEQ, then a success, error, or failure response is displayed automatically.

Response Condition Codes

SQL statements also return condition codes that are useful for handling errors and warnings in embedded SQL and stored procedure applications.

For information about SQL response condition codes, see:

- `SQLSTATE`
- `SQLCODE`
- `ACTIVITY_COUNT`

in the “Result Code Variables” chapter in *Teradata RDBMS SQL Reference, Volume 6*.

SQL Success Response

Definition

A success response contains an activity count that indicates the total number of rows involved in the result.

For example, the activity count for a SELECT statement is the total number of rows selected for the response. For a SELECT, COMMENT, or ECHO statement, the activity count is followed by the data that completes the response.

An activity count is meaningful for statements that return a result set, for example:

- SELECT
- INSERT
- UPDATE
- DELETE
- HELP
- SHOW
- EXPLAIN
- CREATE PROCEDURE
- REPLACE PROCEDURE

For other SQL statements, activity count is meaningless.

Example

The following interactive SELECT statement returns the successful response message.

```
SELECT AVG(f1)
FROM Inventory;

*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
Average(f1)
-----
14
```

SQL Warning

Definition

A success or ok response with a warning indicates either that an anomaly has occurred or informs the user about the anomaly and indicates how it can be important to the interpretation of the results returned.

Example

Assume the current session is running in ANSI mode.

If nulls are included in the data for column f1, then the following interactive query returns the successful response message with a warning about the nulls.

```
SELECT AVG(f1) FROM Inventory;
```

```
*** Query completed. One row found. One column returned.  
*** Warning: 2892 Null value eliminated in set function.  
*** Total elapsed time was 1 second.
```

```
Average(f1)  
-----  
14
```

This warning response is not generated if the session is running in Teradata mode.

SQL Failure Response

Definition

A failure response includes a statement number, an error code, and an associated text string describing the cause of the failure.

If one statement in a macro fails, a single failure response is returned to the client, and the results of any previous statements in the transaction are backed out.

Example

The following SELECT statement

```
SELECT * FROM Inventory;;
```

in BTEQ, returns the failure response message:

```
*** Failure 3709 Syntax error, replace the ':' that follows the  
name with a ';'.  
Statement# 1, Info =20  
*** Total elapsed time was 1 second.
```

SQL Error Response (ANSI Mode Only)

Definition

An error response occurs when a query anomaly is severe enough to prevent the correct processing of the request.

In ANSI mode, an error for a request causes the *request* to rollback, and not the entire transaction.

Example 1

The following command returns the error message immediately following.

```
.SET SESSION TRANS ANSI;  
  
*** Error: You must not be logged on .logoff to change the  
SQLFLAG or TRANSACTION settings.
```

Example Scenario

Assume for the next four examples that the user has logged on in ANSI mode.

```
SHOW TABLE inv;  
  
-----  
CREATE MULTiset TABLE MAIN.inv, FALLBACK,  
    NO BEFORE JOURNAL,  
    NO AFTER JOURNAL  
(  
    item INTEGER CHECK ((item >=10) AND (item <= 20) ))  
PRIMARY INDEX (item);
```

Example 2

You insert a value of 12 into the item column of the inv table.

This is valid because the defined integer check specifies that any integer between 10 and 20 (inclusive) is valid.

```
INSERT INTO inv (12);
```

The following results message returns.

```
*** Insert completed. One row added....
```

Example 3

You insert a value of 9 into the item column of the inv table.

This is not valid because the defined integer check specifies that any integer with a value less than 10 is not valid.

```
INSERT INTO inv (9);
```

The following error response returns.

```
***Error 5317 Check constraint violation: Check error in field  
inv.item.
```

Example 4

You commit the current transaction.

```
COMMIT;
```

The following results message returns.

```
*** COMMIT done. ...
```

Example 5

You select all rows from the inv table.

```
SELECT * FROM inv;
```

The following results message returns.

```
*** Query completed. One row found. Cone column returned.
```

```
      item  
-----  
      12
```


Chapter 2:

SQL Data Definition, Control, and Manipulation

This chapter describes the functional families of the SQL language.

Topics include:

- SQL functional families and binding styles
- SQL Data Definition Language (DDL)
- SQL Data Control Language (DCL)
- SQL Data Manipulation Language (DML)
- SQL stored procedures

SQL Functional Families and Binding Styles

Introduction

The SQL language can be characterized in several different ways. This chapter is organized around functional groupings of the components of the language with minor emphasis on binding styles.

Definition: Functional Family

SQL provides facilities for defining database objects, for defining user access to those objects, and for manipulating the data stored within them.

The following list describes the principal functional families of the SQL language.

- SQL Data Definition Language (DDL)
- SQL Data Control Language (DCL)
- SQL Data Manipulation Language (DML)

Some classifications of SQL group the data control language statements with the data definition language statements.

Definition: Binding Style

The ANSI SQL standards do not define the term binding style. The expression refers to a possible method by which an SQL statement can be invoked.

Teradata supports the following SQL binding styles:

- Direct, or interactive
- Embedded SQL
- Stored procedure
- SQL Call Level Interface (as ODBC)
- JDBC

The direct binding style is usually not qualified in this manual set because it is the default style.

Embedded SQL and stored procedure binding styles are always clearly specified, either explicitly or by context. Embedded SQL and stored procedures are described in [“SQL Data Manipulation Language \(DML\)” on page 2-69](#).

For more information, see:

- [“Embedded SQL” on page 2-89](#)
- [“Stored Procedures” on page 2-90](#)

Neither ODBC nor JDBC are further described in the SQL Reference set. See *ODBC Driver for Teradata User Guide* and *Teradata Driver for the JDBC Interface User Guide* for more information.

SQL Data Definition Language (DDL)

Definition

The SQL Data Definition Language, abbreviated DDL, is a subset of the SQL language that consists of all SQL statements that support the definition of database objects.

Purpose of Data Definition Language Statements

Data definition language statements perform the following functions.

- Create, drop, rename, and alter tables
- Create, drop, rename, and replace stored procedures, views and macros
- Create and drop indexes
- Create, drop, and modify users and databases
- Create, drop, alter, rename, and replace triggers
- Create, drop, and set roles
- Create, drop, and modify profiles
- Collect statistics on a column set or index
- Establish a default database
- Comment on database objects
- Set a different collation sequence for the session

Restrictions on Entering DDL Statements

DDL statements cannot be entered as part of a multistatement request.

A DDL statement can be entered as:

- A single statement request.
- The solitary statement, or the last statement, in an explicit transaction (in Teradata mode, one or more requests enclosed by user-supplied BEGIN TRANSACTION and END TRANSACTION statement, or in ANSI mode, one or more requests ending with the COMMIT keyword).
- The solitary statement in a macro.

Successful execution of a DDL statement automatically creates and updates entries in the Data Dictionary.

SQL Data Definition Statements

DDL statements include the following.

- | | | |
|----------------------|-------------------|---------------------|
| • ALTER TABLE | • CREATE VIEW | • MODIFY USER |
| • ALTER TRIGGER | • DATABASE | • RENAME MACRO |
| • COLLECT STATISTICS | • DELETE DATABASE | • RENAME PROCEDURE |
| • COMMENT | • DELETE USER | • RENAME TABLE |
| • CREATE DATABASE | • DROP DATABASE | • RENAME TRIGGER |
| • CREATE HASH INDEX | • DROP HASH INDEX | • RENAME VIEW |
| • CREATE INDEX | • DROP INDEX | • REPLACE MACRO |
| • CREATE JOIN INDEX | • DROP MACRO | • REPLACE PROCEDURE |
| • CREATE MACRO | • DROP PROCEDURE | • REPLACE TRIGGER |
| • CREATE PROCEDURE | • DROP PROFILE | • REPLACE VIEW |
| • CREATE PROFILE | • DROP ROLE | • SET ROLE |
| • CREATE ROLE | • DROP STATISTICS | • SET SESSION |
| • CREATE TABLE | • DROP TRIGGER | • SET TIME ZONE |
| • CREATE TRIGGER | • MODIFY DATABASE | |
| • CREATE USER | • MODIFY PROFILE | |

Further Information

For detailed information about the function, syntax, and usage of Teradata SQL Data Definition statements, see Chapter 1: “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

Tables

Introduction

All tables are defined using the CREATE TABLE statement.

The CREATE TABLE statement specifies a table name, one or more column names, and the attributes of each column.

Definitions

A table is what is referred to in set theory terminology as a relation, from which the expression *relational database* is derived.

Every relational table consists of one row of column headings (more commonly referred to as column names) and zero or more unique rows of data values.

Formally speaking, each row represents what set theory calls a tuple. Each column represents what set theory calls an attribute.

The number of rows (or tuples) in a table is referred to as its cardinality and the number of columns (or attributes) is referred to as its degree or arity.

Creating Indexes For a Table

The CREATE TABLE statement can also be used to define a primary index and one or more secondary indexes. You can define the primary index (and any secondary index) as unique, depending on whether duplicate values are to be allowed in the indexed column set.

You cannot define hash or join indexes using CREATE TABLE. To create these indexes, you must use the CREATE HASH INDEX and CREATE JOIN INDEX statements, respectively.

CREATE TABLE can also specify datablock size, percent freespace, and other physical attributes of the table.

Duplicate Rows in Tables

Though both set theory and common sense prohibit duplicate rows in relational tables, the ANSI standard defines SQL based not on sets, but on bags, or multisets.

A table defined not to permit duplicate rows is called a SET table because its properties are based on set theory, where set is defined as an unordered group of unique elements with no duplicates.

A table defined to permit duplicate rows is called a MULTiset table because its properties are based on a multiset, or bag, model, where bag and multiset are defined as an unordered group of elements that may be duplicates.

The following rules hold for duplicate rows in a table:

IF a table is defined as this type ...	AND the mode is ...	THEN ...
SET	Teradata	duplicate rows are not allowed. If an INSERT will create a duplicate row, an error message is returned. If an INSERT... SELECT will create a duplicate row, any such row is silently ignored. No duplicate row error message is returned.
	ANSI	duplicate rows are not allowed. If an INSERT or INSERT... SELECT will create a duplicate row, an error message is returned.
MULTISET with one or more UNIQUE indexes	Teradata	duplicate rows are not allowed.
	ANSI	If an INSERT or INSERT... SELECT will create a duplicate row, an error message is returned.
MULTISET with no UNIQUE indexes	Teradata	duplicate rows are allowed.
	ANSI	

For more information on duplicate row rules, see “CREATE TABLE” in the chapter “SQL Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4*.

Global Temporary Tables and Volatile Tables

Overview

Teradata supports two types of temporary tables:

- Global temporary (identical to the ANSI Global Temporary Table)
- Volatile

Global temporary tables are useful for storing temporary, intermediate results from multiple queries into working tables that are frequently used by applications. Because storage of such temporary tables as permanent persistent tables incurs a great deal of overhead, temporary tables are a good solution.

Materialized instances of a global temporary table share the following characteristics with volatile tables:

- Private to the session that created them.
- Contents cannot be shared by other sessions.
- Optionally emptied at the end of each transaction using the ON COMMIT PRESERVE/DELETE rows option in the CREATE TABLE statement.
- Activity optionally logged in the transient journal using the LOG/NO LOG option in the CREATE TABLE statement.
- Dropped automatically when a session ends.

Volatile tables are useful for storing temporary results that are to be used only during the time a session persists.

The individual characteristics of global temporary and volatile tables are described with more detail in [“Global Temporary Tables” on page 2-9](#) and [“Volatile Tables” on page 2-15](#).

Global Temporary Tables

Introduction

Global temporary tables allow you to define a table template in the database schema, providing large savings for applications that require a few well known temporary table definitions.

The definition for a global temporary table is persistent and stored in the data dictionary.

Space usage is charged to login user temporary space.

Each user session can materialize as many as 2000 global temporary tables at a time.

How Global Temporary Tables Work

You create the base definition for a global temporary table by using the CREATE TABLE statement and specifying the keywords GLOBAL TEMPORARY to describe the table type.

Once created, the table exists only as a definition. It has no rows and no physical instantiation.

When any application in a session accesses a table with the same name as the defined base table, and the table has not already been materialized in that session, then that table is materialized as a real relation using the stored definition. Because that initial invocation is generally due to an INSERT statement, a temporary table—in the strictest sense—is usually populated immediately upon its materialization.

There are only two occasions when an empty global temporary table is materialized:

- A CREATE INDEX statement is issued on the table.
- A COLLECT STATISTICS statement is issued on the table.

The following table summarizes this information.

WHEN a local instance of a global temporary table is materialized via this statement ...	THEN when it is materialized it is ...
INSERT	populated with data.
CREATE INDEX COLLECT STATISTICS	not populated with data.

Issuing a SELECT, UPDATE, or DELETE on a global temporary table that is not materialized produces a result that is as if the materialized instance is empty.

For example, suppose there are four sessions, Session 1, Session 2, Session 3, and Session 4 and two users, User_1 and User_2.

Consider the scenario in the following two tables:

Step	Session ...	Does this ...	The result is this ...
0	1	The DBA creates a global temporary table definition in the database scheme named globdb.gt1 according to the following CREATE TABLE statement: <pre>CREATE GLOBAL TEMPORARY TABLE globdb.gt1, LOG (f1 INT NOT NULL PRIMARY KEY, f2 DATE, f3 FLOAT) ON COMMIT PRESERVE ROWS;</pre>	The global temporary table definition is created and stored in the database schema.
1	1	User_1 logs on an SQL session and references globdb.gt1 using the following INSERT statement: <pre>INSERT globdb.gt1 (1, 980101, 11.1);</pre>	Session 1 created a local instance of the global temporary table definition globdb.gt1. This is also referred to as a materialized temporary table. The table is populated, immediately upon materialization, with a single row having the following values. f1=1 f2=980101 f3=11.1 This means that the contents of this local instance of the global temporary table definition is <i>not</i> empty when it is created. From this point on, any INSERT/DELETE/UPDATE statement that references globdb.gt1 in Session 1 is mapped to this local instance of the table.

Step	Session ...	Does this ...	The result is this ...
2	2	User_2 logs on an SQL session and issues the following SELECT statement. SELECT * FROM globdb.gt1;	No rows are returned because Session 2 has not yet materialized a local instance of globdb.gt1.
3	2	User_2 issues the following INSERT statement: INSERT globdb.gt1 (2, 980202, 22.2);	Session 2 created a local instance of the global temporary table definition globdb.gt1. The table is populated, immediately upon materialization, with a single row having the following values. f1=2 f2=980202 f3=22.2 From this point on, any INSERT/DELETE/UPDATE statement that references globdb.gt1 in Session 2 is mapped to this local instance of the table.
4	2	User_2 logs again issues the following SELECT statement: SELECT * FROM globdb.gt1;	A single row containing the data (2, 980202, 22.2) is returned to the application.
5	1	User_1 logs off from Session 1.	The local instance of globdb.gt1 for Session 1 is dropped.
6	2	User_2 logs off from Session 2.	The local instance of globdb.gt1 for Session 2 is dropped.

User_1 and User_2 continue their work, logging onto two additional sessions as described in the following table.

Step	Session ...	Does this ...	The result is this ...
1	3	User_1 logs on another SQL session 3 and issues the following SELECT statement: <code>SELECT * FROM globdb.gt1;</code>	No rows are returned because Session 3 has not yet materialized a local instance of globdb.gt1.
2	3	User_1 issues the following INSERT statement: <code>INSERT globdb.gt1 (3, 980303, 33.3);</code>	Session 3 created a local instance of the global temporary table definition globdb.gt1. The table is populated, immediately upon materialization, with a single row having the following values. f1=3 f2=980303 f3=33.3 From this point on, any INSERT/DELETE/UPDATE statement that references globdb.gt1 in Session 3 is mapped to this local instance of the table.
3	3	User_1 again issues the following SELECT statement: <code>SELECT * FROM globdb.gt1;</code>	A single row containing the data (3, 980303, 33.3) is returned to the application.
4	4	User_2 logs on Session 4 and issues the following CREATE INDEX statement: <code>CREATE INDEX (f2) ON TEMPORARY globdb.gt1;</code>	An empty local global temporary table named globdb.gt1 is created for Session 4. This is one of only two cases in which a local instance of a global temporary table is materialized without data. The other would be a COLLECT STATISTICS statement—in this case, the following statement. COLLECT STATISTICS ON TEMPORARY globdb.gt1;
5	4	User_2 issues the following SELECT statement: <code>SELECT * FROM globdb.gt1;</code>	No rows are returned because the local instance of globdb.gt1 for Session 4 is empty.

Step	Session ...	Does this ...	The result is this ...
6	4	User_2 issues the following SHOW TABLE statement: SHOW TABLE globdb.gt1;	CREATE SET GLOBAL TEMPORARY TABLE globdb.gt1, FALLBACK, LOG (f1 INTEGER NOT NULL, f2 DATE FORMAT 'YYYY- MM-DD', f3 FLOAT) UNIQUE PRIMARY INDEX (f1) ON COMMIT PRESERVE ROWS;
7	4	User_2 issues the following SHOW TEMPORARY TABLE statement: SHOW TEMPORARY TABLE globdb.gt1;	CREATE SET GLOBAL TEMPORARY TABLE globdb.gt1, FALLBACK, LOG (f1 INTEGER NOT NULL, f2 DATE FORMAT 'YYYY- MM-DD', f3 FLOAT) UNIQUE PRIMARY INDEX (f1) INDEX (f2) ON COMMIT PRESERVE ROWS; Note that this report indicates the new index f2 that has been created for the local instance of the temporary table.

With the exception of a few options (see “CREATE TABLE” in Chapter 1 – “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4* for an explanation of the features not available for global temporary base tables), materialized temporary tables have the same properties as permanent tables.

Once a global temporary table definition is materialized in a session, all further references to the table are made to the materialized table. No additional copies of the base definition are materialized for the session. This global temporary table is defined for exclusive use by the session whose application materialized it.

Materialized global temporary tables differ from permanent tables in the following ways:

- They are always empty when first materialized.
- Their contents cannot be shared by another session.
- The contents can optionally be emptied at the end of each transaction.
- The materialized table is dropped automatically at the end of each session.

Limitations

The following CREATE TABLE options are not permitted for global temporary tables:

- PARTITION BY
- Permanent journaling
- Referential integrity constraints

This means that a temporary table cannot be the referencing or referenced table in a referential integrity constraint.

References to global temporary tables are not permitted in FastLoad, MultiLoad, or FastExport.

Archive, Restore, and TableRebuild operate on base temporary tables only.

Non-ANSI Extensions

Transient journaling options on the global temporary table definition are permitted using the CREATE TABLE statement.

You can modify the transient journaling and ON COMMIT options for global temporary tables using the ALTER TABLE statement.

Privileges Required

To materialize a temporary table, you must have the appropriate privilege as required by the statement materializing the table.

No access logging is performed on materialized temporary tables, so no access log entries are generated.

Volatile Tables

Creating Volatile Tables

Neither the definition nor the contents of a volatile table persist across a system restart. You must use CREATE TABLE with the VOLATILE keyword to create a new volatile table each time you start a session in which it is needed.

What this means is that you can create volatile tables as you need them. Being able to create a table quickly provides you with the ability to build scratch tables whenever you need them. Any volatile tables you create are dropped automatically as soon as your session logs off.

Volatile tables are always created in the login user space, regardless of the current default database setting. That is, the database name for the table is the login user name. Space usage is charged to login user spool space. Each user session can materialize as many as 1000 volatile tables at a time.

Limitations

The following CREATE TABLE options are not permitted for volatile tables:

- Permanent journaling
- Referential integrity constraints
This means that a volatile table cannot be the referencing or referenced table in a referential integrity constraint.
- Check constraints
- Compressed columns
- DEFAULT clause
- TITLE clause
- Named indexes
- PARTITION BY clause

References to volatile tables are not permitted in FastLoad or MultiLoad.

For more information, see “CREATE TABLE” in Chapter 1: “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

Non-ANSI Extensions

Volatile tables are not defined in ANSI.

Privileges Required

To create a volatile table, you do not need any privileges.

No access logging is performed on volatile tables, so no access log entries are generated.

Volatile Table Maintenance Among Multiple Sessions

Volatile tables are private to a session. This means that you can log on multiple sessions and create volatile temporary tables with the same name in each session.

However, at the time you create a volatile table, the name must be unique among all permanent and global temporary table names in the database that has the name of the login user.

For example, suppose you log on two sessions, Session 1 and Session 2. Assume the default database name is your login user name. Consider the following scenario:

Stage	In Session 1, you ...	In Session 2, you ...	The result is this ...
1	Create a volatile table named VT1.	Create a volatile table named VT1.	Each session creates its own copy of volatile table VT1 using your login user name as the database.
2	Create a permanent table with an unqualified table name of VT2.		Session 1 creates a permanent table named VT2 using your login user name as the database.
3		Create a volatile table named VT2.	Session 2 receives a CREATE TABLE error, because there is already a permanent table with that name.
4	Create a volatile table named VT3.		Session 1 creates a volatile table named VT3 using your login user name as the database.
5		Create a permanent table with an unqualified table name of VT3.	Session 2 creates a permanent table named VT3 using your login user name as the database. Because a volatile table is known only to the session that creates it, a permanent table with the same name as the volatile table VT3 in Session 1 can be created as a permanent table in Session 2.

Stage	In Session 1, you ...	In Session 2, you ...	The result is this ...
6	Insert into VT3.		Session 1 references volatile table VT3. Note: Volatile tables take precedence over permanent tables in the same database in a session. Because Session 1 has a volatile table VT3, any reference to VT3 in Session 1 is mapped to the volatile table VT3 until it is dropped (see Step 10). On the other hand, in Session 2, references to VT3 remain mapped to the permanent table named VT3.
7		Create volatile table VT3.	Session 2 receives a CREATE TABLE error for attempting to create the volatile table VT3 because of the existence of that permanent table.
8		Insert into VT3.	Session 2 references permanent table VT3.
9	Drop VT3.		Session 2 drops volatile table VT3.
10	Select from VT3.		Session 1 references the permanent table VT3.

Primary Keys and Foreign Keys

Definition: Primary Key

A *primary key* is a field (or combination of fields) in a table that is unique for each row in that table.

Functions of a Primary Key

The function of a primary key is to identify uniquely each row in the table. Uniqueness requires the following rules for the values defining a primary key for a table.

- Must be unique
- Cannot change
- Cannot be null

Because Teradata uses a unique primary or secondary index to enforce a primary key, the primary key can affect how Teradata distributes and retrieves rows. For more information, see [“Primary Indexes” on page 2-26](#) and [“Secondary Indexes” on page 2-31](#).

Definition: Foreign Key

A *foreign key* is a field (or combination of fields) in a table that is also the primary key in one or more additional tables in the same database.

Foreign keys provide a mechanism for linking related tables based on key values.

Keys and Referential Integrity

Teradata uses primary and foreign keys to maintain referential integrity. For additional information, see [“Referential Integrity” on page 2-57](#).

Keys Versus Indexes

The following table summarizes the differences between keys and indexes using the primary key and primary index for purposes of comparison:

Primary Key	Primary Index
Important component of logical data model.	Not used in logical data model.
Used to maintain referential integrity.	Used to distribute and retrieve data.
Must be unique (uniquely identifies each row).	Can be unique or nonunique.
Values can never be changed.	Values can be changed.
Cannot be null.	Can be null.
Does not imply access path.	Defines the most common access path.
Not required for physical table definition.	Mandatory for physical table definition.

Indexes

Definition

An index is a mechanism that can be used by the SQL query optimizer to make table access more performant. Indexes enhance data access by providing a more-or-less direct path to stored data and avoiding the necessity to perform full table scans to locate the small number of rows you typically want to retrieve or update.

The Teradata parallel architecture makes indexing an aid to better performance, not a crutch necessary to ensure adequate performance. Full table scans are not something to be feared in the Teradata environment. This means that the sorts of unplanned, ad hoc queries that characterize the data warehouse process, and that often are not supported by indexes, perform very effectively for Teradata using full table scans.

The classic index for a relational database is itself a file made up of rows having two parts:

- A (possibly unique) data field in the referenced table.
- A pointer to the location of that row in the base table (if the index is unique) or a pointer to all possible locations of the row (if the index is nonunique).

Because the Teradata RDBMS is a massively parallel architecture, it requires a more efficient means of distributing and retrieving its data. One such method is *hashing*. With the exception of hash and join indexes, all Teradata indexes are based on row *hash values* rather than raw table column values, even though secondary, hash, and join indexes can be stored in order of their values to make them more useful for satisfying range conditions.

Another method is *partitioning*, which allows rows with the same value of a partitioning expression within a table to be physically grouped together in partitions.

Selectivity of Indexes

An index that retrieves many rows is said to have *low selectivity*.

An index that retrieves few rows is said to be *highly selective*.

The more strongly selective an index is, the more useful it is. In some cases, it is possible to link several weakly selective nonunique secondary indexes together by bit mapping them. The result is effectively a strongly selective index and a dramatic reduction in the number of table rows that must be accessed.

See [“NUSI Bit Mapping” on page 2-34](#) for further discussion of linking weakly selective secondary indexes into a strongly selective unit using bit mapping.

Row Hash and RowID

Teradata table rows are self-indexing with respect to their primary index and so require no additional storage space. When a row is inserted into a table, the relational database manager stores the 32-bit row hash value of the primary index with it.

Because row hash values are not necessarily unique, the relational database manager also generates a unique 32-bit numeric value (called the Uniqueness Value) that it appends to the row hash value, forming a unique RowID. This RowID makes each row in a table uniquely identifiable and ensures that hash collisions do not occur.

If a table is defined with a partitioned primary index (PPI), the RowID also consists of the partition number to which the row was assigned. For more information on PPIs, see [“Partitioned and Non-Partitioned Primary Indexes” on page 2-23](#).

The first row having a specific row hash value is always assigned a uniqueness value of 0. Additional table rows having the same row hash value are assigned uniqueness values in a numerically increasing order and are stored on disk sorted in the ascending order of RowID.

To determine the next uniqueness value for a row hash, the relational database manager reads through all the rows in the table belonging to that row hash to find the current high value, which is then incremented by 1.

Uniqueness values are not reused except for the special case in which the highest valued row within a row hash is deleted from a table.

A RowID for a row might change, for instance, when a primary index or partitioning column is changed, or when there is complex update of the table.

Index Hash Mapping

Rows are distributed across the AMPs using a hashing algorithm that computes a row hash value based on the primary index. The row hash is a 32-bit value containing a 16-bit destination selection word and a 16-bit remainder.

Teradata relational databases have 65636 hash buckets. The hash buckets are distributed as evenly as possible among the AMPs on a system.

Teradata maintains a *hash map*—an index of which hash buckets live on which AMPs—that it uses to determine whether rows belong to an AMP based on their row hash values. Row assignment is performed in a manner that ensures as equal a distribution as possible among all the AMPs on a system.

Advantages of Indexes

The intent of indexes is to lessen the time it takes to retrieve rows from a database. The faster the retrieval, the better.

Disadvantages of Indexes

Perhaps not so obvious is the disadvantage of using indexes.

- They must be updated every time a row is updated, deleted, or added to a table.
This is only a consideration for secondary indexes in the Teradata environment. The more indexes you have defined for a table, the bigger the potential update downside becomes.
Because of this, secondary indexes are rarely appropriate for OLTP situations.
- All Teradata secondary indexes are stored in subtables and exert a burden on system storage space.
- When FALLBACK is defined for a table, a further storage space burden is created because secondary index subtables are always duplicated whenever FALLBACK is defined for a table.

For this reason, it is extremely important to use EXPLAIN statements to determine optimum data manipulation statement syntax and index usage before putting statements and indexes to work in a production environment.

Teradata Index Types

Teradata provides four different index types:

- Primary index
All Teradata tables require a primary index because the system partitions tables on their primary indexes. Primary indexes can be:
 - Unique or nonunique
 - Partitioned or non-partitioned
- Secondary index
Secondary indexes can be unique or nonunique.
- Join index (JI)
- Hash index

Unique Indexes

A unique index, like a primary key, has a unique value for each row in a table.

Teradata defines two different types of unique index.

- Unique primary index (UPI)
UPIs provide optimal data distribution and are typically assigned to the primary key for a table. When a NUPI makes better sense for a table, then the primary key is frequently assigned to be a USI.
- Unique secondary index (USI)
USIs guarantee that each complete index value is unique, while ensuring that data access based on it is always a two-AMP operation.

You should define a unique index for any table you create. As a rule of thumb, any time you create a table having a nonunique primary index, then you should also create a unique secondary index on the primary key for that table.

Nonunique Indexes

A nonunique index does not require its values to be unique. There are occasions when a nonunique index is the best choice as the primary index for a table.

How evenly a NUPI distributes the rows for a table across the AMPs depends on the relative uniqueness of the column data comprising the primary index. As a rule of thumb, you should never assign a NUPI to a column (or columns) for which much more than 100 rows in the table are likely to have the same NUPI value.

NUSIs are also very useful for many decision support situations.

Partitioned and Non-Partitioned Primary Indexes

Primary indexes can be partitioned or non-partitioned.

A *non-partitioned primary index* (NPPI) is the traditional primary index by which rows are assigned to AMPs.

A *partitioned primary index* (PPI) allows rows to be partitioned, based on some set of columns, on the AMP to which they are distributed, and ordered by the hash of the primary index columns within the partition.

A PPI can be used to improve query performance through partition elimination. A PPI provides a useful alternative to an NPPI for executing range queries against a table, while still providing efficient join and aggregation strategies on the primary index.

Join Indexes

Join indexes are used for queries against table joins that are performed frequently enough that they can be predefined as a join index.

Teradata join indexes can be defined in the following general ways.

- Simple or aggregate.
- Single- or multitable.
- Hash-ordered or value-ordered.

For additional information, see [“Join Indexes” on page 2-39](#).

Hash Indexes

Hash indexes are used for the same purposes as are single-table join indexes, and are less complicated to define.

For additional information, see [“Hash Indexes” on page 2-44](#).

Using EXPLAIN and Teradata Index Wizard to Determine the Usefulness of Indexes

One important thing to remember is that the use of indexes by the optimizer is *not* under user control in a relational database management system. That is, the only references made to indexes in the SQL language concern their definition and *not* their use. The SQL data manipulation language statements do not provide for any specification of indexes.

There are several implications of this behavior.

- First, it is very important to collect statistics regularly to ensure that the optimizer has access to current information about how to best optimize any query or update made to the database.

For additional information concerning collecting and maintaining accurate database statistics, see “COLLECT STATISTICS” in Chapter 1 – “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

- Second, it is even more important to build your queries and updates in such a way that you know their performance will be most optimal.

Apart from good logical database design, the only way to ensure that you are accessing your data in the most efficient manner possible is to use the EXPLAIN modifier to try out various candidate queries or updates and to note which indexes are used by the optimizer in their execution (if any) as well as examining the relative length of time required to complete the operation.

There are several methods you can use to determine optimal sets of secondary indexes tailored to particular application workloads:

- Teradata Index Wizard
- EXPLAIN reports

The Teradata Index Wizard client utility provides a method of determining optimum secondary indexes for a given SQL statement workload automatically and then verifying that the proposed indexes actually produce the expected performance enhancements.

See the following references for more information about the Teradata Index Wizard:

- *Teradata Index Wizard User Guide*
- The chapter “Database Foundations for the Teradata Index Wizard” in *Teradata RDBMS SQL Reference, Volume 2*

You can produce and analyze EXPLAIN reports using either the Visual EXPLAIN client utility or the SQL EXPLAIN statement modifier.

EXPLAIN statement output provides you with the following basic information:

- The step-by-step access method the optimizer would use to execute the specified data manipulation statement given the current set of table statistics it has to work with.
- The *relative* time it would take to perform the data manipulation statement. While you cannot rely on the reported statement execution time as an absolute, you *can* rely on it as a relative means for comparison with other candidate data manipulation statements against the same tables with the same statistics defined.

Extensive information about how to use the EXPLAIN modifier appears in the description of the EXPLAIN statement modifier in the chapter “SQL Data Manipulation Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 6*.

See *Teradata Visual Explain Getting Started Guide* for information on how to use the Visual EXPLAIN client utility.

For additional performance-related information about how to use the access and join plan reports produced by EXPLAIN to optimize the performance of your databases, see:

- *Teradata RDBMS Database Design*
- *Teradata RDBMS Performance Optimization*

Primary Indexes

Introduction

The primary index for a table controls the distribution of the data for that table across the AMPs as well as its retrieval. Both distribution and retrieval of the data is controlled using the Teradata hashing algorithm (see [“Row Hash and RowID” on page 2-21](#) and [“Index Hash Mapping” on page 2-21](#)).

If the primary index is defined as a *partitioned* primary index (PPI), the data is partitioned, based on some set of columns, on each AMP, and ordered by the hash of the primary index columns within the partition.

Data accessed based on a primary index is always a one-AMP operation because a row and its index are stored on the same AMP. This is true whether the primary index is unique or nonunique, and whether it is partitioned or non-partitioned.

Tables Require a Primary Index

All Teradata tables require a primary index.

To create a primary index, use the CREATE TABLE statement.

If you do not assign a primary index explicitly when you create a table, Teradata assigns a primary index, based on the following rules:

WHEN a CREATE TABLE statement defines a ...			THEN Teradata selects ...						
Primary Index	Primary Key	Unique Column Constraint							
No	Yes	No	the primary key column set to be a UPI.						
No	No	Yes	the first column having a UNIQUE constraint to be a UPI.						
No	Yes	Yes	the primary key column set to be a UPI.						
No	No	No	the first column defined for the table to be its primary index by default. <table><tr><th>WHEN the table is ...</th><th>THEN the primary index is defined ...</th></tr><tr><td>a single column SET table</td><td>as a UPI.</td></tr><tr><td>anything else</td><td>as a NUPI.</td></tr></table>	WHEN the table is ...	THEN the primary index is defined ...	a single column SET table	as a UPI.	anything else	as a NUPI.
WHEN the table is ...	THEN the primary index is defined ...								
a single column SET table	as a UPI.								
anything else	as a NUPI.								

Uniform Distribution of Data and Optimal Access Considerations

When choosing the primary index for a table, there are two essential factors to keep in mind: uniform distribution of the data and optimal access.

With respect to uniform data distribution, consider the following factors:

- The more distinct the primary index values, the better.
- Rows having the same primary index value are distributed to the same AMP.
- Parallel processing is more efficient when table rows are distributed evenly across the AMPs.

With respect to optimal data access, consider the following factors:

- Choose the primary index on the most frequently used access path.
For example, if rows are generally accessed by a range query, consider defining a partitioned primary index on the table that creates a useful set of partitions.
If the table is frequently joined with a specific set of tables, consider defining the primary index on the column set that is typically used as the join condition.
- Primary index operations must provide the full primary index value.
- Primary index retrievals on a single value are always one-AMP operations.

While it is true that the columns you choose to be the primary index for a table are often the same columns that define the primary key, it is also true that primary indexes often comprise fields that are neither unique nor components of the primary key for the table.

Unique and Nonunique Primary Index Considerations

Use the following guidelines and performance considerations for selecting a unique or a nonunique column set as the primary index for a table.

- Choose columns for the primary index based on the selection set most frequently used to retrieve rows from the table even when that set is not unique (if and only if the values provide uniform distribution to their hash values and to the AMPs).
- Choose columns for the primary index that distribute table rows evenly across the AMPs. The more singular the values for a column, the more optimal their distribution.
- Base the column selection on an equality search. For equality constraints *only*, the system hashes directly to the stored row that satisfies the condition.

- Consider the kind of primary key:

For tables with this kind of primary key ...	You generally assign the primary index to ...
Single column	the primary key. This is referred to as a Unique Primary Index (UPI).
Multicolumn	one of the foreign key components of the primary key. This is referred to as a Non Unique Primary Index (NUPI).

- Generally speaking, if you do not assign the primary index to the primary key for a table, then you should define the primary key as the Unique Secondary Index (USI) for the table.
- Consider whether access involves multiple rows or a spool file or both:

UPI-based access ...	WHILE NUPI-based access ...
at most involves one row	can involve multiple rows.
does not require a spool file	often creates a spool file.

- Duplicate NUPI values are always stored on the same AMP and in the same data block (for a partition, if partitioned) if possible.
- NUPI retrieval only requires one I/O operation when the rows are stored in the same data block.
- PPI retrieval may require one I/O operation for each partition.
- Consider the type of value range seen when using predicates in a WHERE clause:

This type of value range ...	Seen when using this predicate in a WHERE clause ...	Results in this kind of retrieval action ...
Implicit	BETWEEN	full table scan, irrespective of any indexes defined for the table. The exception is PPI tables, where partition elimination can be exploited.
Explicit	IN	individual row hashing.

Partitioning Considerations

The decision to define a Partitioned Primary Index (PPI) for a table depends on how its rows are most frequently accessed. PPIs are designed to optimize range queries while also providing efficient primary index join strategies. For range queries, only rows of the qualified partitions need to be accessed.

PPI increases query efficiency by avoiding full table scans without the overhead and maintenance costs of secondary indexes.

Various partitioning strategies are possible:

- For some applications, defining the partitions such that each has approximately the same number of rows might be an effective strategy.
- For other applications, it might be desirable to have a varying number of rows per partition. For example, more frequently accessed data (such as for the current year) might be divided into finer partitions (such as weeks) but other data (such as previous years) may have coarser partitions (such as months or multiples of months).
- Alternatively, it might be important to define each range with equal width, even if the number of rows per range varies.

The most important factors for PPIs are accessibility and maximization of partition elimination. In all cases, it is critical for parallel efficiency to define a primary index that distributes the rows of the table fairly evenly across the AMPs.

For more information on partitioning considerations, see *Teradata RDBMS Database Design*.

Primary Index Summary

Teradata SQL primary indexes have the following properties.

- Defined with the CREATE TABLE data definition statement. CREATE INDEX is used *only* to create secondary indexes.
- Modified with the ALTER TABLE data definition statement.
- If you do not explicitly define a primary index, then CREATE TABLE automatically assigns one.
- If the primary index is not defined explicitly as unique, then the definition defaults to non-unique.
- Can be composed of as many as 64 columns.
- A maximum of one can be defined per table.
- Can be partitioned or non-partitioned.
- Can be unique or non-unique.

Note that a partitioned primary index can only be unique if all the partitioning columns are also included as primary index columns.

- Controls data distribution and retrieval using the Teradata hashing algorithm.
- Improves performance when used correctly in the WHERE clause of an SQL data manipulation statement to perform the following actions.
 - Single-AMP retrievals
 - Joins between tables with identical primary indexes, the optimal scenario.

For More Information About Primary Indexes

Consult the following books for more detailed information on using primary indexes to enhance the performance of your databases.

- *Teradata RDBMS Database Design*
- *Teradata RDBMS Performance Optimization*

Secondary Indexes

Introduction

Secondary indexes are never required for Teradata tables, but they can often improve system performance. Secondary indexes are less frequently used by the optimizer to select rows from a table than the primary index for that table.

You create secondary indexes *explicitly* using the CREATE TABLE and CREATE INDEX statements. Teradata can *implicitly* create unique secondary indexes; for example, when you use a CREATE TABLE statement that specifies a primary index, Teradata implicitly creates unique secondary indexes on column sets that you specify using PRIMARY KEY or UNIQUE constraints.

Creating a secondary index causes the Teradata RDBMS to build a separate internal subtable to contain the index rows, thus adding another set of rows that requires updating each time a table row is inserted, deleted, or updated.

Nonunique secondary indexes (NUSIs) can be specified as either hash-ordered or value-ordered. Value-ordered NUSIs are limited to a single numeric-valued sort key whose size is four or fewer bytes.

Secondary index subtables are also duplicated whenever a table is defined with FALLBACK.

After the table is created and usage patterns have developed, additional secondary indexes can be defined with the CREATE INDEX statement.

Unique Versus Nonunique Secondary Indexes

The RDBMS processes USIs and NUSIs very differently, as summarized in the following table, which is based on the following simple SQL statements.

- For the build portion of a USI, the statement is

```
CREATE UNIQUE INDEX (customer_number)
ON customer_table;
```
- For the build portion of a NUSI, the statement is

```
CREATE INDEX (customer_name)
ON customer_table;
```
- For the access portion of a USI, the statement is

```
SELECT *
FROM customer_table
WHERE customer_number=12;
```
- For the access portion of a NUSI, the statement is

```
SELECT *
FROM customer_table
WHERE customer_name = 'SMITH';
```

Build		Access	
USI	NUSI	USI	NUSI
Each AMP accesses its subset of the base table rows.		The supplied index value hashes to the corresponding secondary index row.	A message containing the secondary index value is broadcast to every AMP.
Each AMP copies the secondary index value and appends the RowID for the base table row.	Each AMP builds a spool file containing each secondary index value found followed by the RowID for the row it came from.	The retrieved base table RowID is used to access the specific data row.	For a hash-ordered NUSI, each AMP creates a local row hash and uses it to access its portion of the index subtable to see if a corresponding row exists. Value-ordered NUSI index subtable values are scanned only for the range of values specified by the query.
Each AMP creates a Row Hash on the secondary index value and puts all three values onto the BYNET.	For hash-ordered NUSIs, each AMP sorts the RowIDs for each secondary index value into ascending order. For value-ordered NUSIs, the rows are sorted by NUSI value order.	The process is complete. This is typically a two-AMP operation.	If an index row is found, the AMP uses the RowID or value order list to access the corresponding base table rows.
The appropriate AMP receives the data and creates a row in the index subtable.	For hash-ordered NUSIs, each AMP creates a row hash value for each secondary index value on a local basis and creates a row in its portion of the index subtable. For value-ordered NUSIs, storage is based on NUSI value rather than the row hash value for the secondary index. Each row contains one or more RowIDs for the index value.		The process is complete. This is always an all-AMPs operation.

Note: The NUSI will not be used if the estimated number of rows to be read in the base table is equal to or greater than the estimated number of data blocks in the base table; in this case, a full table scan is done, or, if appropriate, partition scans are done.

NUSIs and Covering

The Optimizer aggressively pursues NUSIs when they cover a query. Covered columns can be specified anywhere in the query, including the select list, the WHERE clause, aggregate functions, GROUP BY clauses, expressions, and so on. Presence of a WHERE condition on each indexed column is *not* a prerequisite for using a NUSI to cover a query.

Value-Ordered NUSIs

Value-ordered NUSIs are very efficient for range conditions. Because the NUSI rows are sorted by data value, it is possible to search only a portion of the index subtable for a given range of key values.

Value-ordered NUSIs have the following limitations.

- The sort key is limited to a single numeric column.
- The sort key column must be four or fewer bytes.

The following query is an example of the sort of SELECT statement for which value-ordered NUSIs were designed.

```
SELECT *
FROM Orders
WHERE o_date BETWEEN DATE '1998-10-01' AND DATE '1998-10-07';
```

Dual Secondary Indexes and Composites

Dual secondary indexes can be a fruitful choice provided that their combination is strongly selective (the number of rows returned is a small fraction of the total number of table rows).

For example, consider the following query:

```
SELECT last_name, first_name, salary_amount
FROM employee
WHERE department_number = 500
AND job_code = 2147;
```

The conditions have secondary indexes defined on them.

```
CREATE INDEX (department_number) ON EMPLOYEE;
CREATE INDEX (job_code) ON EMPLOYEE;
```

IF ...	THEN the Optimizer ...
both indexes are only weakly selective	uses bit mapping.
one index is more selective	uses it to select the rows and then uses the more weakly selective index to qualify the accessed rows.

IF ...	THEN the Optimizer ...
both columns are indexed	probably specifies a full table scan of the index subtables, calling for a non-duplicate spool file of the RowIDs, and then uses the result to access the base table.
one column is indexed	always specifies a full table scan.

Because OR conditions specify separate set selections, this query might execute faster if done using a UNION operation instead.

ANDed expressions in a WHERE clause represent a set selection.

A composite set selection can reference any number of indexes, including none and is used by the Optimizer only when values for all components of the index are supplied in the condition.

When an expression references the primary index, then it is used to access the table and any remaining conditions are applied to the intermediate returned result set for additional selectivity.

The following table explains how the Optimizer selects secondary indexes in the same scenario.

The Optimizer specifies ...	WHEN ...
a single secondary index	its selectivity is strong (returning a small percentage of the table rows).
multiple secondary indexes	their combined selectivity is strong.

NUSI Bit Mapping

Bit mapping is a technique used by the Optimizer to effectively link several weakly selective indexes in a way that creates a result that drastically reduces the number of base rows that must be accessed to retrieve the desired data.

When WHERE conditions are connected by a logical OR, the Optimizer typically performs a full table scan and applies the ORd conditions to every row without considering an index.

IF all conditions are ANDed together and ...	AND their composite selectivity is ...	THEN ...
one index is strongly selective		the Optimizer selects it alone and applies the remaining constraints to the selected rows as residual constraints.

IF all conditions are ANDed together and ...	AND their composite selectivity is ...	THEN ...
all of the indexes are weakly selective	also weakly selective	the Optimizer performs a full table scan and applies the conditions to every row.
all of the indexes are weakly selective	strongly selective	the Optimizer can instruct each AMP to construct bit maps to determine which RowIDs their local NUSI rows have in common and then access just those rows, applying the conditions to them exclusively.

For example, consider the following SELECT statement with three WHERE conditions, each of which has a weakly selective NUSI defined on its column.

```
SELECT *
FROM employee_table
WHERE column_1 = 'condition_1'
AND column_2 = 'condition_2'
AND column_3 = 'condition_3';
```

Suppose that the index on column_1 has 30 percent selectivity, the index on column_2 50 percent selectivity, and the index on column_3 70 percent selectivity. The individual selectivity of these indexes is very weak.

If the overlap among their selectivities is such that the effect of combining their relative weaknesses results in a significant strength, then the Optimizer will instruct the AMPs to use bit mapping to do just that, and the end result is vastly improved performance of the retrieval in comparison with a full table scan.

The method for computing NUSI bit maps is described in the following table (using the previous example for the scenario). The actions occur concurrently on all AMPs.

Stage	Process
1	The literals condition_1, condition_2, and condition_3 are used to access the corresponding rows from the index subtables.
2	For $n-1$ indexes, each AMP creates separate bit maps of all 0 values. In this case, $n-1$ is 2 because there are three NUSIs defined, one for each row in the WHERE clause.
3	Using a proprietary algorithm, the AMPs equate each base table RowID in a NUSI row with a single bit in the map, and each map bit is turned ON.
4	The resulting bit maps are ANDed together to form the final bit map.

Stage	Process	
5	Each base table RowID in the remaining NUSI is equated to a single bit in the final bit map, and the state of that map is checked.	
	IF the map bit is ...	THEN ...
	OFF	no action is taken and the next map bit is checked.
	ON	the RowID is used to access the table row and all conditions are applied to that row.

For more detailed information about composite indexes, NUSI bit mapping, and other aspects of index selection see *Teradata RDBMS Database Design*.

To determine whether bit mapping is being used for your indexes, use the EXPLAIN statement and examine the reports it produces.

Also see “EXPLAIN Modifier” and “EXPLAIN and Indexed Access” in the chapter “SQL Data Manipulation Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 6*.

Secondary Index Summary

Teradata SQL secondary indexes have the following properties.

- Can enhance the speed of data retrieval.
Because of this, secondary indexes are most useful in decision support applications.
- Do not affect data distribution.
- Can be a maximum of 32 defined per table.
- Can be composed of as many as 64 columns (except for value-ordered NUSIs, which are restricted to a single numeric column of 4 or fewer bytes).
- Can be created or dropped dynamically as data usage changes or if they are found not to be useful for optimizing data retrieval performance.
- Require additional disk space to store subtables.
- Require additional I/Os on INSERTs and DELETEs.
Because of this, secondary indexes might not be as useful in OLTP applications.
- Should not be defined on columns whose values change frequently.
- Should not define individual secondary indexes if typical queries access more than 10 percent of the rows in the table.
- Should not include columns that do not enhance selectivity.
- Should not use composite secondary indexes when multiple single column indexes and bit mapping might be used instead.
- Composite secondary indexes is useful if it reduces the number of rows that must be accessed.

- The Optimizer does not use composite secondary indexes unless there are explicit values for each column in the index.
- Most efficient for selecting a small number of rows.
- Can be unique or non-unique.
- NUSIs can be hash-ordered or value-ordered.
- Cannot be partitioned, but can be defined on a table with a partitioned primary index.

USI Summary

Unique secondary indexes have the following properties.

- Guarantee that each complete index value is unique.
- Any access is a two-AMP operation.

NUSI Summary

Nonunique secondary indexes have the following properties.

- Useful for locating rows having a specific (typically concatenated) value in the index.
- Can be hash-ordered or value-ordered.
Value-ordered NUSIs are particularly useful for enhancing the performance of range queries.
- Any access is an all-AMP operation.

For More Information About Secondary Indexes

See “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4* under “CREATE TABLE” and “CREATE INDEX” for more information.

Also consult the following manuals for more detailed information on using secondary indexes to enhance the performance of your databases.

- *Teradata RDBMS Database Design*
- *Teradata RDBMS Performance Optimization*

Full Table Scans

Introduction

If indexed or partitioned access is not suitable for a query, even when the results are further qualified using a WHERE clause, the result is a full table scan.

If you do not specify a WHERE clause in your query, then a full table scan is *always* used by the RDBMS to access the data.

Definition

A full table scan is a retrieval mechanism in which all rows in a table are touched.

Full table scans are always all-AMPs operations. Because of the architecture used by the Teradata RDBMS, full table scans are always an efficient procedure, and their optimization is scalable based on the number of AMPs defined for the system.

How a Full Table Scan Accesses Rows

Because full table scans necessarily touch every row on every AMP, they do not use the following mechanisms for locating rows.

- Hashing algorithm and hash map
- Primary indexes
- Secondary indexes or their subtables
- Partitioning

Instead, a full table scan uses the file system tables known as the Master Index and Cylinder Index to locate each data block. Each row within a data block is located by a forward scan.

Because rows from different tables are never mixed within the same data block and because rows never span blocks, an AMP can scan up to 128K bytes of the table on each block read, making a full table scan a very efficient operation. Cylinder reads can also increase the efficiency.

For more information on full table scans, see [“Full Table Scan” on page 2-81](#) and [“Access Types” on page 2-82](#).

For information on cylinder reads, see *Teradata RDBMS Database Administration*.

Join Indexes

Introduction

Join indexes are not indexes in the usual sense of the word. They are file structures designed to permit queries (join queries in the case of multitable join indexes) to be resolved by accessing the index instead of having to access and join their underlying base tables.

You can use join indexes to:

- Define a prejoin table on frequently joined columns (with optional aggregation) without denormalizing the database.
- Create a full or partial replication of a base table with a primary index on a foreign key column table to facilitate joins of very large tables by hashing their rows to the same AMP as the large table.
- Define a summary table without denormalizing the database.

You can define a join index on one or several tables.

There are two types of join index, each of which can also be simple or aggregated.

Join Index	Description
Multitable	An index table that stores and maintains the joined rows of two or more tables and, optionally, aggregates selected columns. Multitable join indexes are for join queries that are performed frequently enough to justify defining a prejoin on the joined columns.
Single-table	Very useful for resolving joins on large tables without having to redistribute the joined rows across the AMPs. Single-table join indexes facilitate joins by hashing a frequently joined subset of base table columns to the same AMP as the table rows to which they are frequently joined. This enhanced geography eliminates BYNET traffic as well as often providing a smaller sized row to be read and joined. For more information, see “Using a Single-Table Join Index” on page 2-41.

A multitable join index is useful for queries where the index structure contains all the columns referenced by one or more joins, thereby allowing the index to cover that part of the query, making it possible to retrieve the requested data from the index rather than accessing its underlying base tables. For obvious reasons, an index with this property is often referred to as a *covering index*.

Depending on how the index is defined, single-table join indexes can also be useful for queries where the index structure contains only some of the columns referenced in the statement. This situation is referred to as a *partial cover* of the query.

Join indexes are also useful for queries that aggregate columns from tables with large cardinalities. These indexes play the role of prejoin and summary tables without denormalizing the logical design of the database and without incurring the update anomalies and performance problems presented by denormalized tables. While it is true that denormalization often enhances the performance of a particular query or family of queries, it can and often does make other queries perform more poorly.

Unlike traditional indexes, join indexes do not implicitly store pointers to their associated base table rows. Instead, they are generally used as a fast path final access point that eliminates the need to access and join the base tables they represent. They substitute for rather than point to base table rows. The only exception to this is the case where an index partially covers a query. If the index is defined using either the ROWID keyword or the UPI of its base table as one of its columns, then it can be used to join with the base table to cover the query.

Further Information

FOR more information on ...	SEE ...
SQL data definition language statements related to join indexes	• “CREATE JOIN INDEX” and “DROP JOIN INDEX” in the chapter “SQL Data Definition Language Statement Syntax” in <i>Teradata RDBMS SQL Reference, Volume 4</i> • HELP JOIN INDEX” in the chapter “SQL Help and Database Object Definition Tools: HELP and SHOW” in <i>Teradata RDBMS SQL Reference, Volume 4</i>
database design considerations for join indexes	<i>Teradata RDBMS Database Design</i>

Improving Performance

Depending on the complexity of the joins or the cardinalities of base tables that must be aggregated, join indexes optimize the performance of certain types of workloads.

To improve the performance of creating a join index and maintaining it during updates, collect statistics on its base tables. The guidelines for collecting statistics on the relevant columns are the same as those for any regular join query that is frequently executed or whose performance is critical. The only difference with join indexes is that the produced join result is persistently stored and maintained by the AMP software without user intervention.

Using a Single-Table Join Index

When you have an application for which join queries against a table would benefit from replicating some or all of its columns in another table that is hashed on the join field (usually the primary index of the table to which it is to be joined) rather than the primary index of the original base table, then you should consider creating one or more single-table join indexes on that table.

For example, you might want to create a single-table join index to avoid redistributing a large base table or to avoid the sometimes prohibitive storage requirements of a multitable join index. A single-table join index might be useful for commonly made joins having low predicate selectivity but high join selectivity, for example.

This scheme substitutes the join index for the underlying base table and defines a primary index that ensures that rows containing only the columns to be joined are hashed to the same AMPs, eliminating any need for row redistribution when the join is made.

Even though each single-table join index you create partly or entirely replicates its base table, you cannot query or update them directly just as you cannot directly query or update any other join index.

For more information on this technique, see *Teradata RDBMS Database Design* and “CREATE JOIN INDEX” in Chapter 1: “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

Collecting Statistics on Join Indexes

You must collect statistics directly on the index columns of a multitable join index. In general, collect statistics on the base tables underlying a single-table join index rather than on the join index columns directly.

To provide the Optimizer with the information it needs to generate an optimal plan, collect statistics on the primary index of the join index.

To assist the Optimizer to evaluate alternate access paths for scanning a join index, collect statistics on secondary indexes defined on a join index.

You can also collect statistics on additional join index columns that frequently appear in WHERE clause search conditions, especially if the column is the sort key for a value-ordered join index.

In general, there is no benefit in collecting statistics on a join index for joining columns specified in the join index definition itself. Collect statistics related to these columns on the underlying base tables rather than on the join index.

Effects of Join Indexes

Join index limits affect the following Teradata RDBMS functions and features.

- **Load Utilities**
MultiLoad and FastLoad utilities cannot be used to load or unload data into base tables that have a join index defined on them because join indexes are not maintained during the execution of these utilities. If an error occurs, the join index must be dropped and recreated after that table has been loaded.
The TPump utility, which performs standard SQL row inserts and updates, can be used to load or unload data into base tables with join indexes because it properly maintains join indexes during execution.
- **ARC (Archive and Recovery)**
Archive and Recovery cannot be used on a join index itself. Archiving is permitted on a base table or database that has an associated join index defined. During a restore of such a base table or database, the system does not rebuild the join index. You must drop the existing join index definition and create a new join index definition before any such index can be used again in the execution of queries.
- **Permanent Journal Recovery**
Using a permanent journal to recover a base table (that is, ROLLBACK or ROLLFORWARD) with an associated join index defined is permitted. The join index is not automatically rebuilt during the recovery process. Instead, it is marked as non-valid and it must be dropped and recreated before it can be used again in the execution of queries.

Queries Using a Join Index

In most respects, a join index is similar to a base table. For example, you can do the following things to a join index:

- Create nonunique secondary indexes on its columns.
- Execute COLLECT STATISTICS, DROP STATISTICS, HELP, and SHOW statements.

Unlike base tables, you *cannot* do the following things with join indexes:

- Query or update join index rows explicitly.
- Store and maintain arbitrary query results such as expressions.
Note that you *can* maintain aggregates if you define the join index to do so.
- Create explicit unique indexes on its columns.
- Partition its primary index.

For More Information About Join Indexes

For more detailed information on using join indexes to enhance the performance of your databases, consult the following books:

- *Teradata RDBMS Database Design*
- *Teradata RDBMS Performance Optimization*
- *Teradata RDBMS SQL Reference, Volume 4*

Hash Indexes

Introduction

Hash indexes are used for the same purposes as single-table join indexes. The principal differences between hash indexes and single-table join indexes are listed in the following table.

Hash Index	Single-Table 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.

Hash indexes create a full or partial replication of a base table with a primary index on a foreign key column table to facilitate joins of very large tables by hashing them to the same AMP.

You can define a hash index on one table only. The functionality of hash indexes is a superset to that of single-table join indexes.

Hash indexes are not indexes in the usual sense of the word. They are base tables that cannot be accessed directly.

The Optimizer includes a hash index in a query plan in the following situations.

- The index covers all or part of a join query, thus eliminating the need to redistribute rows to make the join.
In the case of partial query covers, the Optimizer uses certain implicitly defined elements in the hash index to join it with its underlying base table to pick up the columns necessary to complete the cover.
- A query requests that one or more columns be aggregated, thus eliminating the need to perform the aggregate computation

For information on SQL data definition language statements related to hash indexes, see the following topics in *Teradata RDBMS SQL Reference, Volume 4*:

- “CREATE HASH INDEX” and “DROP HASH INDEX” in Chapter 1 – “SQL Data Definition Language Statement Syntax”
- “HELP HASH INDEX” in Chapter 3 – “SQL Help and Database Object Definition Tools: HELP and SHOW”

For information on database design considerations for join indexes, see *Teradata RDBMS Database Design*.

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 also implicitly 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 the system 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
 - 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, but for a hash index, Teradata makes this decision.
- Both can be FALLBACK protected.
- Neither can be queried or directly updated.
- Neither can store an arbitrary query result.
- Both share the same restrictions for use with the MultiLoad, FastLoad, and Archive/Recovery utilities.
- A hash index implicitly defines a direct access path to base table rows. A join index may be explicitly specified to define a direct access path to base table rows.

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 2-46](#) for further information.

When you create a hash index with a BY clause, the columns specified in that clause become the primary index for that index. 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.

Alternatively, you could collect the same statistics using the column clause syntax, but you would have to perform two separate statements.

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

The same restrictions hold for the DROP STATISTICS statement.

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.

There are two reasons why you might need to collect statistics on hash index 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 hash index columns.
- If the 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 *must* collect the statistics for that column set on the hash index itself.

Effects of Hash Indexes

Join index limits affect the following Teradata RDBMS functions and features.

- **Load Utilities**
MultiLoad and FastLoad utilities cannot be used to load or unload data into base tables that have an associated hash index defined on them because hash indexes are not maintained during the execution of these utilities. The hash index must be dropped and recreated after that table has been loaded. The TPump utility, which performs standard SQL row inserts and updates, can be used because hash indexes are properly maintained during its execution.
- **ARC (Archive and Recovery)**
Archive and Recovery cannot be used on a hash index itself. Archiving is permitted on a base table or database that has an associated hash index defined. During a restore of such a base table or database, the system does not rebuild the hash index. You must drop the existing hash index definition and create a new one before any such index can be used again in the execution of queries.
- **Permanent Journal Recovery**
Using a permanent journal to recover a base table using ROLLBACK or ROLLFORWARD with an associated hash index defined is permitted. The hash index is not automatically rebuilt during the recovery process. Instead, the hash index is marked as non-valid and it must be dropped and recreated before it can be used again in the execution of queries.

Queries Using a Hash Index

In most respects, a hash index is similar to a base table. For example, you can perform COLLECT STATISTICS, DROP STATISTICS, HELP, and SHOW statements on a hash index:

Unlike base tables, you *cannot* do the following things with hash indexes:

- Query or update hash index rows explicitly.
- Store and maintain arbitrary query results such as expressions.
- Create explicit unique indexes on its columns.
- Partition the primary index of the hash index.

For More Information About Hash Indexes

Consult the following manuals for more detailed information on using hash indexes to enhance the performance of your databases.

- *Teradata RDBMS Database Design*
- *Teradata RDBMS Performance Optimization*
- *Teradata RDBMS SQL Reference, Volume 4*

Table Access Summary

The following table characterizes the access methods available to the Optimizer for accessing tables. Definitions for the access method abbreviations are provided in the following table:

Access Method Abbreviation	Definition
UPI	Unique Primary Index
UPPI	Unique Partitioned Primary Index
NUPI	Nonunique Primary Index
NUPPI	Nonunique Partitioned Primary Index
USI	Unique Secondary Index
NUSI	Nonunique Secondary Index
JI	Join Index
HI	Hash Index
FTS	Full Table Scan
PS	Partition Scan

For many table join operations, a join index is often a more efficient means for accessing data. See [“Join Indexes” on page 2-39](#).

Access Method	Relative Efficiency	Number of AMPs Used	Number of Rows Accessed	Spool Space Required?	
UPI	Very efficient.	1	1	No	
UPPI	Very efficient	1	1	No	
NUPI	Efficient when selectivity is high and skew is low.	1 NUPPI access may need to access multiple partitions on the AMP depending on the number of non-eliminated partitions.	Multiple	Yes	
NUPPI				Depends.	
				IF a query returns ...	THEN ...
				a single response	no
				more rows than can be handled by a single response	yes
USI	Very efficient.	2	1	No	

Access Method	Relative Efficiency	Number of AMPs Used		Number of Rows Accessed		Spool Space Required?	
NUSI	Efficient when the number of rows accessed is less than the number of data blocks in the table.	All		Multiple		Yes	
JI	Very efficient.	If the JI is ...	THEN the number of AMPs used is ...	If the JI is ...	THEN the number of rows accessed is ...	Depends.	
		accessed by its primary index	1	accessed by its primary index	multiple	covers the query	no
		not accessed by its primary index	all	not accessed by its primary index	all	is a JI that partially covers the query and its definition specifies either ROWID or the UPI for the base tables	yes
HI	Very efficient.	Depends on how the index is distributed.		Multiple		Depends.	
						IF the HI ...	THEN ...
						covers the query	no
FTS	Efficient because each row is touched only once.	All		All		partially covers the query	yes
						Can require spool as large as the table being scanned.	
PS	Efficient because each row in only non-eliminated partitions is touched only once.	All		Multiple		Yes	

Views

Views and Tables

A view can be compared to a window through which you can see selected portions of a database. Views are used to retrieve portions of one or more tables or other views.

Views look like tables to a user, but they are virtual, not physical, tables. They display data in columns and rows and, in general, can be used as if they were physical tables. However, only the column definitions for a view are stored: views are *not* physical tables.

A view does not contain data: it is a virtual table whose definition is stored in the data dictionary. The view is not materialized until it is referenced by a statement. Some operations that are permitted for the manipulation of tables are not valid for views, and other operations are restricted, depending on the view definition.

Defining a View

The CREATE VIEW statement defines a view. The statement names the view and its columns, defines a SELECT on one or more columns from one or more underlying tables and/or views, and can include conditional expressions and aggregate operators to limit the row retrieval.

Why Use Views?

The primary reason to use views is to simplify end user access to the Teradata database. Views provide a constant vantage point from which to examine and manipulate the database. Their perspective is altered neither by adding or nor by dropping columns from its component base tables unless those columns are part of the view definition.

From an administrative perspective, views are useful for providing an easily maintained level of security and authorization. For example, users in a Human Resources department can access tables containing sensitive payroll information without being able to see salary and bonus columns. Views also provide administrators with an ability to control read and update privileges on the database with little effort.

Restrictions on Views

Some operations that are permitted on base tables are not permitted on views—sometimes for obvious reasons and sometimes not.

The following set of rules outlines the restrictions on how views can be created and used.

- You cannot create an index on a view.
- A view definition cannot contain an ORDER BY clause.
- Any derived columns in a view must define a NAMED clause for those columns.
- You cannot update tables from a view under the following circumstances.
 - The view is defined as a join view (defined on more than one table)
 - The view contains derived columns.
 - The view definition contains a DISTINCT clause.
 - The view definition contains a GROUP BY clause.
 - The view defines the same column more than once.

Triggers

Definition

Triggers are active database objects associated with a subject table.

A trigger is defined by a CREATE TRIGGER statement (see Chapter 1 – “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*). It is essentially a stored SQL statement or sequence of SQL statements associated with a table.

Triggers execute when an INSERT, UPDATE, or DELETE modifies a specified column or columns in the subject table.

Typically, the stored trigger performs an UPDATE, INSERT, or DELETE operation on a table different than the subject table.

Process Flow for a Trigger

The following table outlines a general process flow for a trigger. Note that this is a logical flow, not a physical re-enactment of how the Teradata RDBMS processes a trigger.

Stage	Description	
1	The trigger event (triggering action) occurs on the subject table.	
2	A determination is made as to whether triggers defined on the subject table are to become active upon an insert, a delete, or an update operation against that table.	
3	Qualified triggers are examined to determine whether they are to fire before or after the trigger event.	
	IF the trigger fires ...	THEN control flows ...
	before	to the triggered action. Upon completion of the triggered action, control returns to the trigger event, which then acts on the subject table.
	after	to the trigger event, which acts on the subject table. Upon completion of the trigger event, control passes to the triggered action.

Stage	Description						
4	When multiple triggers qualify, and all have the same trigger action time, trigger event, and FOR EACH clause, then their firing order is determined by the integer variable defined for ORDER in the CREATE TRIGGER statement. Note that the ORDER variable does not override the BEFORE or AFTER specifications from stage 3 on page 2-52, it just provides a mechanism for determining the execution order among triggers that otherwise have identical execution orders.						
5	The triggered SQL statement (triggered action) performs. If the trigger definition specifies that old, new, or both old and new data for the triggered action is to be collected under an alias, then that information is stored under the defined aliases as follows. <table> <tr> <th>IF defined for this trigger type ...</th><th>THEN qualified data is stored under this alias, depending on how the REFERENCING clause is defined ...</th></tr> <tr> <td>Statement</td><td>the OLD_TABLE <i>identifier</i> variable, the NEW_TABLE <i>identifier</i> variable, or both.</td></tr> <tr> <td>Row</td><td>the OLD <i>identifier</i> variable, the NEW <i>identifier</i>, or both.</td></tr> </table>	IF defined for this trigger type ...	THEN qualified data is stored under this alias, depending on how the REFERENCING clause is defined ...	Statement	the OLD_TABLE <i>identifier</i> variable, the NEW_TABLE <i>identifier</i> variable, or both.	Row	the OLD <i>identifier</i> variable, the NEW <i>identifier</i> , or both.
IF defined for this trigger type ...	THEN qualified data is stored under this alias, depending on how the REFERENCING clause is defined ...						
Statement	the OLD_TABLE <i>identifier</i> variable, the NEW_TABLE <i>identifier</i> variable, or both.						
Row	the OLD <i>identifier</i> variable, the NEW <i>identifier</i> , or both.						
6	If defined, the trigger passes control to the next trigger in a cascaded sequence. Otherwise, control is passed to the next statement in the application.						

Altering Table Structure and Definition

Introduction

You may need to change the structure or definition of an existing table or temporary table. In many cases, you can use ALTER TABLE and RENAME to make the changes. Some changes, however, may require you to use CREATE TABLE to recreate the table.

How to Make Changes

TO perform this function ...	USE this statement ...
add or drop columns on an existing table or temporary table	ALTER TABLE
add column default control, FORMAT, and TITLE attributes on an existing table or temporary table	
add or remove journaling options on an existing table or temporary table	
add or remove the FALLBACK option on an existing table or temporary table	
change the DATABLOCKSIZE or percent FREESPACE on an existing table or temporary table	
add or drop column and table level constraints on an existing table or temporary table	
change the LOG and ON COMMIT options for a global temporary table	
modify referential constraints	
change the properties of the primary index for a table	
change the partitioning properties of the primary index for a table, including modifications to the partitioning expression defined for use by a partitioned primary index	
regenerate table headers and optionally validate and correct the partitioning of PPI table rows	

TO perform this function ...	USE this statement ...
change column attributes (that do not affect stored data) on an existing table or temporary table	ALTER TABLE For rules on changing column attributes, including the data type attribute, see “ALTER TABLE” in Chapter 1: “SQL Data Definition Language Statement Syntax” of <i>Teradata RDBMS SQL Reference, Volume 4</i>.
change the name of a table or temporary table	RENAME
redefine the primary index for a table	CREATE TABLE to recreate the table. Interactively, the SHOW TABLE statement can call up the current table definition, which can then be modified and resubmitted to create a new table. If the stored data is not affected by incompatible data type changes, an INSERT... SELECT statement can be used to transfer data from the existing table to the new table.
change a data type attribute that affects existing data	
define or delete the COMPRESS attribute for an existing column	
add a column that would exceed the maximum column count	

Dropping Tables, Views, Triggers, Procedures, and Indexes

To drop this type of database object ...	Use this SQL statement ...
Secondary Index	DROP INDEX
Stored procedure	DROP PROCEDURE
Table	DROP TABLE
Temporary table	
Primary index	
Trigger	DROP TRIGGER
View	DROP VIEW

For further information on these statements, see “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

Referential Integrity

Introduction

Referential integrity (RI) is defined as all the following notions.

- The concept of relationships between tables, based on the definition of a primary key and a foreign key.
- A mechanism that provides for specification of columns within a *referencing* table that are foreign keys for columns in some other *referenced* table.

Referenced columns must be defined as one of the following.

- Primary key columns
- Unique columns
- A reliable mechanism for preventing accidental database corruption when performing inserts, updates, and deletes.

Referential integrity requires that a row having a non-null value for a *referencing* column cannot exist in a table if an equal value does not exist in a *referenced* column.

Varieties of Referential Integrity Enforcement Supported by Teradata

Teradata supports two forms of declarative SQL for enforcing referential integrity: a standard method that enforces RI on a row-by-row basis and a batch method that enforces RI on a statement basis. Both methods offer the same measure of integrity enforcement, but perform it in different ways.

A third form is related to these in that it provides a declarative definition for a referential relationship, but it does not enforce that relationship. Enforcement of the declared referential relationship is left to the user by any appropriate method.

Referencing (Child) Table

The referencing table is referred to as the child table, and the specified child table columns are the referencing columns.

Referencing columns must have the same numbers and types of columns, data types, and sensitivity as the referenced table keys. COMPRESS is not allowed on either referenced or referencing columns and column-level constraints are not compared.

Referenced (Parent) Table

A child table must have a parent, and the referenced table is referred to as the parent table.

The parent key columns in the parent table are the referenced columns.

Because the referenced columns are defined as unique constraints, they must be one of the following *unique* indexes.

- A unique primary index (UPI), defined as NOT NULL
- A unique secondary index (USI), defined as NOT NULL

Terms Related to Referential Integrity

The following terms are used to explain the concept of referential integrity.

Term	Definition						
Child Table	A table where the referential constraints are defined. Child table and referencing table are synonyms.						
Parent Table	The table referenced by a child table. Parent table and referenced table are synonyms.						
Primary Key	A unique identifier for a row of a table.						
Foreign Key	A column set in the child table that is also the primary key in the parent table. Foreign keys can consist of as many as sixteen different columns.						
Referential Constraint	A constraint defined on a column set or a table to ensure referential integrity. For example, consider the following table definition: <pre>CREATE TABLE A (A1 CHAR(10) REFERENCES B (B1) , A2 INTEGER FOREIGN KEY (A1,A2) REFERENCES C PRIMARY INDEX (A1)) ;</pre> This CREATE TABLE statement specifies the following referential integrity constraints. <table> <tr> <th>This constraint ...</th><th>Is defined at this level ...</th></tr> <tr> <td>1</td><td>column. Implicit foreign key A1 references the parent key B1 in table B.</td></tr> <tr> <td>2</td><td>table. Explicit composite foreign key (A1, A2) implicitly references the UPI of parent table C, which must be two columns, the first typed CHAR(10) and the second typed INTEGER. Both parent table columns must also be defined as NOT NULL.</td></tr> </table>	This constraint ...	Is defined at this level ...	1	column. Implicit foreign key A1 references the parent key B1 in table B.	2	table. Explicit composite foreign key (A1, A2) implicitly references the UPI of parent table C, which must be two columns, the first typed CHAR(10) and the second typed INTEGER. Both parent table columns must also be defined as NOT NULL.
This constraint ...	Is defined at this level ...						
1	column. Implicit foreign key A1 references the parent key B1 in table B.						
2	table. Explicit composite foreign key (A1, A2) implicitly references the UPI of parent table C, which must be two columns, the first typed CHAR(10) and the second typed INTEGER. Both parent table columns must also be defined as NOT NULL.						

Why Referential Integrity Is Important

Consider the employee and payroll tables for any business.

With referential integrity constraints, the two tables work together as one. When one table gets updated, the other table also gets updated.

The following case depicts a useful referential integrity scenario.

Looking for a better career, Mr. Clark Johnson leaves his company. Clark Johnson is deleted from the employee table.

The payroll table, however, does not get updated because the payroll clerk simply forgets to do so. Consequently, Mr. Clark Johnson keeps getting paid.

With good database design, referential integrity relationship would have been defined on these tables. They would have been linked and, depending on the defined constraints, the deletion of Clark Johnson from the employee table could not be performed unless it was accompanied by the deletion of Clark Johnson from the payroll table.

Besides data integrity and data consistency, referential integrity also has the benefits listed in the following table.

Benefit	Description
Increases development productivity	It is not necessary to code SQL statements to enforce referential constraints. The Teradata RDBMS automatically enforces referential integrity.
Requires fewer programs to be written	All update activities are programmed to ensure that referential constraints are not violated. The Teradata RDBMS enforces referential integrity in all environments. No additional programs are required.
Improves performance	The Teradata RDBMS chooses the most efficient method to enforce the referential constraints.

Rules for Assigning Columns as FOREIGN KEYS

The FOREIGN KEY columns in the *referencing* table must be identical in definition with the keys in the *referenced* table. Corresponding columns must have the same data type and case sensitivity.

- The COMPRESS option is not permitted on either the referenced or referencing column(s).
- Column level constraints are not compared.
- A one-column FOREIGN KEY cannot reference a single column in a multi-column primary or unique key—the foreign and primary/unique key must contain the same number of columns.

Circular References Are Allowed

References can be defined as circular in that TableA can reference TableB, which can reference TableA. In this case, at least one set of FOREIGN KEYS must be defined on nullable columns.

If the FOREIGN KEYS in TableA are on columns defined as nullable, then rows could be inserted into TableA with nulls for the FOREIGN KEY columns. Once the appropriate rows exist in TableB, the nulls of the FOREIGN KEY columns in TableA could then be updated to contain non-null values which match the TableB values.

References Can Be to the Table Itself

FOREIGN KEY references can also be to the same table that contains the FOREIGN KEY. The referenced columns must be different columns than the FOREIGN KEY, and both the referenced and referencing columns must subscribe to the referential integrity rules.

CREATE and ALTER TABLE Syntax

Referential integrity affects the syntax and semantics of CREATE TABLE and ALTER TABLE. See “ALTER TABLE” and “CREATE TABLE” in the chapter “SQL Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4* for more details.

Maintaining Foreign Keys

Definition of a FOREIGN KEY requires that the Teradata RDBMS maintain the integrity defined between the *referenced* and *referencing* table.

The Teradata RDBMS maintains the integrity of foreign keys as explained in the following table.

FOR this data manipulation activity ...	The system verifies that ...
A row is inserted into a <i>referencing</i> table and foreign key columns are defined to be NOT NULL.	a row exists in the <i>referenced</i> table with the same values as those in the foreign key columns. If such a row does not exist, then an error is returned. If the foreign key contains multiple columns, and if any one column value of the foreign key is null, then none of the foreign key values are validated.
The values in foreign key columns are altered to be NOT NULL.	a row exists in the <i>referenced</i> table that contains values equal to the altered values of all of the foreign key columns. If such a row does not exist, then an error is returned.

FOR this data manipulation activity ...	The system verifies that ...										
A row is deleted from a <i>referenced</i> table.	that no rows exist in <i>referencing</i> tables with foreign key values equal to those of the row to be deleted. If such rows exist, then an error is returned.										
Before a <i>referenced</i> column in a <i>referenced</i> table is updated.	that no rows exist in a referencing table with foreign key values equal to those of the referenced columns. If such rows exist, then an error is returned.										
Before the structure of columns defined as foreign keys or referenced by foreign keys is altered.	is not changed in any way that would violate the rules for definition of a foreign key constraint. An ALTER TABLE or DROP INDEX statement attempting to change such a columns structure returns an error.										
A table <i>referenced</i> by another is dropped.	the referencing table has dropped its foreign key reference to the referenced table.										
An ALTER TABLE statement adds a foreign key reference to a table. The same processes occur whether the reference is defined for standard or for soft referential integrity.	all of the values in the foreign key columns are validated against columns in the referenced table. The following process validates the column references. <table border="1"> <thead> <tr> <th>Stage</th><th>Process</th></tr> </thead> <tbody> <tr> <td>1</td><td>An error table is generated when ALTER TABLE is parsed.</td></tr> <tr> <td>2</td><td>The error table is defined with the same columns and PRIMARY KEY as the target table of the ALTER TABLE statement.</td></tr> <tr> <td>3</td><td>The name of the error table is the target table name suffixed with the reference index number.</td></tr> <tr> <td>4</td><td> A reference index number is assigned to each foreign key constraint for a table. The number can be determined by using of one of the following system views. - RI_Child_Tables - RI_Distinct_Children - RI_Distinct_Parents - RI_Parent_Tables </td></tr> </tbody> </table>	Stage	Process	1	An error table is generated when ALTER TABLE is parsed.	2	The error table is defined with the same columns and PRIMARY KEY as the target table of the ALTER TABLE statement.	3	The name of the error table is the target table name suffixed with the reference index number.	4	A reference index number is assigned to each foreign key constraint for a table. The number can be determined by using of one of the following system views. - RI_Child_Tables - RI_Distinct_Children - RI_Distinct_Parents - RI_Parent_Tables
Stage	Process										
1	An error table is generated when ALTER TABLE is parsed.										
2	The error table is defined with the same columns and PRIMARY KEY as the target table of the ALTER TABLE statement.										
3	The name of the error table is the target table name suffixed with the reference index number.										
4	A reference index number is assigned to each foreign key constraint for a table. The number can be determined by using of one of the following system views. - RI_Child_Tables - RI_Distinct_Children - RI_Distinct_Parents - RI_Parent_Tables										

FOR this data manipulation activity ...	The system verifies that ...	
(continued)		
	Stage	Process
	5	The error table is created under the same user or database as the table being altered. 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. No error is returned if rows in the referencing table contain values in the foreign key columns that 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. You can use the rows in the error table to define which corrections must be made. Maintenance of the error table is also the responsibility of the user.

Referential Integrity and the ARC Utility

The Archive (ARC) utility archives and restores individual tables. It also copies tables from one Teradata RDBMS to another.

When a table is restored or copied into a Teradata RDBMS, the dictionary definition of that table is also restored. The dictionary definitions of both the *referenced* and *referencing* table contain the complete definition of a reference.

By restoring a single table, it is possible to create an inconsistent reference definition in the Teradata RDBMS. When either a referenced or referencing table is restored, the reference is marked as inconsistent in the dictionary definitions. The ARC utility can validate these references once the restore is done.

See *Teradata Archive/Recovery for Reference* or *Teradata ASF2 Reference* for further information.

While a table is marked as inconsistent, updates, inserts, and deletes to it are not permitted. The table is not fully usable until the inconsistencies are resolved.

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

The DROP INCONSISTENT REFERENCES option of the ALTER TABLE statement removes these inconsistent references from a table.

The syntax for this application is the following.

```
ALTER TABLE [dbname.]tname DROP INCONSISTENT REFERENCES [;]
```

The user executing the statement must have DROP rights on the target table of the statement.

The result of this statement is to remove those internal indexes used to establish references which contain inconsistencies. The base data of the table is not modified.

Referential Integrity and the FastLoad and MultiLoad Utilities

Foreign key references are not supported for any table that is the target table for a FastLoad or MultiLoad.

For further details, see:

- *Teradata RDBMS Database Design*
- *Teradata FastLoad Reference*
- *Teradata MultiLoad Reference*

Profiles

Definition

Profiles define values for the following system parameters:

- Default database
- Spool space
- Temporary space
- Account
- Password security attributes

An administrator can define a profile and assign it to a group of users who share the same settings.

Advantages of Using Profiles

Use profiles to:

- Simplify system administration.

Administrators can create a profile that contains system parameters such as account, default database, spool space, temporary space, and password attributes, and assign the profile to a group of users. To change a parameter, the administrator updates the profile instead of each individual user.

- Control user-level password security.

A profile can define password attributes such as the number of:

- Days before a password expires
- Days before a password can be used again
- Minutes to lock out a user after a certain number of failed logon attempts

Administrators can assign the profile to an individual user or to a group of users.

Usage

The following steps describe how to use profiles to manage a common set of parameters for a group of users.

Step	Action
1	Define a user profile. A CREATE PROFILE statement defines a profile, and lets you set: • Account identifier to charge for the space used • Default database • Space to allocate for spool files • Space to allocate for temporary tables • Number of days before the password expires • 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
2	Assign the profile to users. Use the CREATE USER or MODIFY USER statement to assign a profile to a user. Profile settings override the values set for the user.
3	If necessary, change any of the system parameters for a profile. Use the MODIFY PROFILE statement to change a profile.

For More Information

See “SQL Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4* for information on the syntax and usage of profiles.

Roles

Definition

Roles define access privileges on database objects. A user who is assigned a role can access all the objects that the role has privileges to.

Roles simplify management of user access rights. 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.

Advantages of Using Roles

- Use roles to:
- Simplify access rights administration.
A database administrator can grant rights on database objects to a role and have the rights automatically applied to all users assigned to that role.
When a user’s function within an organization changes, changing the user’s role is far easier than deleting old rights and granting new rights to go along with the new function.
 - Reduce dictionary disk space.
Maintaining rights on a role level rather than on an individual level makes the size of the DBC.AccessRights table much smaller. Instead of inserting one row per user per right on a database object, the Teradata RDBMS inserts one row per role per right in DBC.AccessRights, and one row per role member in DBC.RoleGrants.

Usage

The following steps describe how to manage user access privileges using roles.

Step	Action
1	Define a role. A CREATE ROLE statement defines a role. A newly created role does not have any associated privileges.
2	Add access privileges to the role. Use the GRANT statement to grant privileges to roles on databases, tables, views, macros, columns, triggers, stored procedures, join indexes, and hash indexes.

Step	Action
3	Add members to the role. Members of a role can be users or other roles. Use the GRANT statement to add members to a role.
4	Assign a default role to users. Use the CREATE USER or MODIFY USER statement to specify the default role for a user. Only one role can be <i>current</i> in a session. At logon time, the current role is the user's default role. More than role can be <i>active</i>, however. The active roles for a user include the current role and all of its members that are roles. Rights validation uses the active roles for a user.
5	If necessary, change the current role for a session. Use the SET ROLE statement to change the current role a session.

Managing role-based access rights requires sufficient privileges. For example, the CREATE ROLE statement is only authorized to users who have the CREATE ROLE system privilege.

For More Information

See “SQL Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4* for information on the syntax and usage of roles.

SQL Data Control Language (DCL)

Definition

The SQL Data Control Language, abbreviated DCL, is a subset of the SQL language that consists of all SQL statements that support the definition of security authorization for accessing database objects.

Purpose of Data Control Statements

Data control statements perform the following functions.

- Grant and revoke privileges
- Give ownership of a database to another user

Restrictions on Entering Data Control Statements

A data control statement cannot be entered as part of a multistatement request.

A data control statement can be entered as:

- A single statement request
- The solitary statement, or as the last statement, in an “explicit transaction” (one or more requests enclosed by user-supplied BEGIN TRANSACTION and END TRANSACTION statement in Teradata mode, or in ANSI mode, one or more requests ending with the COMMIT keyword).
- The solitary statement in a macro

Successful execution of a data control statement automatically creates and updates entries in the Data Dictionary.

Teradata SQL Data Control Statements

Data control statements include the following.

- GIVE
- GRANT
- GRANT LOGON
- REVOKE
- REVOKE LOGON

Further Information

For detailed information about the function, syntax, and usage of Teradata SQL Data Control statements, see Chapter 2 – “SQL Data Control Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

SQL Data Manipulation Language (DML)

Definition

The SQL Data Manipulation Language, abbreviated DML, is a subset of the SQL language that consists of all SQL statements that support the manipulation or processing of database objects.

Selecting Columns

The SELECT statement returns information from the tables in a relational database. SELECT specifies the table columns from which to obtain the data, the corresponding database (if not defined by default), and the table (or tables) to be accessed within that database.

For example, to request the data from the name, salary, and jobtitle columns of the *Employee* table, type:

```
SELECT name, salary, job_title FROM employee ;
```

The response might be something like the following results table:

Name	Salary	JobTitle
Newman P	28600.00	Test Tech
Chin M	38000.00	Controller
Aquilar J	45000.00	Manager
Russell S	65000.00	President
Clements D	38000.00	Salesperson

Note: The left-to-right order of the columns in a result table is determined by the order in which the column names are entered in the SELECT statement. Columns in a relational table are not ordered logically.

As long as a statement is otherwise constructed properly, the *spacing* between statement elements is not important as long as at least one pad character separates each element that is not otherwise separated from the next.

For example, the SELECT statement in the above example could just as well be formulated like this:

```
SELECT name,salary,job_title
FROM employee;
```

Notice that there are multiple pad characters between most of the elements and that a comma only (with no pad characters) separates column name salary from column name jobtitle.

To select all the data in the employee table, you could enter the following SELECT statement:

```
SELECT * FROM employee ;
```

The asterisk specifies that the data in *all* columns of the table is to be returned.

Selecting Rows

The SELECT statement retrieves stored data from a table. All rows, specified rows, or specific fields of all or specified rows can be retrieved. The FROM, WHERE, ORDER BY, DISTINCT, WITH, GROUP BY, and HAVING clauses provide for a fine detail of selection criteria.

To obtain data from specific rows of a table, use the WHERE clause of the SELECT statement. That portion of the clause following the keyword WHERE causes a search for rows that satisfy the condition specified.

For example, to get the name, salary, and title of each employee in Department 100, use the WHERE clause:

```
SELECT name, salary, jobtitle FROM employee  
WHERE deptno = 100 ;
```

The response is:

Name	Salary	JobTitle
Chin M	38000.00	Controller
Greene W	32500.00	Payroll Clerk
Moffit H	35000.00	Recruiter
Peterson J	25000.00	Payroll Clerk

To obtain data from a multirow result table in embedded SQL, declare a cursor for the SELECT statement and use it to fetch individual result rows for processing.

Adding Rows

The INSERT statement is used to add rows to a table.

One statement is required for each new row, except in the case of an INSERT...SELECT statement. For more detail on this, see “INSERT” in the chapter “SQL Data Manipulation Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 6*.

Values for all the fields (columns) or one or more selected fields of the row can be specified in any order. When no value is specified for a field on insert, then the system either stores its column default value or a null. One statement is required for each new row except in the case of INSERT ... SELECT.

Defaults and constraints defined by the CREATE TABLE statement affect an insert operation in the following ways:

WHEN an INSERT statement ...	THEN the system ...
attempts to add a duplicate row • for any unique index • to a table defined as SET (not to allow duplicate rows)	returns an error, with one exception. The system silently ignores duplicate rows that an INSERT ... SELECT would create when the: • table is defined as SET • mode is Teradata
omits a value for a field for which a default value is defined	stores the default value for that field.
omits a value for a field for which both of the following statements are true: • NOT NULL is specified • no default is specified	rejects the operation and returns an error message.
supplies a value that does not satisfy the constraints specified for a field or violates some defined constraint on a field or fields	

Updating Rows

The UPDATE statement allows you to modify data in one or more rows of a table. In the UPDATE statement, the column name of the data to be modified is specified along with the new value. A WHERE clause is used to qualify the row(s) whose data is changed.

Attributes specified in the CREATE TABLE statement affect an update operation in the following ways:

- When an update supplies a value that does not satisfy the constraints specified for a field, or violates some defined constraint on a field or fields, the update operation is rejected and an error message is returned.
- When an update supplies the value NULL and a NULL is allowed, any existing data is removed from the field.
- If the result of an UPDATE will violate uniqueness constraints or create a duplicate row in a table which does not allow duplicate rows, an error message is returned.

To update rows in a multirow result table in embedded SQL, declare a cursor for the SELECT statement and use it to fetch individual result rows for processing, then use a WHERE CURRENT OF clause in a positioned UPDATE statement to update the selected rows.

Deleting Rows

The DELETE statement allows you to remove an entire row or rows from a table. A WHERE clause qualifies the rows that are to be deleted.

To delete rows in a multirow result table in embedded SQL, follow the process described in the following table.

Step	Action
1	Declare a cursor for the SELECT statement.
2	Fetch individual result rows for processing using the cursor you declared.
3	Use a WHERE CURRENT OF clause in a positioned DELETE statement to delete the selected rows.

Query Processing

Introduction

An SQL query (the definition for “query” here includes DELETE, INSERT, and UPDATE as well as SELECT) can affect one AMP, several AMPs, or all AMPs in the configuration.

IF a query ...	THEN ...
involving a single table uses a unique primary index	the row hash can be used to identify a single AMP. At most one row can be returned.
involving a single table uses a nonunique primary index	the row hash can be used to identify a single AMP. Any number of rows can be returned.
uses a unique secondary index	one or two AMPs are affected (one AMP if the subtable and base table are on the same AMP). At most one row can be returned.
uses a nonunique secondary index	all AMPs take part in the operation. Any number of rows can be returned.

The SELECT statements in subsequent examples reference the following table data.

Abbreviation	Meaning
PK	Primary Key
FK	Foreign Key
UPI	Unique Primary Index

Employee

Employee Number	Manager Employee Number	Dept. Number	Job Code	Last Name	First Name	Hire Date	Birth Date	Salary Amount
PK/UIP	FK	FK	FK					
1006	1019	301	312101	Stein	John	76105	531015	2945000
1008	1019	301	312102	Kanieski	Carol	770201	580517	2925000
1005	0801	403	431100	Ryan	Loretta	761015	550910	3120000
1004	1003	401	412101	Johnson	Darlene	761015	460423	3630000
1007	1005	403	432101	Villegas	Arnando	770102	370131	4970000
1003	0801	401	411100	Trader	James	760731	470619	3755000
1016	0801	302	321100	Rogers	Nora	780310	590904	5650000
1012	1005	403	432101	Hopkins	Paulene	770315	420218	3790000
1019	0801	301	311100	Kubic	Ron	780801	421211	5770000
1023	1017	501	512101	Rabbit	Peter	790301	621029	2650000
1083	0801	619	414221	Kimble	George	910312	410330	3620000
1017	0801	501	511100	Runyon	Irene	780501	511110	6600000
1001	1003	401	412101	Hoover	William	760818	500114	2552500

Single AMP Request

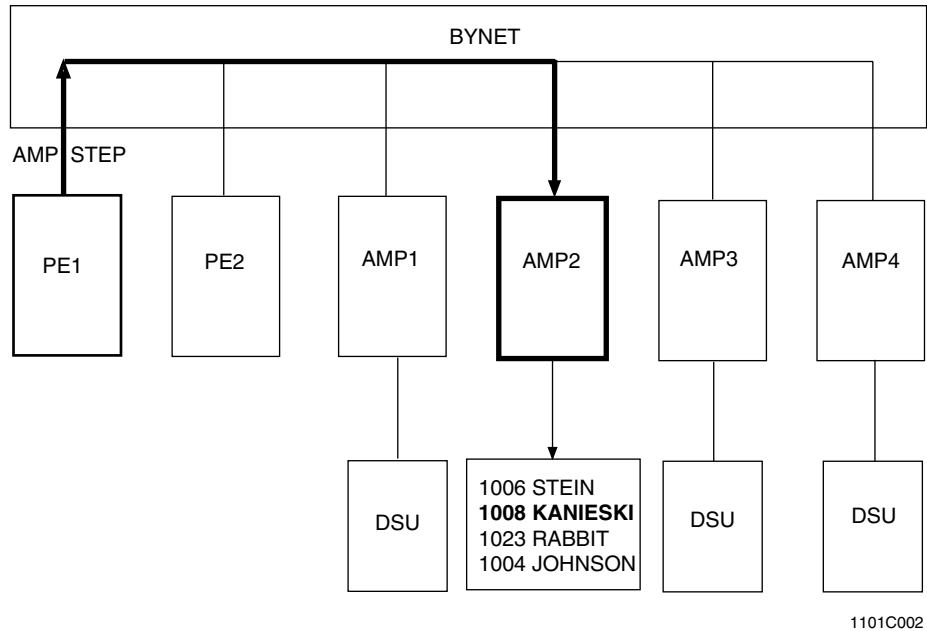
Assume that a PE receives the following SELECT statement:

```
SELECT last_name
FROM Employee
WHERE employee_number = 1008;
```

Because a unique primary index value is used as the search condition (the column employee_number is the primary index for the Employee table), PE1 generates a single AMP step requesting the row for employee 1008. The AMP step, along with the PE identification, is put into a message, and sent via the BYNET to the relevant AMP (processor).

This process is illustrated by the graphic under [“Flow Diagram of a Single AMP Request” on page 2-75](#). Only one BYNET is shown to simplify the illustration.

Flow Diagram of a Single AMP Request



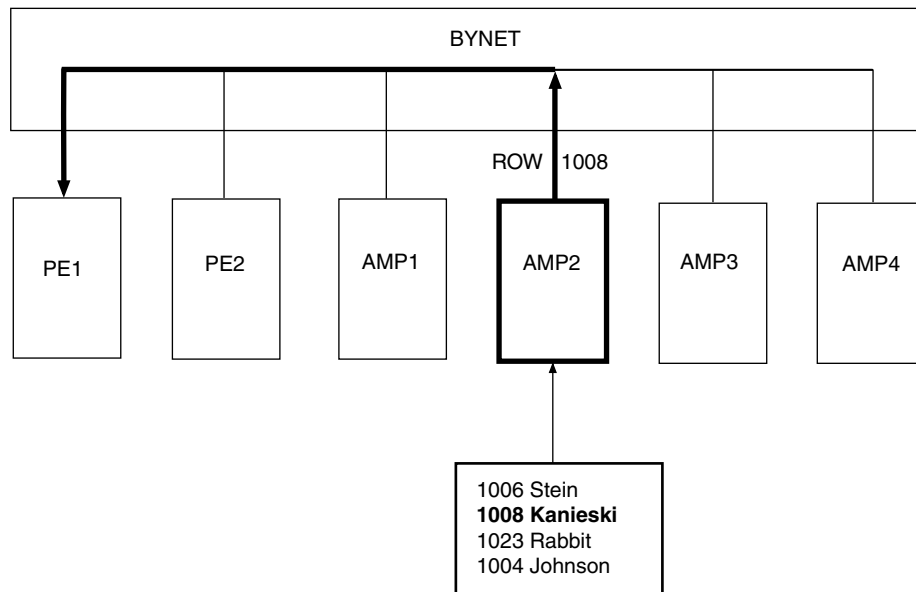
Assuming that AMP2 has the row, it accepts the message.

As illustrated by the graphic under [“Single AMP Response to Requesting PE” on page 2-76](#), AMP2 retrieves the row from its DSU (disk storage unit), includes the row and the PE identification in a return message, and sends the message back to PE1 via the BYNET.

PE1 accepts the message and returns the response row to the requesting application.

For an illustration of a single AMP request with partition elimination, see [“Single AMP Request With Partition Elimination” on page 2-80](#).

Single AMP Response to Requesting PE



1101C003

All AMP Request

Assume PE1 receives a SELECT statement that specifies a range of primary index values as a search condition as shown in the following example:

```
SELECT last_name, employee_number
FROM employee
WHERE employee_number BETWEEN 1001 AND 1010
ORDER BY last_name;
```

In this case, each value hashes differently, and all AMPs must search for the qualifying rows.

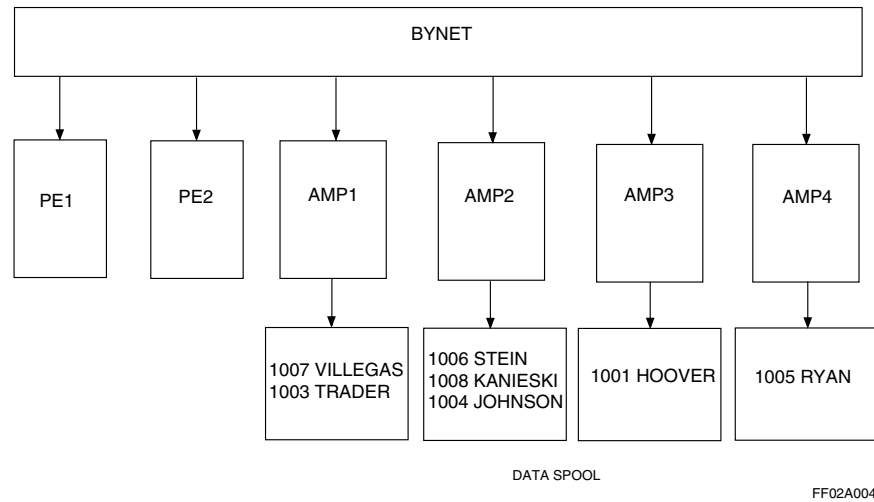
PE1 first parses the request and creates the following AMP steps:

- Retrieve rows between 1001 and 1010
- Sort ascending on last_name
- Merge the sorted rows to form the answer set

PE1 then builds a message for each AMP step and puts that message onto the BYNET. Typically, each AMP step is completed before the next one begins; note, however, that some queries can generate parallel steps.

When PE1 puts the message for the first AMP step on the BYNET, that message is broadcast to all processors as illustrated by the graphic under [“Flow Diagram for an All AMP Request”](#) on page 2-77.

Flow Diagram for an All AMP Request

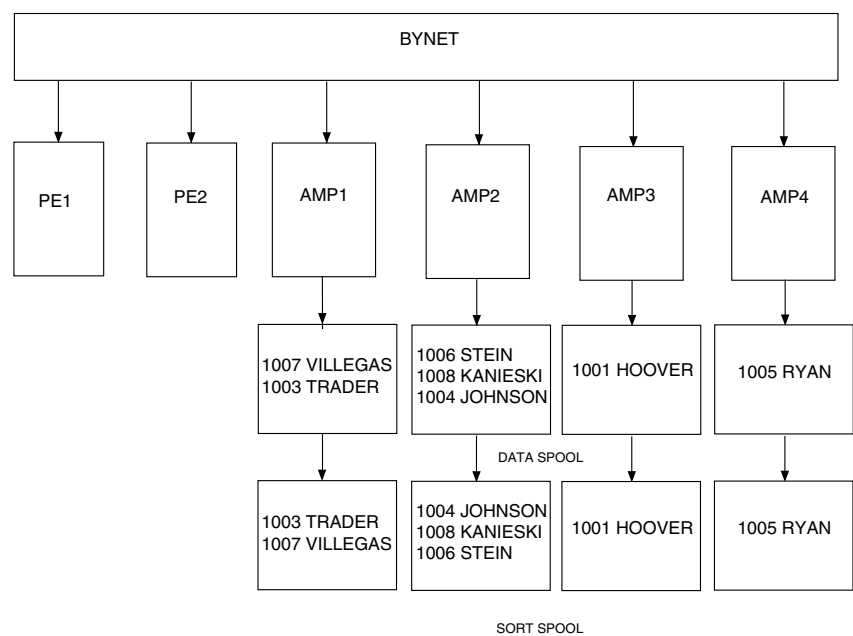


The process is as follows:

Stage	Process
1	All AMPs accept the message, but the PEs do not.
2	Each AMP checks for qualifying rows on its disk storage units.
3	If any qualifying rows are found, the data in the requested columns is converted to the client format and copied to a spool file.
4	Each AMP completes the step, whether rows were found or not, and puts a completion message on the BYNET. The completion messages flow across the BYNET to PE1.
5	When all AMPs have returned a completion message, PE1 transmits a message containing AMP Step 2 to the BYNET. Upon receipt of Step 2, the AMPs sort their individual answer sets into ascending sequence by <i>last_name</i> (see the illustration under “Flow Diagram for an AMP Sort” on page 2-78).

If partitioned on `employee_number`, the scan may be limited to a few partitions based on partition elimination.

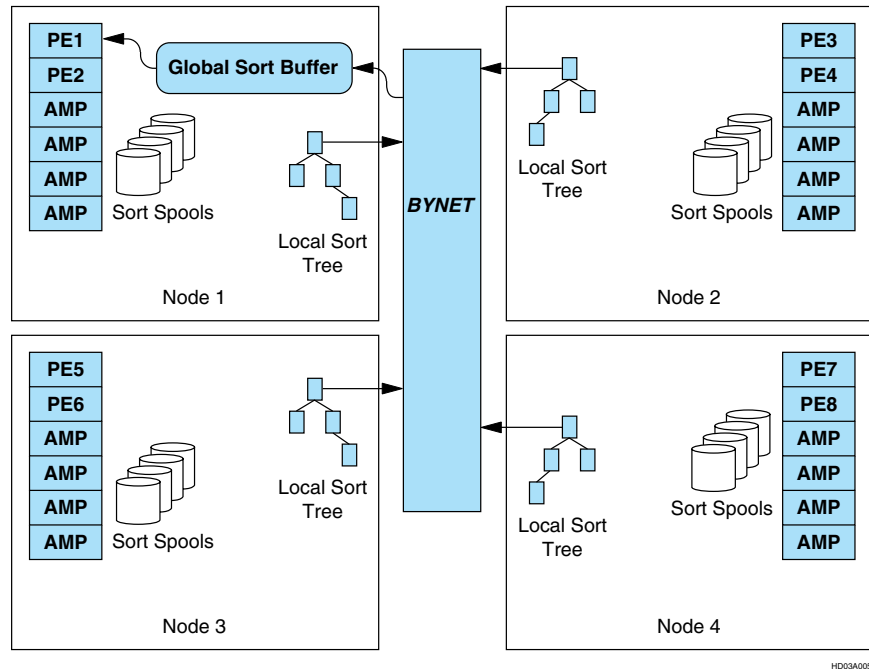
Flow Diagram for an AMP Sort



FF02A005

Stage	Process
6	Each AMP sorts its answer set, then puts a completion message on the BYNET.
7	When PE1 has received all completion messages for Step 2, it sends a message containing AMP Step 3.
8	Upon receipt of Step 3, each AMP copies the first block from its sorted spool to the BYNET. Because there can be multiple AMPs on a single node, each node might be required to handle sort spools from multiple AMPs (see the illustration under “Flow Diagram for a BYNET Merge” on page 2-79).

Flow Diagram for a BYNET Merge



Stage	Process
9	Nodes that contain multiple AMPs must first perform an intermediate sort of the spools generated by each of the local AMPs. When the local sort is complete on each node, the lowest sorting row from each node is sent over the BYNET to PE1. From this point on, PE1 acts as the Merge coordinator among all the participating nodes.
10	The Merge continues with PE1 building a globally sorted buffer. When this buffer fills, PE1 forwards it to the application and begins building subsequent buffers.
11	When a participant node has exhausted its sort spool, it sends a Done message to PE1. This causes PE1 to prune this node from the set of Merge participants. When there are no remaining Merge participants, PE1 sends the final buffer to the application along with an End Of File message.

Partition Elimination

A PPI can increase query efficiency via *partition elimination*. The degree of partition elimination depends both on the partition expression in the CREATE TABLE statement and the conditions in the query.

It is not always required that all values of the partitioning columns be specified in a query to have partition elimination occur.

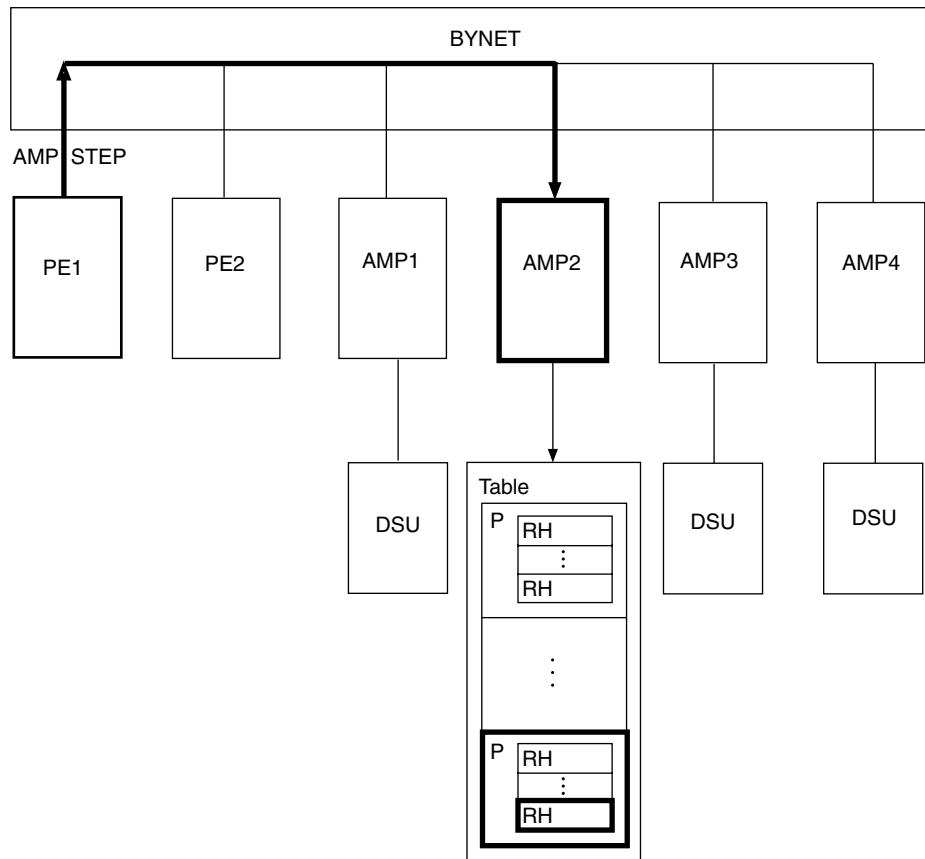
IF a SELECT ...	THEN ...						
specifies values for all the primary index columns	the AMP where the rows reside can be determined and only a single AMP is accessed.						
	<table><tr><th>IF conditions are ...</th><th>THEN ...</th></tr><tr><td>not specified on the partitioning columns</td><td>each partition can be probed to find the rows based on the hash value.</td></tr><tr><td>also specified on the partitioning columns</td><td>partition elimination may reduce the number of partitions to be probed on that AMP.</td></tr></table>	IF conditions are ...	THEN ...	not specified on the partitioning columns	each partition can be probed to find the rows based on the hash value.	also specified on the partitioning columns	partition elimination may reduce the number of partitions to be probed on that AMP.
	IF conditions are ...	THEN ...					
	not specified on the partitioning columns	each partition can be probed to find the rows based on the hash value.					
also specified on the partitioning columns	partition elimination may reduce the number of partitions to be probed on that AMP.						
For an illustration, see “Single AMP Request With Partition Elimination” on page 2-80.							
does not specify the values for all the primary index columns	an all-AMP full file scan is required for a table with an NPPI. However, with a PPI, if conditions are specified on the partitioning columns, partition elimination may reduce an all-AMP full file scan to an all-AMP scan of only the non-eliminated partitions.						

Single AMP Request With Partition Elimination

If a SELECT specifies values for all the primary index columns, the AMP where the rows reside can be determined and only a single AMP is accessed.

If conditions are also specified on the partitioning columns, partition elimination may reduce the number of partitions to be probed on that AMP.

The following diagram illustrates this process:



1101A094

The AMP Step includes the list of partitions (P) to access. Partition elimination reduces access to the partitions that satisfy the query requirements. In each partition, look for rows with a given row hash value (RH) of the PI.

Full Table Scan

A full table scan should be avoided when possible because it requires that each row of the table be accessed. For full table scans, the efficiency is dependent on the number of AMPs, and thus the extent of parallelism. Full table scans generate spool files that can have as many rows as the base table.

Whether the system can use *row hashing*, or do a table scan with partition elimination, or whether it must do a *full table scan* depends on the predicates or conditions that appear in the WHERE clause associated with an UPDATE, DELETE, or SELECT statement.

The following functions are applied to rows identified by the WHERE clause, and have no effect on the selection of rows from the base table:

- GROUP BY
- HAVING
- INTERSECT
- MINUS/EXCEPT
- ORDER BY
- QUALIFY
- SAMPLE
- UNION
- WITH ... BY
- WITH

In summary, a query influences processing choices as follows.

- A full table scan (possibly with partition elimination if the table has a PPI) is required if the query includes an implicit range of values, such as in the following WHERE examples.

Note that when a small BETWEEN range is specified, the optimizer can use row hashing rather than a full table scan.

```
... WHERE column_name [BETWEEN <, >, <>, <=, >=]
... WHERE column_name [NOT] IN (SELECT...)
... WHERE column_name NOT IN (val1, val2 [,val3])
```

- Row hashing can be used if the query includes an explicit value, as shown in the following WHERE examples:

```
... WHERE column_name = val
... WHERE column_name IN (val1, val2, [,val3])
```

Access Types

The following table indicates the conditions that generate full table scans (FTS). If the table has a PPI, partition elimination might reduce the FTS access to only the affected partitions.

Condition Generating Full Table Scans	Access Type
nonequality comparisons	FTS
<i>column_name</i> = value or constant expression	*
<i>column_name</i> IS NULL	*
<i>column_name</i> IS NOT NULL	FTS
<i>column_name</i> IN (explicit list of values)	*
<i>column_name</i> NOT IN (explicit list of values)	FTS
<i>column_name</i> IN (subquery)	*
<i>column_name</i> NOT IN (subquery)	FTS
<i>column_name</i> BETWEEN ... AND ...	FTS
condition_1 AND condition_2	*
condition_1 OR condition_2	FTS

Condition Generating Full Table Scans	Access Type
NOT condition_1	FTS
table_1.column_x = table_1.column_y	FTS
table_1.column_x [computation] = value	FTS
table_1.column_x [computation] - table_1.column_y	FTS
column_1 column_2 = value	FTS
column_name LIKE	FTS
INDEX (column_name)	FTS
SUBSTR (column_name)	FTS
SUM	FTS
MIN	FTS
MAX	FTS
AVG	FTS
DISTINCT	FTS
COUNT	FTS
ANY	FTS
ALL	FTS
different data types	*
table_1.column_x = table_2.column_x	*
missing WHERE clause	FTS
Notes	
*	Access type depends on whether the column or columns are indexed, the type of index, and its selectivity.
FTS	Access type is a full table scan.

Subqueries

Introduction

Subqueries are nested SELECT statements. They can be used to ask a series of questions in order to arrive at a single answer.

The maximum nesting level for subqueries is 64 if the DBSControl field MaxParTreeSegs is set to 128 or higher. You can set the value for MaxParTreeSegs with the DBS Control Utility.

See “Subqueries in Search Conditions” and “Correlated Subqueries” in Chapter 1 – “The SELECT Statement” of *Teradata RDBMS SQL, Volume 6*.

Three Level Subqueries: Example

The following subqueries, nested to three levels, are used to answer the question “Who manages the manager of Marston?”

```
SELECT Name
FROM Employee
WHERE EmpNo IN
  (SELECT MgrNo
   FROM Department
  WHERE DeptNo IN
    (SELECT DeptNo
     FROM Employee
     WHERE Name = 'Marston A')
  ) ;
```

The subqueries that pose the questions that lead to the final answer are inverted:

- The third subquery asks the Employee table for the number of Marston’s department.
- The second subquery asks the Department table for the employee number (MgrNo) of the manager associated with this department number.
- The first subquery asks the Employee table for the name of the employee associated with this employee number (MgrNo).

The result table looks like the following.

Name

Watson L

This result can be obtained using only two levels of subquery, as the following example shows.

```
SELECT Name
FROM Employee
WHERE EmpNo IN
  (SELECT MgrNo
   FROM Department, Employee
   WHERE Employee.Name = 'Marston A'
   AND Department.DeptNo = Employee.DeptNo) ;
```

In this example, the second subquery defines a join of Employee and Department tables.

Teradata SQL Macros

Introduction

A frequently used SQL statement or series of statements can be incorporated into a *macro* and defined using the SQL CREATE MACRO statement. See “CREATE MACRO” in Chapter 1 – “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

The statements in the macro are performed using the EXECUTE statement. See “EXECUTE (Macro Form)” in Chapter 3 – “SQL Data Manipulation Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 6*.

A macro can include an EXECUTE statement that executes another macro.

Definition

A macro consists of one or more statements that can be executed by performing a single statement. Each time the macro is performed, one or more rows of data can be returned.

Performing a macro is similar to performing a multistatement request (see “Multistatement Requests”).

Single-User and Multiuser Macros

You can create a macro for your own use, or grant execution authorization to others.

For example, your macro might enable a user in another department to perform operations on the data in the Teradata RDBMS. When executing the macro, a user need not be aware of the database being accessed, the tables affected, or even the results.

Multistatement Transactions Versus Macros

Although you can enter a multistatement operation interactively using an explicit transaction (either BT/ET or COMMIT), a better practice is to define such an operation as a macro because an explicit transaction holds locks placed on objects by statements in the transaction until the statement sequence is completed with an END TRANSACTION or COMMIT statement.

If you were to enter such a sequence interactively from BTEQ, items in the database would be locked to others while you typed and entered each statement.

Contents of a Macro

A macro can contain a data definition statement only if it is the only SQL statement in that macro. Such a data definition statement is not resolved until the macro is executed, at which time unqualified database object references are fully resolved using the default database of the user submitting the EXECUTE statement.

Therefore, you should fully qualify all object references in a macro data definition statement in the macro body.

A macro can contain parameters that are substituted with data values each time the macro is executed. It also can include a USING modifier, which allows the parameters to be filled with data from an external source such as a disk file. A COLON character prefixes references to a parameter name in the macro.

Executing a Macro

Regardless of the number of statements in a macro, the RDBMS treats it as a single request.

When you execute a macro, either all its statements are processed successfully or none are processed. If a macro fails, it is aborted, any updates are backed out, and the database is returned to its original state.

As a macro is executed, each of the statements acquires the appropriate READ or WRITE lock.

A READ lock enables multiple users to query the same rows at the same time, but prohibits updates.

A WRITE lock prevents queries against data that is being updated.

Ways to Perform SQL Macros in Embedded SQL

Macros in an embedded SQL program are performed in one of the following ways:

IF the macro ...	THEN use ...
is a single statement, and that statement returns no data	- the EXEC statement to specify static execution of the macro or - use the PREPARE and EXECUTE statements to specify dynamic execution. Use DESCRIBE to verify that the single statement of the macro is not a data returning statement.
- consists of multiple statements - returns data	a cursor, either static or dynamic. The type of cursor used depends on the specific macro and on the needs of the application.

Static SQL Macro Execution in Embedded SQL

Static SQL macro execution is associated with a macro cursor using the macro form of the DECLARE CURSOR statement.

When you perform a static macro, you must use the EXEC form to distinguish it from the dynamic SQL statement EXECUTE.

Dynamic SQL Macro Execution in Embedded SQL

Define dynamic macro execution using the PREPARE statement with the statement string containing an EXEC *macro_name* statement rather than a single-statement request.

The dynamic request is then associated with a dynamic cursor. See “DECLARE CURSOR (Macro Form)” in the chapter “SQL Data Manipulation Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 6* for further information on the use of macros.

Deleting a Macro

To delete a macro, use the DROP MACRO statement, described in the chapter “SQL Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4*.

Embedded SQL

You can perform SQL statements from within client application programs. The expression embedded SQL refers to SQL statements performed or declared from within a client application.

An embedded Teradata SQL client program consists of the following:

- Client programming language statements
- One or more embedded SQL statements
- Depending on the host language, one or more embedded SQL declare sections

SQL declare sections are optional in COBOL and PL/I, but *must* be used in C.

A special prefix, EXEC SQL, distinguishes the SQL language statements embedded into the application program from the host programming language.

Embedded SQL statements must follow the rules of the host programming language concerning statement continuation and termination, construction of variable names, and so forth. Aside from these rules, embedded SQL is host language-independent.

Details of Teradata support for embedded SQL are described in *Teradata RDBMS SQL Reference, Volume 6*.

Stored Procedures

Introduction

Stored procedures are called Persistent Stored Modules in the ANSI SQL-99 standard. They consist of a set of control and condition handling statements that make SQL a computationally complete programming language.

These features provide a server-based procedural interface to the Teradata RDBMS for application programmers.

Teradata stored procedure facilities conform to the ANSI SQL-99 standards for semantics.

Elements of Stored Procedures

Teradata stored procedures can contain any of the following elements:

Stored Procedures Elements	Exclusions
Interactive SQL statements	• The following DDL statements: – ALTER PROCEDURE – CREATE PROCEDURE – REPLACE PROCEDURE – DATABASE – EXPLAIN (in any form) – SET ROLE – SET SESSION ACCOUNT – SET SESSION COLLATION – SET SESSION DATEFORM – SET TIME ZONE • Any form of the following Teradata Development tools: – HELP – SHOW • DML <i>except</i> for the following statements: – CHECKPOINT – COMMENT (in any form) – DATABASE – DECLARE CURSOR (Except Selection Form) – DUMP EXPLAIN – ECHO – EXEC – EXECUTE (in any form) – EXPLAIN Modifier – INSERT EXPLAIN – SELECT (Except in cursors) – USING Row Descriptor – WITH...BY Clause • All Preprocessor statements:

Stored Procedures Elements	Exclusions
Dynamic SQL statement specified as a string expression in the following statement: CALL DBC.SysExecSQL (<i>string_expression</i>)	The following DDL statements: • ALTER PROCEDURE • CALL • CREATE PROCEDURE • Cursor statements (OPEN, FETCH, CLOSE) • DATABASE • EXPLAIN • HELP • REPLACE PROCEDURE • SELECT • SELECT - INTO • SET ROLE • SET SESSION ACCOUNT • SET SESSION COLLATION • SET SESSION DATAFORM • SET TIME ZONE • SHOW
Flow of control statements	
Condition handling statements	• RESIGNAL[†] • SIGNAL[†] • UNDO[†]
Parameters, local variables and labels	
Notes	
† Defined by ANSI but not implemented by Teradata for Release 5.0.	

See “SQL Stored Procedures” in *Teradata RDBMS SQL Reference, Volume 6* for information about control statements, parameters, local variables, and labels.

Privileges for Stored Procedures

The security for stored procedures is similar to that for other Teradata database objects like tables, macros, views, and triggers.

The rights to ALTER PROCEDURE, CREATE PROCEDURE, DROP PROCEDURE, and EXECUTE PROCEDURE can be granted using the GRANT statement and revoked using the REVOKE statement. Of these,

- CREATE PROCEDURE is only a database-level privilege.
- ALTER PROCEDURE, DROP PROCEDURE and EXECUTE PROCEDURE privileges can be granted at the object level and database or user level.

- Only DROP PROCEDURE is an automatic privilege for all users. This is granted when a new user or database is created.
- EXECUTE PROCEDURE is an automatic privilege only for the creator of a stored procedure, granted at the time of creation.

Creating Stored Procedures

A stored procedure can be created from BTEQ utility using the COMPILE command, and from CLIV2 applications, ODBC, JDBC, and the QueryMan utility using the SQL CREATE PROCEDURE or REPLACE PROCEDURE statement.

The procedures are stored in the user database space as objects and are executed on the server.

See "SQL Stored Procedures" in *Teradata RDBMS SQL Reference, Volume 6* for the syntax of the CREATE PROCEDURE and REPLACE PROCEDURE statements.

See "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4* for the syntax of the DROP PROCEDURE and RENAME PROCEDURE statements.

Note: The stored procedure definitions in the following examples are designed only to demonstrate the usage of the feature. They are not recommended for use.

Example

Assume you want to define a stored procedure NewProc for adding new employees to the Employee table and retrieve the department name of the department to which the employee belongs. You can also report an error, in case the row that you are trying to insert already exists, and handle that error condition.

The CREATE PROCEDURE statement looks like this:

```
CREATE PROCEDURE NewProc (IN name CHAR(12),
                          IN number INTEGER,
                          IN dept INTEGER,
                          OUT dname CHAR(10),
                          INOUT errstr VARCHAR(30))
BEGIN
  DECLARE CONTINUE HANDLER
    FOR SQLSTATE VALUE '23505'
    SET errstr = 'Duplicate Row.';

  INSERT INTO Employee (Name, EmpNo, DeptNo )
    VALUES (:name, :number, :dept);
  SELECT DeptName
    INTO :dname FROM Department
      WHERE DeptNo = :dept;
END;
```

This stored procedure defines parameters that must be filled in each time it is called. The parameters are indicated with a leading colon (:) character when used in an SQL statement inside the procedure.

Modifying Stored Procedures

You can modify a stored procedure definition using the REPLACE PROCEDURE statement.

Example

You can perform the same tasks as in the previous example using a stored procedure with nested, labeled compound statements.

In the modified procedure, compound statement labeled L3 is nested within the outer compound statement L1. Note that the compound statement labeled L2 is the handler action clause of the condition handler.

The REPLACE PROCEDURE statement looks like this:

```
REPLACE PROCEDURE NewProc (IN name CHAR(12),
                           IN num INTEGER,
                           IN dept INTEGER,
                           OUT dname CHAR(10)
                           INOUT p1 VARCHAR(30))
L1: BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE value '23505'
  L2: BEGIN
    SET p1='Duplicate Row';
    END L2;
  L3: BEGIN
    INSERT INTO Employee (Name, EmpNo, DeptNo)
      VALUES (:name, :num, :dept);

    SELECT DeptName
      INTO :dname FROM Department
        WHERE DeptNo = :dept;
    IF SQLCODE <> 0 THEN LEAVE L3;
    ...

    END L3;
  END L1;
```

Performing Stored Procedures

You can execute a stored procedure from any supporting client utility or interface using the SQL CALL statement. You have to specify arguments for all the parameters contained in the stored procedure.

The CALL statement for executing the procedure created in the CREATE PROCEDURE example looks like this:

```
CALL NewProc (Jonathan, 1066, 34, dname);
```

See “CALL” in the chapter “SQL Data Manipulation Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 6* for details on performing stored procedures and on call arguments.

Recompiling Stored Procedures

The ALTER PROCEDURE feature enables recompilation of stored procedures without having to execute SHOW PROCEDURE and REPLACE PROCEDURE statements.

This new feature provides the following benefits:

- Stored procedures created in earlier releases of Teradata RDBMS can be recompiled in Teradata release V2R5.0 to derive the benefits of new features and performance improvements.
- Recompilation is also useful for cross-platform archive and restoration of stored procedures.

- ALTER PROCEDURE allows changes in the following compile-time attributes of a stored procedure:
 - SPL option
 - Warnings option

Note: Stored procedures created in Teradata release V2R5.0 need to be recompiled in the case of a reversion of the RDBMS to an earlier release.

Deleting Stored Procedures

You can delete a stored procedure from a database using the DROP PROCEDURE statement. Assume you want to drop the NewProc procedure from the database.

The DROP PROCEDURE statement looks like this:

```
DROP PROCEDURE NewProc;
```

Renaming Stored Procedures

You can rename a stored procedure using the RENAME PROCEDURE statement. Assume you want to rename the NewProc stored procedure as NewEmp. The statement looks like this:

```
RENAME PROCEDURE NewProc TO NewEmp;
```

Getting Stored Procedure Information

You can get information about the parameters specified in a stored procedure and their attributes using the HELP PROCEDURE statement. The output contains a list of all the parameters specified in the procedure and the attributes of each parameter. The statement you have to specify is:

```
HELP PROCEDURE NewProc;
```

If you want to view the creation-time attributes of the stored procedure, you have to specify the following statement:

```
HELP PROCEDURE NewProc ATTRIBUTES;
```

Archiving Procedures

Stored procedures are archived and restored as part of a database archive and restoration. Individual stored procedures cannot be archived or restored using the ARCHIVE (DUMP) or RESTORE statements.

Related Information

For details on stored procedures, see “SQL Stored Procedures” in *Teradata RDBMS SQL Reference, Volume 6*.



Chapter 3:

SQL Lexicon

This chapter explains the lexicon for Teradata SQL (Structured Query Language) language, a single, unified, nonprocedural language that provides capabilities for queries, data definition, data modification, and data control of the Teradata RDBMS.

Topics include:

- Names
- Keywords
- Delimiters
- Literals
- SQL operators
- Lexical separators
- Statement separators
- Request terminators

SQL Lexicon Characters

Client Character Data

The characters that make up the SQL lexicon can be represented on the client system in ASCII, EBCDIC, or in an installed user-defined character set.

If the client system character data is not ASCII, then it is converted by the Teradata RDBMS to an internal form for processing and storage. Data returned to the client system is converted to the client character set.

Server Character Data

The internal forms used for character support are described in:

- Chapter 4: “Server Character Sets,” in *Teradata RDBMS International Character Set Support*
- Appendix B: “Japanese Character Sets,” in *Teradata RDBMS International Character Set Support*

The notation used for Japanese characters is described in:

- [“Character Shorthand Notation Used In This Book”](#)
- [Appendix A: “Notation Conventions.”](#)

Names

Introduction

In Teradata SQL, various database objects such as tables, views, stored procedures, macros, columns, and collations are identified by a name.

The set of valid names depends on whether the system is standard, or includes Japanese character support.

Rules

The rules for naming Teradata database objects are as follows.

- You must define and reference each *object* (user, database, table, column, view, macro, stored procedure and checkpoint) by a name.
- In general, names consist of 1 to 30 characters chosen from the following sets.
 - Uppercase or lowercase letters (A to Z and a to z)
 - Digits (0 through 9)
 - The special characters DOLLAR SIGN (\$), NUMBER SIGN (#), and LOW LINE (_)
- Names have the following syntactic restrictions.
 - They must not begin with a digit.
 - They must not be a keyword.
- Names can appear as a sequence of characters within double quotes. There are fewer restrictions on the characters that can be included in such names. The restrictions are described in [“Name Validation For Japanese Language Support” on page 3-6](#).
- Japanese character support allows various Japanese characters to be used for names, but determining the maximum number of characters allowed in a name becomes much more complex (see [“Name Validation For Japanese Language Support” on page 3-6](#)).
- Words having any of the following characteristics are not ANSI SQL-99-compliant:
 - Contains lower case letters.
 - Contains either a \$ or a #.
 - Begins with an underscore.
 - Has more than 18 characters.

- Names that define databases and objects must observe the following rules.
 - Databases must have unique names.
 - Tables, views, stored procedures, or macros in a database must have unique names within that database.
 - Table and view columns must have unique names.
 - Parameters defined for a macro or stored procedure must have unique names.
 - Indexes on a table must have unique names.
 - Named constraints on a table must have unique names.
 - Indexes and constraints can have the same name as the table they are associated with.
- CHECK constraints, REFERENCE constraints, and INDEX objects can also have *assigned* names. Names are optional for these objects.

QUOTATION MARKS Characters and Names

By enclosing names in QUOTATION MARKS characters (U+0022), the valid set of characters for defining names is increased greatly.

Pad characters and special characters can also be included. For example, the following strings are both valid names.

- "Current Salary"
- "D'Augusta"

The QUOTATION MARKS characters are not part of the name, but they *are* required, if the name is not valid otherwise.

For example, these two names are identical, even though one is enclosed within QUOTATION MARKS characters.

- This_Name
- "This_Name"

The NULL character (U+0000) is not allowed in any names, including quoted names.

All of the following examples are valid names.

- Employee
- job_title
- CURRENT_SALARY
- DeptNo
- Population_of_Los_Angeles
- Totaldollars
- "Table A"
- "Today's Date"

Caution: If you use quoted names, the QUOTATION MARKS characters that delineate the names are *not* counted in the length of the name and are *not* stored in Dictionary tables used to track name usage.

If a Dictionary view is used to display such names, they are displayed without the double quote characters, and if the resulting names are used without adding double quotes, the likely outcome is an error report.

For example, “D’Augusta” might be the name of a column in the Dictionary view DBC.Columns, and the HELP statements that return column names return the name as D’Augusta (without being enclosed in QUOTATION MARKS characters).

Case Sensitivity and Names

Names are not case-dependent—a name cannot be used twice by changing its case. Any mix of uppercase and lowercase can be used when referencing symbolic names in a request.

For example, the following statements are identical.

```
SELECT Salary FROM Employee WHERE EmpNo = 10005;
SELECT SALARY FROM EMPLOYEE WHERE EMPNO = 10005;
SELECT salary FROM employee WHERE eMpNo = 10005;
```

The case in which a column name is defined can be important. The column name is the default title of an output column, and symbolic names are returned in the same case in which they were defined.

For example, assume that the columns in the SalesReps table are defined as follows:

```
CREATE TABLE SalesReps
( last_name VARCHAR(20) NOT NULL,
  first_name VARCHAR(12) NOT NULL, ...
```

In response to a query that does not define a TITLE phrase, such as the following example, the column names are returned exactly as defined they were defined, for example, *last_name*, then *first_name*.

```
SELECT Last_Name, First_Name
FROM SalesReps
ORDER BY Last_Name;
```

You can use the TITLE phrase to specify the case, wording, and placement of an output column heading either in the column definition or in an SQL statement.

For more information, see “TITLE Phrases” in the section “WITH Clause” in the chapter “The SELECT Statement” of *Teradata RDBMS SQL Reference, Volume 6*.

Name Validation For Japanese Language Support

Rules

Japanese language support allows thousands of additional characters to be used for names, but also introduces some restrictions. For unquoted names the following characters may be used in names when Japanese Language support is enabled.

- Any character valid in an unquoted name under standard language support.
 - Uppercase or lowercase letters (A to Z and a to z)
 - Digits (0 through 9)
 - The special characters dollar sign (\$), pound sign (#), and underscore (_)
- The fullwidth (zenkaku) versions of the characters valid for names under standard language support.
 - Fullwidth uppercase or lowercase letters (A to Z and a to z)
 - Fullwidth digits (0 through 9)
 - The special characters fullwidth dollar sign (\$), fullwidth pound sign (#), and fullwidth underscore (_)
- Fullwidth (zenkaku) and halfwidth (hankaku) Katakana characters and sound marks.
- Hiragana characters.
- Kanji characters from JIS-x0208.

The following list describes the restrictions on names under Japanese character support.

- The extra characters available using quoted names are restricted.
- The length of a name is restricted in a complex fashion.

Charts of the supported Japanese character sets, the Teradata RDBMS internal encodings, the valid character ranges for Japanese object names and data, and the non-valid character ranges for Japanese data are given in Appendix B: "Japanese Character Sets," in *Teradata RDBMS International Character Set Support*. The non-valid character ranges for Japanese object names are given in Appendix C: "Non-Valid Japanese Character Codes," in *Teradata RDBMS International Character Set Support*.

Cross-Platform Integrity

If you need to access metadata from heterogeneous clients, you should restrict their names to those allowed under standard language support.

Character Ranges for KanjiEBCDIC Object Names

The following passages describe the JIS characters that are valid and non-valid for KanjiEBCDIC object names.

JIS Character Set	Relevant Passages
JIS-x0201	Appendix B: "Japanese Character Sets," in <i>Teradata RDBMS International Character Set Support</i>, documents the valid ranges for the KATAKANAEBCDIC character set for JIS-x0201. The same appendix documents the valid ranges for the KANJI EBCDIC5026_0I character set for JIS-x0201. Neither set permits Katakana symbols 0x41—0x45, nor Latin symbols other than \$, #, and _. Appendix B: "Japanese Character Sets," in <i>Teradata RDBMS International Character Set Support</i>, also documents the valid ranges for the KANJI EBCDIC5035_0I character set for JIS-x0201. This set does not permit Katakana symbols 0x42—0x46 nor Latin symbols other than \$, #, and _.
JIS-x0208	Appendix B: "Japanese Character Sets," in <i>Teradata RDBMS International Character Set Support</i>, documents the valid ranges for JIS-x0208. Note that characters in the User and Marker areas in table are not allowed. Also, many KanjiEBCDIC characters cannot be mapped to JIS-x0208. A list of these characters is provided in Appendix C: "Non-Valid Japanese Character Codes," in <i>Teradata RDBMS International Character Set Support</i>.

Character Ranges for KanjiEUC Object Names

The following table describes the JIS characters that are valid and non-valid when creating metadata under the KanjiEUC character set.

JIS Character Set	Relevant Passages in the Book
JIS-x0201	The valid ranges for JIS-x0201 characters in KanjiEUC metadata can be found in rows cs0 and cs2 as documented in Appendix B: “Japanese Character Sets,” in <i>Teradata RDBMS International Character Set Support</i> . The set does not permit Katakana symbols 0x8EA1—0x8EA5 nor Latin symbols other than \$, #, and _.
JIS-x0208	The valid ranges for JIS-x0208 are given in row cs1 of the “EUC Code Set Localization” table. However, note that within the JIS-x0208 ranges, there are unassigned characters that are not allowed in metadata; for example, the unassigned character 0xA9A1 is illegal. In addition, certain characters that are valid in JIS-x0208 do not map to the KanjiEBCDIC encoding and are not allowed at all under KanjiEUC. These are shown in Appendix C: “Non-Valid Japanese Character Codes,” in <i>Teradata RDBMS International Character Set Support</i>.
JIS-x0212	The JIS-x0212 characters found in row cs3 of the JIS-x0212 table in Appendix B: “Japanese Character Sets,” in <i>Teradata RDBMS International Character Set Support</i> , are not allowed in KanjiEUC metadata.

Character Ranges for KanjiShift-JIS Object Names

The following table describes the JIS characters that are valid and non-valid when creating metadata under the KanjiShift-JIS character set.

JIS Character Set	Relevant Passages in the Book
JIS-x0201	The valid ranges for JIS-x0201 characters in KanjiShift-JIS metadata are listed in Appendix B: “Japanese Character Sets,” in <i>Teradata RDBMS International Character Set Support</i>. The set does not permit Katakana symbols 0x82A1 through 0x82A5 nor Latin symbols other than \$, #, and _.
JIS-x0208	The valid ranges for JIS-x0208 are detailed in Appendix B: “Japanese Character Sets,” in <i>Teradata RDBMS International Character Set Support</i>. Note the following issues. - The following characters, listed in Appendix B: “Japanese Character Sets,” in <i>Teradata RDBMS International Character Set Support</i>, are not allowed in metadata. - Gaiji, with first byte values 0xF0-0xF9 - IBM-defined characters with first byte values 0xFA-0xFC. - Within the JIS-x0208 ranges, there are unassigned characters that are not allowed in object names. For example, the unassigned character 0x8540 is illegal. - Certain characters that are valid in JIS-x0208 do not map to the KanjiEBCDIC encoding and are not allowed under KanjiShift-JIS. These are given in Appendix C: “Non-Valid Japanese Character Codes,” in <i>Teradata RDBMS International Character Set Support</i>.

Calculating the Length of a Name

The length of a name is measured by the physical bytes of its internal representation, not by the number of viewable characters. Under the KanjiEBCDIC character sets, the Shift-Out and Shift-In characters that delimit a multibyte character string are included in the byte count.

For example, the following table name contains six logical characters of mixed single byte characters/multibyte characters, defined during a KanjiEBCDIC session:

`<TAB1>QR`

All single byte characters, including the Shift-Out/Shift-In characters, are translated into the Teradata RDBMS internal encoding, based on JIS-x0201. Under the KanjiEBCDIC character sets, all multibyte characters remain in the client encoding.

Thus, the processed name is stored as a string of twelve bytes, padded on the right with the single byte space character to a total of 30 bytes.

The internal representation is as follows:

```
0E 42E3 42C1 42C2 42F1 0F 51 52 20 20 20 20 20 20 20 20 20 20 ...
<   T   A   B   1   > Q R
```

To ensure upgrade compatibility, an object name created under one character set cannot exceed 30 bytes in *any* supported character set.

For example, a single Katakana character occupies 1 byte in KanjiShift-JIS. However, when KanjiShift-JIS is converted to KanjiEUC, each Katakana character occupies two bytes. Thus, a 30-byte Katakana name in KanjiShift-JIS would expand in KanjiEUC to 60 bytes, which is illegal.

The formula for calculating the correct length of an object name is as follows:

$$\text{Length} = \text{ASCII} + (2 * \text{KANJI}) + \text{MAX} (2 * \text{KATAKANA}, (\text{KATAKANA} + 2 * \text{S2M} + 2 * \text{M2S}))$$

where:

This variable ...	Represents the number of ...
ASCII	single byte ASCII characters in the name.
KATAKANA	single byte Hankaku Katakana characters in the name.
Kanji	double byte characters in the name from the JIS-x0208 standard.
S2M	transitions from ASCII, or KATAKANA, to JIS-x0208.
M2S	transitions from JIS-x0208 to ASCII or KATAKANA.

How Validation Occurs

Name validation occurs when the object is created or is given a new name, as follows:

- User names, database names, and account names are verified during the CREATE/MODIFY USER and CREATE/MODIFY DATABASE statements.
- Table names and column names are verified during the CREATE/ALTER TABLE and RENAME TABLE statements. View and macro names are verified during the CREATE/RENAME VIEW and CREATE/RENAME MACRO statements.
Stored procedure names are verified during the execution of CREATE/RENAME/REPLACE PROCEDURE statements.
- Alias object names used in the SELECT, UPDATE, and DELETE statements are verified. The validation occurs only when the SELECT statement is used in a CREATE/REPLACE VIEW statement, and when the SELECT, UPDATE, or DELETE TABLE statement is used in a CREATE/REPLACE MACRO statement.
- Names of work tables and error tables are validated by the MultiLoad and FastLoad client utilities. See *Teradata RDBMS Utilities*.

Examples of Validating Japanese Object Names

The following tables illustrate valid and non-valid metadata under the Japanese character sets: KanjiEBCDIC, KanjiEUC, and KanjiShift-JIS.

KanjiEBCDIC Metadata Examples

Name	ASCII	Katakana	Kanji	S2M	M2S	LEN	Result
<ABCDEFGHJKLMN>	0	0	14	1	1	32	Not valid because LEN > 30.
<ABCDEFGHIJ>kl<MN>	2	0	12	2	2	34	Not valid because LEN > 30.
<ABCDEFGHIJ>kl<>	2	0	10	2	2	30	Not valid because consecutive SO and SI characters are not allowed.
<ABCDEFGHIJ><K>	0	0	11	2	2	30	Not valid because consecutive SI and SO characters are not allowed.
ABCDEFGHJKLMNO	0	15	0	0	0	30	Valid.
<ABCDEFGHIJ> <u>KLMNO</u>	0	5	10	1	1	30	Valid.
<Δ>	0	0	1	1	1	6	Not valid because the double byte space is not allowed.

KanjiEUC Metadata Examples

Name	ASCII	Katakana	Kanji	S2M	M2S	LEN	Result
ABCDEFGHJKLM	6	0	7	3	3	32	Not valid because LEN > 30 bytes.
ABCDEFGHJKLM	6	0	7	2	2	28	Valid.
ss ₂ <u>A</u> BCDEFGHIJKL	0	1	11	1	1	27	Valid.
A _{ss2} <u>B</u> CDEFGHIJKL	0	1	11	2	2	31	Not valid because LEN > 30 bytes.
ss ₃ <u>C</u>	0	0	0	1	1	4	Not valid because characters from code set 3 are not allowed.

KanjiShift-JIS Metadata Examples

Name	ASCII	Katakana	Kanji	S2M	M2S	LEN	Result
ABCDEFGHIJK <u>LMNOPQR</u>	6	7	5	1	1	30	Valid.
ABCDEFGHIJK <u>LMNOPQR</u>	6	7	5	2	2	31	Not valid because LEN > 30 bytes.

Metadata Translation and Storage

Metadata (object names) is stored in the dictionary tables using the following translation conventions:

Character Type	Description								
Single byte	All single byte characters in a name, including the KanjiEBCDIC Shift-Out/Shift-In characters, are translated into the Teradata internal representation (based on JIS-x0201 encoding).								
Multibyte	Multibyte characters in object names are handled according to the character set in effect for the current session, as follows: <table><tr><th>Multibyte Character Set</th><th>Description</th></tr><tr><td>KanjiEBCDIC</td><td>Each multibyte character within the Shift-Out/Shift-In delimiters is stored without translation; that is, it remains in the client encoding. The name string must have matched (but not consecutive) Shift-Out and Shift-In delimiters.</td></tr><tr><td>KanjiEUC</td><td> Under code set 1, each multibyte character is translated from KanjiEUC to KanjiShift-JIS. Under code set 2, byte ss2 (0x8E) is translated to 0x80; the second byte is left unmodified. This translation preserves the relative ordering of code set 0, code set 1, and code set 2. </td></tr><tr><td>KanjiShift-JIS</td><td>Each multibyte character is stored without translation; it remains in the client encoding.</td></tr></table>	Multibyte Character Set	Description	KanjiEBCDIC	Each multibyte character within the Shift-Out/Shift-In delimiters is stored without translation; that is, it remains in the client encoding. The name string must have matched (but not consecutive) Shift-Out and Shift-In delimiters.	KanjiEUC	Under code set 1, each multibyte character is translated from KanjiEUC to KanjiShift-JIS. Under code set 2, byte ss2 (0x8E) is translated to 0x80; the second byte is left unmodified. This translation preserves the relative ordering of code set 0, code set 1, and code set 2.	KanjiShift-JIS	Each multibyte character is stored without translation; it remains in the client encoding.
Multibyte Character Set	Description								
KanjiEBCDIC	Each multibyte character within the Shift-Out/Shift-In delimiters is stored without translation; that is, it remains in the client encoding. The name string must have matched (but not consecutive) Shift-Out and Shift-In delimiters.								
KanjiEUC	Under code set 1, each multibyte character is translated from KanjiEUC to KanjiShift-JIS. Under code set 2, byte ss2 (0x8E) is translated to 0x80; the second byte is left unmodified. This translation preserves the relative ordering of code set 0, code set 1, and code set 2.								
KanjiShift-JIS	Each multibyte character is stored without translation; it remains in the client encoding.								

Both the ASCII character set and the EBCDIC character set are stored on the server as ASCII.

Metadata Comparisons

Comparison Rules

In comparing two names, the following rules apply.

- A simple Latin lowercase letter is equivalent to its corresponding simple Latin uppercase letter. For example, 'a' is equivalent to 'A'.
- Multibyte characters that have the same logical presentation but have different physical encodings under different character sets do not compare as equivalent.
- Two names compare as identical when their internal hexadecimal representations are the same, even if their logical meanings are different under the originating character sets.

Note that identical characters on keyboards connected to different clients are not necessarily identical in their internal encoding on the Teradata RDBMS. The Teradata RDBMS could interpret two logically identical names as different names if the character sets under which they were created are not the same.

For example, the following strings illustrate the internal representation of two names, both of which were defined with the same logical multibyte characters. However, the first name was created under KanjiEBCDIC, and the second name was created under KanjiShift-JIS.

```
KanjiEBCDIC:      0E 42E3 42C1 42C2 42F1 0F 51 52
KanjiShift-JIS:   8273 8260 8261 8250 D8 D9
```

In order to ensure upgrade compatibility, you must avoid semantically duplicate object names in situations where duplicate object names would not normally be allowed.

Also, two different character sets might have the same internal encoding for two logically different multibyte characters. Thus, two names might compare as identical even if their logical meanings are different.

Using the Internal Hexadecimal Representation of a Name

The Teradata RDBMS knows metadata by its internal hexadecimal representation, and this is how it is stored in the various system tables of the Data Dictionary.

The encoding of the internal representation of metadata depends on the components of the name string (are there single byte characters, multibyte characters, or both; are there Shift Out/Shift In (SO/SI) characters, and so on) and the character set in effect when the name was created.

Assume that a user under one character set needs to reference metadata created by a user under a different character set. If the current user attempts to reference the name with the actual characters (that is, by typing the characters or by selecting non-specific entries from a dictionary table), the access could fail or the returned name could be meaningless.

For example, assume that User_1 invokes a session under KanjiEBCDIC and creates a table name with multibyte characters.

User_2 invokes a session under KanjiEUC and issues the following statement.

```
SELECT TableName
FROM DBC.Tables
```

The result returns the KanjiEBCDIC characters in KanjiEUC presentation, which probably does not make sense.

You can avoid this problem by creating and using metadata in the following ways.

- Create names using only simple single byte Latin letters (A...Z, a...z) digits, and the dollar sign, pound sign, and underscore symbols.
Because these characters always translate to the same internal representation, they display exactly the same presentation to any session, regardless of the client or the character set.
- Use the following syntax to reference a name by its internal representation.

```
—— 'hexadecimal_digit(s)' —— XN ——  
HH01A099
```

where:

Syntax element ...	Specifies ...
'hexadecimal_digits'	a quoted hexadecimal string representation of the Teradata RDBMS internal encoding.

The key letters XN specify that the string is a hexadecimal name.

Example

The following table name, which contains mixed single byte characters and multibyte characters, was created under a KanjiEBCDIC character set:

```
<TAB1>KAN
```

The client encoding in which this name was received is as follows:

```
0E 42E3 42C1 42C2 42F1 0F D2 C1 D5
< T A B 1 > K A N
```

The single byte characters (the letters K, A, and N, and the SO/SI characters) were translated into internal JIS-x0201 encoding. The multibyte characters were not translated and remained in the host encoding.

The resulting internal string by which the name was stored is as follows:

```
0E 42E3 42C1 42C2 42F1 0F 4B 41 4E
< T A B 1 > K A N
```

To access this table from a KanjiShift-JIS or KanjiEUC character set, you could use the following Teradata SQL statement:

```
SELECT *
FROM '0E42E342C142C242F10F4B414E' XN;
```

The response would be all rows from table <TAB1>KAN.

Finding the Internal Hexadecimal Representation for Metadata

Introduction

The CHAR2HEXINT function converts a character string to its internal hexadecimal representation. You can use this function to find the internal representation of any Teradata RDBMS name.

For more information on CHAR2HEXINT, see “CHAR2HEXINT Function” in Chapter 8 – “Character String Functions” of *Teradata RDBMS SQL Reference, Volume 5*.

Example 1

For example, to find the internal representation of all Teradata RDBMS table names, issue the following Teradata SQL statement.

```
SELECT CHAR2HEXINT(T.TableName) (TITLE 'Internal Hex  
Representation of TableName'),T.TableName (TITLE 'TableName')  
FROM DBC.Tables T  
WHERE T.TableKind = 'T'  
ORDER BY T.TableName;
```

This statement selects all rows from the DBC.Tables view where the value of the TableKind field is T.

For each row selected, both the internal hexadecimal representation and the character format of the value in the TableName field are returned, sorted alphabetically.

An example of a portion of the output from this statement is shown below. In this example, the first name (double byte-A) was created using the KanjiEBCDIC character set.

[illegible]

Note that the first name, <double byte A>, cannot be interpreted. To obtain a printable version of a name, you must log onto a session under the same character set under which the name was created.

Example 2

You can use the same syntax to obtain the internal hexadecimal representations of all views or all macros.

To do this, modify the WHERE condition to TableKind='V' for views and to TableKind='M' for macros.

To obtain the internal hexadecimal representation of all database names, you can issue the following statement:

```
SELECT CHAR2HEXINT(D.DatabaseName) (TITLE 'Internal Hex  
Representation of DatabaseName'),D.DatabaseName (TITLE  
'DatabaseName')  
FROM DBC.Databases D  
ORDER BY D.DatabaseName;
```

This statement selects every DatabaseName from DBC.Databases. For each DatabaseName, it returns the internal hexadecimal representation and the name in character format, sorted by DatabaseName.

An example of the output from this statement is as follows:

[illegible]

Example 3

Note that these statements return the padded hexadecimal name. The value 0x20 represents a space character in the internal representation.

You can use the TRIM function to obtain the hexadecimal values without the trailing spaces, as follows.

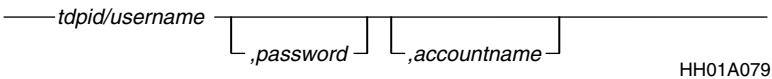
```
SELECT CHAR2HEXINT(TRIM(T.TableName)) (TITLE 'Internal Hex  
Representation of TableName'),T.TableName (TITLE 'TableName')  
FROM DBC.Tables T  
WHERE T.TableKind = 'T'  
ORDER BY T.TableName;
```

Specifying Names in a Logon String

Purpose

Identifies a user to the Teradata RDBMS and, optionally, permits the user to specify a particular account to log onto.

Syntax



where:

Syntax element ...	Specifies ...
<i>tdp_id/user_name</i>	the client TDP the user wishes to use to communicate with the Teradata RDBMS and the name by which the RDBMS knows the user. The <i>user_name</i> parameter can contain mixed single byte/multibyte characters if the current character set permits them.
<i>password</i>	an optional (depending on how the user is defined) password required to gain access to the Teradata RDBMS. The <i>password</i> parameter can contain mixed single byte/multibyte characters if the current character set permits them.
<i>account_name</i>	an optional account name or account string that specifies a particular user account or account and performance-related variable parameters the user can use to tailor the session being logged onto. The <i>account_name</i> parameter can contain mixed single byte/multibyte characters if the current character set permits them.

The Teradata RDBMS does not support the hexadecimal representation of a *user_name*, a *password*, or an *account_id* in a logon string.

For example, if you attempt to log on as user DBC by entering '444243'XN, the logon is not successful and an error message is generated.

Passwords

The password format options allows the site administrator to change the minimum and maximum number of characters allowed in the password string, and control the use of digits and special characters.

Password string rules are identical to those for naming objects. See [“Name Validation For Japanese Language Support”](#) on page 3-6.

The password formatting feature does not apply to Japanese character sets.

Keywords

Introduction

Keywords are words that have special meanings in SQL statements. There are two types of keywords: *reserved* and *non-reserved*. You cannot use reserved keywords to name database objects. Although you can use non-reserved keywords as object names, you should not because of possible confusion resulting from their use.

The first keyword in a SQL statement is usually a verb.

For example, in the INSERT statement, the first keyword is INSERT.

Other keywords appear throughout a statement as modifiers (for example, DISTINCT, PERMANENT), or as words that introduce clauses (for example, IN, AS, AND, TO, WHERE).

In this book, keywords are shown entirely in uppercase letters, though SQL does not discriminate between uppercase and lowercase letters in a keyword.

For example, the following SELECT statements are interpreted to be identical by SQL.

```
Select Salary from Employee where EmpNo = 10005;  
SELECT Salary FROM Employee WHERE EmpNo = 10005;  
select Salary fRom Employee WhErE EmpNo = 10005;
```

For a list of Teradata SQL keywords, see [Appendix B: “Restricted Words for V2R5.0.”](#)

All keywords must be from the ASCII repertoire. Fullwidth letters are not valid regardless of the character set being used.

Entry-level ANSI requires that all keywords be expressed as uppercase.

Keywords and Object Names

Note that you cannot use reserved keywords to name database objects. Because new keywords are frequently added to new releases of the Teradata RDBMS, you may experience a problem with database object names that were valid in prior releases but which become nonvalid in a new release.

The workaround for this is to do one of the following things:

- Put the newly nonvalid name in double quotes.
- Rename the object.

In either case you must change your applications.

Delimiters

Introduction

Delimiters are special characters whose meaning depends on context.

The function of each delimiter is described in the following table.

Delimiter	Name	Purpose
()	LEFT PARENTHESIS RIGHT PARENTHESIS	Group expressions and define the limits of various phrases.
,	COMMA	Separates items in a list, or as a date separator.
:	COLON	Prefixes a referenced parameter or client system variable. Also used as a date separator.
.	FULLSTOP	Separates a database name from a table name, or a table name from a column name. In numeric constants, the period is the decimal point. Also used as a date separator.
;	SEMICOLON	Separates statements in a request. In BTEQ, a semicolon at the end of an input line terminates a request.
'	APOSTROPHE	Defines the boundaries of character string constants. Also used as a date separator.
"	QUOTATION MARK	Defines the boundaries of nonstandard names.
@	COMMERCIAL AT	Obsolete—replaced by : (colon)
/	SOLIDUS	Used as a date separator.
B b	Uppercase B Lowercase b	Used as a date separator.
-	HYPHEN- MINUS	Used as a date separator.

Example

In the following statement, the FULLSTOP separates the database name (Examp and Personnel) from the table name (Profile and Employee), and, where reference is qualified to avoid ambiguity, it separates the table name (Profile, Employee) from the column name (DeptNo).

```
UPDATE Examp.Profile SET FinGrad = 'A'
WHERE Name = 'Phan A' ; SELECT EdLev, FinGrad, JobTitle,
YrsExp FROM Examp.Profile, Personnel.Employee
WHERE Profile.DeptNo = Employee.DeptNo ;
```

The first SEMICOLON separates the UPDATE statement from the SELECT statement. The second SEMICOLON terminates the entire multistatement request.

The semicolon is not a delimiter in ANSI SQL: it is required in Teradata SQL only to separate multiple statements (either in a request or macro definition) or to terminate a request submitted through BTEQ.

Literals

Literals are values coded directly in the text of an SQL statement, view or macro definition text, or CHECK constraint definition text. In general, the system is able to determine the type of a literal by its form.

Kinds of Literals

The Teradata RDBMS supports several kinds of literals:

- Numeric literals (or constants), of which there are three types:
 - Integer
 - Decimal
 - Floating point
- Date and time literals (also called DateTime literals), of which there are 3 types:
 - Date
 - Time
 - Timestamp
- Interval literals
- Character data literals
- Graphic literals
- Hexadecimal literals

Numeric Literals

Definition

A numeric literal (also referred to as a constant) is a character string of 1 to 20 characters selected from the following:

- digits 0 through 9
- plus sign
- minus sign
- decimal point

Types of Numeric literals

There are three types of numeric literals:

- Integers
- Decimals
- Floating point numbers

Integer Literals

An integer literal declares literal strings of integer numbers in an expression, view or macro definition text, or CHECK constraint text. Integer literals consist of an optional sign followed by a sequence of digits.

For detailed information, see "INTEGER Literals" in Chapter 2 - Data Literals" of *Teradata RDBMS SQL Reference, Volume 3*.

Decimal Literals

A decimal literal strings of decimal numbers in an expression, view or macro definition text, or CHECK constraint text.

DECIMAL literals consist of the following components, reading from left-to-right: an optional sign, a sequence of digits (including none), a decimal point, an optional sequence of digits. The scale and precision of a DECIMAL literal are determined by the total number of digits in the literal and the number of digits to the right of the decimal point, respectively.

For detailed information, see "DECIMAL Literals" in Chapter 2 - "Data Literals" of *Teradata RDBMS SQL Reference, Volume 3*.

Floating Point Literals

A floating point literal declares literal strings of floating point numbers in an expression, view or macro definition text, or CHECK constraint text.

Floating point literals consist of the following components, reading from left-to-right: an optional sign, a sequence of digits (including none) representing the whole number portion of the mantissa, an optional decimal point, a sequence of digits (including none) representing the fractional portion of the mantissa, the literal character E, an optional sign, a sequence of digits (including none) representing the exponent.

For detailed information, see "FLOATING POINT Literals" in the chapter "Data Literals" of *Teradata RDBMS SQL Reference, Volume 3*.

DateTime Literals

Purpose

Date and time literals declare date, time, or timestamp values in a SQL expression, view or macro definition text, or CONSTRAINT definition text.

Types of DateTime literals

There are three types of DateTime literals:

- Date
- Time
- Timestamp

Date Literals

A date literal declares a date value in ANSI DATE format in an expression, view or macro definition text, or CHECK constraint text. ANSI DATE literal is the preferred format for DATE constants. All DATE operations accept this format.

For detailed information, see "DATE Literals" in the chapter "Data Literals" of *Teradata RDBMS SQL Reference, Volume 3*.

Time Literals

A time literal declares a time value in an expression, view or macro definition text, or CHECK constraint text.

For detailed information, see "TIME Literals" in Chapter 2 - "Data Literals" of *Teradata RDBMS SQL Reference, Volume 3*.

Timestamp Literals

A timestamp literal declares a timestamp value in an expression, view or macro definition text, or CHECK constraint text.

For detailed information, see "TIMESTAMP Literals" in the chapter "Data Literals" of *Teradata RDBMS SQL Reference, Volume 3*.

Interval Literals

Purpose

Interval literals provide a means for declaring values for DateTime interval values in expressions, view or macro definition text, or CHECK constraint text.

Interval literals differ from other SQL literals in that they are introduced by keywords and also followed by keywords.

Types of Interval Literals

There are two mutually exclusive categories of interval literals.

- Year-Month
- Day-Time

Year-Month Types

Year-month interval literals include the following interval literals:

- YEAR
- YEAR TO MONTH
- MONTH

For detailed information, see the appropriate section in the chapter "Data Literals" in *Teradata RDBMS SQL Reference, Volume 3*.

Day-Time Types

Date-time interval literals include the following interval literals:

- DAY
- DAY TO HOUR
- DAY TO MINUTE
- DAY TO SECOND
- HOUR
- HOUR TO MINUTE
- HOUR TO SECOND
- MINUTE
- MINUTE TO SECOND
- SECOND

For detailed information, see the appropriate section in the chapter "Data Literals" in *Teradata RDBMS SQL Reference, Volume 3*.

Character Data Literals

Purpose

A character data literal declares a character string literal value in an expression, view or macro definition text, or CHECK constraint definition text.

Character literals consist of 0 to 31000 bytes delimited by a matching pair of single quotes. A zero-length character literal is represented by two consecutive single quotes ('').

For detailed information, see "CHARACTER DATA Literals" in the chapter "Data Literals" in *Teradata RDBMS SQL Reference, Volume 3*.

Graphic Literals

Purpose

A graphic literal declares a GRAPHIC character string literal in an expression, view, or macro definition text or CHECK constraint definition text.

For detailed information on INTERVAL Literals, see "GRAPHIC Literals" in the chapter "Data Literals" in *Teradata RDBMS SQL Reference, Volume 3*.

Hexadecimal Literals

Purpose

A hexadecimal literal declares the character set associated with an SQL internal format hexadecimal literal in an expression. All values are converted to UNICODE with the exception of the KANJI1 form, which remains encoded as entered.

For detailed information, see "Hexadecimal Literals" in the chapter "Data Literals" in *Teradata RDBMS SQL Reference, Volume 3*.

Built-In Functions Used As Literals

Introduction

The built-in, or special register function literals, which are niladic (have no arguments), return various information about the system and can be used like other literals within SQL expressions. In an SQL query, the appropriate system value is substituted for the literal by the Parser.

The Teradata built-in functions are documented in the chapter “Built-In Functions” in *Teradata RDBMS SQL Reference, Volume 5*.

Available Built-In Functions

Available built-in functions include all of the following:

- ACCOUNT
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- DATABASE
- DATE
- PROFILE
- ROLE
- SESSION
- TIME
- USER

NULL as a Literal

Introduction

The keyword NULL is sometimes available as a special construct similar to, but not identical with, a literal. In this case, the keyword NULL represents null.

ANSI Compliance

NULL is ANSI SQL-99-compliant with extensions.

Using NULL as a Literal

NULL as a literal can be used in the following ways.

- A CAST source operand, for example:

```
SELECT CAST (NULL AS DATE) ;
```
- A CASE result, for example:

```
SELECT CASE WHEN orders = 10 THEN NULL END FROM sales_tbl;
```
- A referential specification, for example:

```
SET NULL
```
- An insert item specifying a null is to be placed in a column position on INSERT.
- An update item specifying a null is to be placed in a column position on UPDATE.
- A default column definition specification, for example:

```
DEFAULT NULL
```
- An explicit SELECT item, for example:

```
SELECT NULL
```

This is a Teradata extension to ANSI.
- An operand of a function, for example:

```
SELECT TYPE (NULL)
```

This is a Teradata extension to ANSI.

Data Type of NULL

When you use NULL as an explicit SELECT item or as the operand of a function, its data type is INTEGER. In all other cases NULL has *no* data type because it has no value.

For example, if you perform SELECT TYPE(NULL), then INTEGER is returned as the data type of NULL.

SQL Operators

Introduction

SQL operators are used to express logical and arithmetic operations. Operators of the same precedence are evaluated from left to right. See [“SQL Operations and Precedence” on page 3-36](#) for more detailed information.

Parentheses can be used to control the order of precedence. When parentheses are present, operations are performed from the innermost set of parentheses outward.

Definitions

The following definitions apply to SQL operators.

Term	Definition
numeric	Any literal, data reference, or expression having a numeric value.
string	Any character string or string expression.
logical	A Boolean expression (resolves to either TRUE or FALSE).
value	Any numeric, character, or byte data item.
set	A collection of values returned by a subquery, or a list of values separated by commas and enclosed by parentheses.

SQL Operations and Precedence

SQL operations, and the order in which they are performed when no parentheses are present, are listed in the following table. Operators of the same precedence are evaluated from left to right.

Precedence	Result Type	Operation
highest	numeric	+ numeric (unary plus) - numeric (unary minus)
intermediate	numeric	numeric ** numeric (exponentiation)
	numeric	numeric * numeric (multiplication) numeric / numeric (division) numeric MOD numeric (modulo operator)
	numeric	numeric + numeric (addition) numeric - numeric (subtraction)
	string	concatenation operator
	logical	value EQ value value NE value value GT value value LE value value LT value value GE value value IN set value NOT IN set value BETWEEN value AND value character value LIKE character value
	logical	NOT logical
	logical	logical AND logical
lowest	logical	logical OR logical

Lexical Separators

Definition

A lexical separator is a character string that can be placed between words, literals, and delimiters without changing the meaning of a statement.

Valid lexical separators are any of the following.

- Comments
- Pad characters (several pad characters are treated as a single pad character except in a string literal)
- RETURN characters (X'0D')

Comment lexical separators are complicated enough to warrant further explanation.

SQL Statement Comments

You can embed comments within an SQL request anywhere a blank can occur.

The SQL parser and the preprocessor recognize the following types of embedded comments:

- Bracketed
- Simple

Simple Comments

The simple form of a comment is delimited by two consecutive HYPHEN-MINUS (U+002D) characters (--) at the beginning of the comment and the newline character at the end of the comment.

The newline character is implementation-specific, but is typed by pressing the Enter (non-3270 terminals) or Return (3270 terminals) key.

Simple SQL comments cannot span multiple lines.

Example

The following SELECT statement illustrates the use of a simple comment.

```
SELECT EmpNo, Name FROM Payroll_Test
ORDER BY Name -- Alphabetic order
;
```

Bracketed Comments

A bracketed comment is a text string of unrestricted length that is delimited by the beginning comment characters SOLIDUS (U+002F) and ASTERISK (U+002A) /* and the end comment characters ASTERISK and SOLIDUS */.

Bracketed comments can begin anywhere on an input line and can span multiple lines.

Example

The following CREATE TABLE statement illustrates the use of a bracketed comment.

```
CREATE TABLE Payroll_Test  /* This is a test table
                             set up to process actual payroll data on a test
                             basis.  The data generated from this table will
                             be compared with the existing payroll system
                             data for 2 months as a parallel test. */
(EmpNo  INTEGER          NOT NULL FORMAT 'ZZZZ9',
 Name   VARCHAR(12) NOT NULL,
 DeptNo INTEGER          FORMAT 'ZZZZ9',
 .
 .
 .
```

Comments With Multibyte Character Set Strings

You can include multibyte character set strings in both simple and bracketed comments.

When using mixed mode in comments, you must have a properly formed mixed mode string, which means that a Shift-In (SI) must follow its associated Shift-Out (SO).

If an SI does not follow the multibyte string, the results are unpredictable.

When using bracketed comments that span multiple lines, the SI must be on the same line as its associated SO.

If the SI and SO are not on the same line, the results are unpredictable.

You *must* specify the bracketed comment delimiters (/* and */) as single byte characters.

Statement Separator (Teradata)

Definition

The semicolon is a Teradata statement separator.

Each statement of a multistatement request must be separated from any subsequent statement with a semicolon.

The following multistatement request illustrates the use of the semicolon as a statement separator.

```
SHOW TABLE Payroll_Test ; INSERT INTO Payroll_Test  
(EmpNo, Name, DeptNo) VALUES ('10044', 'Jones M',  
'300') ; INSERT INTO ...
```

For statements entered using BTEQ, a request terminates with an input line-ending semicolon unless that line is a comment line, beginning with two dashes.

Everything to the right of the - - is a comment. In this case, the semicolon must be on the following line.

ANSI Compliance

The SEMICOLON as a statement separator is a Teradata extension to the ANSI SQL-99 standard.

Null Statement

A SEMICOLON that precedes the first (or only) statement of a request is taken as a null statement. A null statement may or may not be preceded by a comment.

Example1

The semicolon in the following request is a null statement.

```
/* This example shows a comment followed by  
a semicolon used as a null statement */  
; UPDATE Pay_Test SET ...
```

Example 2

The first SEMICOLON in the following request is a null statement. The second SEMICOLON is taken as statement separator:

```
/*      This example shows a semicolon used as a null
        statement and as a statement separator */
;  UPDATE Payroll_Test SET Name = 'Wedgewood A'
    WHERE Name = 'Wedgewood A'
;    SELECT ...
-- This example shows the use of an ANSI component
-- used as a null statement and statement separator ;
```

Request Terminator (Teradata)

Definition

The SEMICOLON is a Teradata request terminator when it is the last non-blank character on an input line in BTEQ unless that line is a comment line beginning with two dashes. In this case, the SEMICOLON request terminator should be on the following line, after the comment line.

A request is considered complete when either the “End of Text” character or the request terminator character is detected.

ANSI Compliance

The SEMICOLON as a request terminator is Teradata extension to the ANSI SQL-99 standard.

Example

For example, on the following input line:

```
SELECT *  
FROM Employee ;
```

the SEMICOLON terminates the single-statement request “SELECT * FROM Employee”.

BTEQ uses SEMICOLONS to terminate multistatement requests.

A request terminator is mandatory for request types that are:

- In the body of a macro
- Triggered action statements in a trigger definition
- Entered using the BTEQ interface
- Entered using other interfaces that require BTEQ

Example 1: Macro Request

The following statement illustrates the use of a request terminator in the body of a macro.

```
CREATE MACRO Test_Pay (number (INTEGER),  
                      name (VARCHAR(12)),  
                      dept (INTEGER) AS  
( INSERT INTO Payroll_Test (EmpNo, Name, DeptNo)  
  VALUES (:number, :name, :dept) ;  
  UPDATE DeptCount  
  SET EmpCount = EmpCount + 1 ;  
  SELECT *  
  FROM DeptCount ; )
```

Example 2: BTEQ Request

When entered through BTEQ, the entire CREATE MACRO statement must be terminated.

```
CREATE MACRO Test_Pay
(number (INTEGER),
 name  (VARCHAR(12)),
 dept  (INTEGER) AS
(INSERT INTO Payroll_Test (EmpNo, Name, DeptNo)
VALUES (:number, :name, :dept) ;
UPDATE DeptCount
SET EmpCount = EmpCount + 1 ;
SELECT *
FROM DeptCount ; ) ;
```



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:

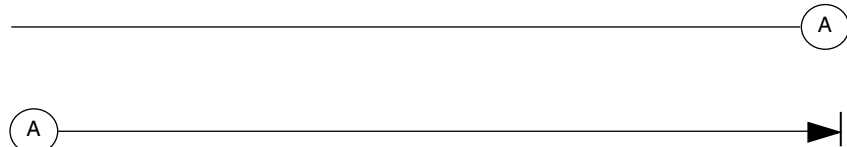
Item	Definition / Comments	
Letter	An uppercase or lowercase alphabetic character ranging from A through Z.	
Number	A digit ranging from 0 through 9. Do not use commas when entering a number with more than three digits.	
Word	Variables and reserved words.	
	IF a word is shown in ...	THEN it represents ...
	UPPERCASE LETTERS	a keyword. Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase. If a keyword is shown in uppercase, you may enter it in uppercase or mixed case.
	lowercase letters	a keyword that you must enter in lowercase, such as a UNIX command.
	lowercase italic letters	a variable such as a column or table name. You must substitute a proper value.
	lowercase bold letters	a variable that is defined immediately following the diagram that contains it.
	UNDERLINED LETTERS	the default value. This applies both to uppercase and to lowercase words.
Spaces	Use one space between items, such as keywords or variables.	
Punctuation	Enter all punctuation exactly as it appears in the diagram.	

Paths

The main path along the syntax diagram begins at the left, and proceeds, left to right, to the vertical bar, which marks the end of the diagram. Paths that do not have an arrow or a vertical bar only show portions of the syntax.

The only part of a path that reads from right to left is a loop.

Paths that are too long for one line use continuation links. Continuation links are small circles with letters indicating the beginning and end of a link:



FE0CA002

When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue.

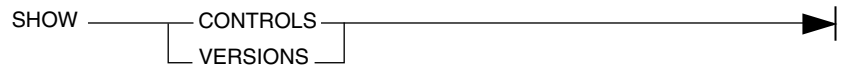
Required Items

Required items appear on the main path:



FE0CA003

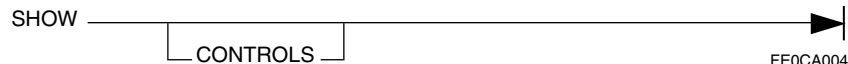
If you can choose from more than one item, the choices appear vertically, in a stack. The first item appears on the main path:



FE0CA005

Optional Items

Optional items appear below the main path:



FE0CA004

If choosing one of the items is optional, all the choices appear below the main path:



FE0CA006

You can choose one of the options, or you can disregard all of the options.

Abbreviations

If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.

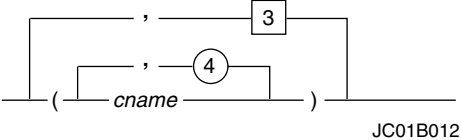


In the above syntax, the following formats are valid:

- `SHOW CONTROLS`
- `SHOW CONTROL`

Loops

A loop is an entry or a group of entries that you can repeat one or more times. Syntax diagrams show loops as a return path above the main path, over the item or items that you can repeat.



The following rules apply to loops:

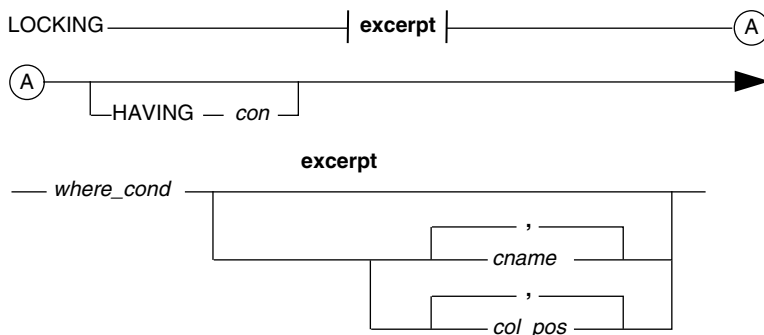
IF ...	THEN ...
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may enter <i>cname</i> a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must enter 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.



JC01A014

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.
<u>a..z</u> <u>A..Z</u> <u>0..9</u>	Unicode compatibility zone	Any fullwidth Latin letter or digit.
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.

Symbol	Encoding	Meaning
T	Any	Any multibyte character. Its encoding depends on the current character set. For KanjiEUC, code set 3 characters sometimes appear 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.
<u>Δ</u>	Any	Represents the graphic pad character.
Δ	Any	Represents either a single or multibyte pad character, depending on context.
ss ₂	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss ₃	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set

LMN<**TEST**>QRS

is represented as:

D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2

Pad Characters

The following table lists the pad characters for the various server character sets.

Server Character Set	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	SPACE	0x20
KANJI1	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
$\forall$	For all
$\exists$	There exists

Appendix B:

Restricted Words for V2R5.0

Release V2R5.0 restricts the use of certain terminology in SQL queries and in other user application programs that interface with the Teradata RDBMS.

Certain words cannot be used as object names, such as database, table, column, index, or stored procedure names, because Teradata RDBMS already uses the words and might misinterpret them. Other words are permitted as object names, but discouraged because of possible confusion that might result.

Reserved and Keywords for V2R5.0

For a complete list of Teradata RDBMS V2R5.0 reserved words, non-reserved words, words reserved for future use, and SQL-99 reserved and non-reserved keywords, see Appendix A: “Restricted Words for V2R5.0” in *Teradata RDBMS Release Summary*.

Reserved Words Script

Before upgrading from your current release of Teradata RDBMS to Release V2R5.0, run:

```
V2R5.0 Reserved Words Script
```

This script reports the names of existing data objects that conflict with Teradata RDBMS reserved words, including those newly created for the current release. Customers are advised to either rename or use double quotes around any conflicting object names.

Notes:

- This script does not check for table name aliases or embedded SQL statements.
- The script only checks for Teradata RDBMS reserved keywords and future reserved words. You must search manually for SQL-99 keywords and Teradata non-reserved keywords. For an explanation of the various classes of restricted words, see Appendix A: “Restricted Words for V2R5.0” in *Teradata RDBMS Release Summary*.

Running the Script

To get the latest list of Reserved Words and Keywords, and the Reserved Words Script to check out your system, follow these steps:

Step	Action
1	Go to http://www.info.ncr.com
2	Click Teradata Data Warehousing >Teradata Database >V2R5.0.0.
3	Click V2R5.0 Reserved Words Script (B035-1952-122K). The script opens in another browser window.
4	Copy and paste the script text into your own ASCII Text editor, and save this file on your local disk drive as a .bteq file. Note: Be sure to use an ASCII Text editor that does not add any control characters or extra carriage returns. Wordpad does not work for this purpose. Notepad and Editpad are good editors. If you use Notepad, be sure to keep the extension of the file as .bteq and not as a .txt file.
5	Run the file on your system as a BTEQ script.

Appendix C:

Teradata RDBMS Maxima

This appendix provides the following Teradata RDBMS limits.

- System maxima
- Database maxima
- Session maxima

System Maxima

The system specifications in the following table apply to an entire Teradata RDBMS configuration.

Parameter	Value
Maximum number of databases/users	4.2 x 10 ⁹
Total data capacity	43 Gbytes per AMP or optional 120 Gbytes per AMP
Maximum number of active transactions	2048
Maximum data format descriptor size	30 characters
Maximum error message text size in failure parcel	255 bytes
Maximum number of sectors per data block	255
Maximum data block size	130560 bytes
Maximum number of sessions per PE	120
Maximum number of gateways per node	1
Maximum number of sessions per Gateway	Tunable. 1200 maximum certified
Maximum number of parcels in one message	256
Maximum message size	Approximately 65000 bytes Note: This limit applies to messages to/from host systems and to some internal RDBMS messages.
Maximum number of PEs per system	1024
Maximum number of AMP and PE vprocs, in any combination, per system	16384

Database Maxima

The database specifications in the following table apply to a single database.

Parameter	Value
Maximum number of journal tables per database	1
Maximum number of data tables per database	4.2×10^9
Maximum number of columns per table or view	2048
Maximum number of columns created over the life of a table	2560
Maximum number of rows per table	Limited by disk capacity
Maximum row size	Approximately 64000 bytes
Maximum field size	Approximately 64000 bytes
Maximum database, user, table, view, macro, index, constraint or column name size	30 bytes
Maximum number of columns per index	64
Maximum SQL request size	1 Mbyte (Includes SQL statement text, USING data, and parcel overhead)
Maximum SQL response size	Approximately 64000 bytes
Maximum expanded text size for macros and views	2 Mbytes
Maximum SQL title size	60 characters
Maximum number of characters in a string constant	32000
Maximum number of columns per DML statement ORDER BY clause	16
Maximum number of primary indexes per table	1
Maximum number of partitions for a partitioned primary index	65535
Maximum number of secondary and join indexes, in any combination, per table	32
Maximum number of columns and indexes on which statistics can be recollected for a table.	512

Parameter	Value
Maximum number of parameters specified in a macro	2048
Maximum number of table-level constraints per table	100
Maximum number of referential constraints per table	64
Maximum number of tables that can be joined per query block	64
Maximum number of subquery nesting levels per query	64
Maximum number of columns in foreign and parent keys	16
Maximum number of columns referenced per single table in a hash or join index	64
Maximum number of columns referenced in the fixed part of a compressed hash or join index	64
Maximum number of columns referenced in the repeating part of a compressed hash or join index	64
Maximum number of columns in an uncompressed join index	2048
Maximum number of columns in a compressed join index	128
Maximum number of compressed values per column	255 plus nulls
Maximum number of fields in a USING row descriptor	2550

Session Maxima

The session specifications in the following table apply to a single session.

Parameter	Value
Active request result spool files	16
Parallel steps Parallel steps can be used to process a request submitted within a transaction (which may be either explicit or implicit). The maximum number of steps generated per request is determined as follows: - Per request, if no channels Note: Channels are not required for a prime key request with an equality constraint. - A request that involves redistribution of rows to other AMPs, such as a join or an INSERT-SELECT - A request that does not involve row distribution	20 steps Requires 4 channels Requires 2 channels
Maximum number of materialized global temporary tables per session	2000
Maximum number of volatile tables per session	1000

Index

Numerics

2PC, request processing 1–18

A

Account priority 1–44
ACTIVITY_COUNT 1–49
Aggregates, null and 1–39
ANSI DateTime, null and 1–37
ANSI SQL
 defined 1–2
 Teradata compliance with 1–3
 Teradata terminology and 1–6
 terminology differences 1–6
ARC
 join indexes and 2–42, 2–47
 referential integrity and 2–62
Archive and Recovery. See ARC
Arithmetic function, nulls and 1–35
Arithmetic operators, nulls and 1–35

B

BulkLoad, join indexes and 2–47

C

Call-Level Interface. See CLI
Cardinality, defined 2–6
Character names 3–6
Character set, request change of 1–45
Character sets, Teradata SQL lexicon 3–3
Child table, defined 2–57
Circular reference, referential integrity 2–60
CLI
 parcels, contents of 1–47
 session management 1–46
Column alias 1–29
Column, referencing, syntax for 1–28
Comment syntax. See Null statement
Comparison operators, null and 1–36

Constants

 character 3–30
Covering index 2–39
Covering, secondary index, non-unique, and 2–33
Cylinder reads 2–38

D

Data Control Language. See DCL
Data definition
 character data types
 Japanese GRAPHIC 3–30
 DateTime data types
 DATE literals 3–28
 TIME literals 3–28
 TIMESTAMP literals 3–28
 Japanese character graphic data types 3–30
Data Definition Language. See DDL
Data Manipulation Language. See DML
Database
 default, establishing for session 1–34
 default, establishing permanent 1–33
Database, defined 1–28
DATE literals 3–28
Date, change format of 1–44
Date, standard form of, Teradata RDBMS 1–28
DCL statements, defined 2–68
DDL statements, defined 2–4
Degree, defined 2–6
Delimiters 3–23
DML statements, defined 2–69

E

Embedded SQL
 binding style 2–2
 comments with multibyte character sets 3–38
 macros 2–86
 statement comments 3–37
Executable SQL statements 1–11
Express logon 1–45

F

Fallback, join indexes and 2-42, 2-47
 FastLoad
 join indexes and 2-42, 2-47
 referential integrity and 2-63
 Foreign key
 defined 2-18
 maintaining 2-60
 Foreign key. See also Key
 Foreign key. See also Referential integrity
 Full table scan 2-38
 cylinder reads 2-38

G

Graphic constants 3-31

H

Hexadecimal
 constants 3-32
 get representation of name 3-18

I

Index
 advantages of 2-21
 covering 2-39
 defined 2-20
 disadvantages of 2-22
 dropping 2-56
 EXPLAIN, using 2-24
 hash mapping and 2-21
 join 2-23
 keys and 2-19
 maximum number of columns C-3
 non-unique 2-23
 partitioned 2-23
 row hash value and 2-21
 RowID and 2-21
 selectivity of 2-20
 types of (Teradata) 2-22
 unique 2-22
 uniqueness value and 2-21

J

Japanese character code notation, how to read A-6
 Japanese character graphic data types 3-30
 Japanese character names 3-6
 JDBC 2-2
 Join index
 collecting statistics on 2-41
 described 2-39
 effects of 2-42, 2-47
 multitable 2-39
 performance and 2-40
 queries using 2-42
 single-table 2-39, 2-41
 Join Index. See also Index

K

KanjiEUC
 object name 3-8
 Key
 foreign 2-18
 indexes and 2-19
 primary 2-18
 referential integrity and 2-18
 Keywords 3-22, B-1
 non-reserved B-1
 reserved B-1
 SQL-99 B-1

L

Lexical separators 3-37
 Limits
 database C-3
 session C-5
 system C-2
 Literals
 DateTime data types
 DATE 3-28
 TIME 3-28
 TIMESTAMP 3-28
 Logon, express 1-45

M
Macros

- contents 2-87
- defined 2-86
- executing 2-87
- maximum expanded text size C-3
- maximum number of parameters C-4
- SQL statements and 2-86

Maxima

- database C-3
- session C-5
- system C-2

Message, components of 1-48**Multibyte character sets, comments 3-37****MultiLoad**

- join indexes and 2-42, 2-47
- referential integrity and 2-63

Multi-statement requests, performance 1-19**Multi-statement transactions 1-20****Multitable join index 2-39**

N
Name

- calculate length of 3-10
- fully qualified 1-29
- get hexadecimal representation 3-18
- identify in logon string 3-20
- maximum size C-3
- multiword 3-4
- object 3-6
 - KanjiEUC 3-8
 - KanjiShift-JIS 3-9
 - translation and storage 3-14
- resolving 1-32

Nonexecutable SQL statements 1-12**Non-partitioned primary index. See NPPI.****Non-reserved keywords B-1****Non-unique index. See Index, Primary index, Secondary index****NPPI**

- defined 2-23

Null

- aggregates and 1-39
- ANSI DateTime and 1-37
- arithmetic functions and 1-35
- arithmetic operators and 1-35
- collation sequence 1-38
- comparison operators and 1-36
- excluding 1-36
- searching for 1-36
- searching for, null and non-null 1-37

Null statement

- bracketed form, defined 1-15
- simple form, defined 1-15

NUPI. See Primary index, non-unique**NUSI. See Secondary index, non-unique**

O
Object names 3-6**Object, name comparison 3-15****ODBC 2-2****Operators 3-35**

P
Parallel step processing 1-20**Parameters, session 1-41****Parent table, defined 2-58****Partial cover 2-40****Partition elimination 2-80****Partitioned primary index. See PPI.****Permanent journal, join indexes and 2-42, 2-47****PP2, declaratives 3-38****PPI**

- defined 2-23
- maximum number of partitions C-3
- partition elimination and 2-80

Precedence

- SQL operators 3-36

Primary index

- choosing 2-27
- default 2-26
- described 2-26
- non-unique 2-27
- summary 2-29
- unique 2-27

Primary key, defined 2-18**Primary key. See also Key****Procedure, dropping 2-56****Profiles**

- defined 2-64

Punctuation, SQL syntax statement 1-9

Q

Query processing
 access types 2-82
 all AMP request 2-76
 AMP sort 2-78
 BYNET merge 2-79
 defined 2-73
 full table scan 2-81
 single AMP request 2-74
 single AMP response 2-76
 Query, defined 2-73

R

Referential integrity
 ARC and 2-62
 circular references and 2-60
 described 2-57
 FastLoad and 2-63
 foreign keys and 2-59
 importance of 2-59
 MultiLoad and 2-63
 terminology 2-58
 Request processing
 2PC 1-18
 ANSI mode 1-17
 Teradata mode 1-18
 Request terminator 3-41
 Requests
 maximum size C-3
 multi-statement 1-13
 single-statement 1-13
 Requests. See also Blocked requests, Multi-statement requests, Request processing
 Reserved keywords B-1
 Reserved words script B-1
 Roles
 defined 2-66
 Rows
 maximum size C-3

S

Secondary index
 defined 2-31
 dual 2-33
 non-unique 2-31
 bit mapping 2-34
 covering and 2-33
 value-ordered 2-33
 summary 2-36
 unique 2-31
 Security
 user-level password attributes 2-64
 Selectivity
 high 2-20
 low 2-20
 Semicolon
 request terminator 3-41
 statement separator 3-39
 Separator
 lexical 3-37
 statement 3-39
 Session collation 1-43
 Session control 1-41
 Session handling, session control 1-46
 Session management
 CLI 1-46
 ODBC 1-46
 packets 1-48
 parcels 1-47
 requests 1-47
 session reserve 1-46
 Session parameters 1-41
 Single-table join index 2-39
 Specifications
 database C-3
 session C-5
 system C-2
 SQL
 dynamic 1-22
 dynamic, SELECT statement and 1-26
 static 1-22
 SQL binding styles
 CLI 2-2
 defined 2-2
 direct 2-2
 embedded 2-2
 JDBC 2-2
 ODBC 2-2
 stored procedure 2-2
 SQL error response (ANSI) 1-56

SQL Flagger 1–8
 session control 1–42
 SQL functional families, defined 2–2
 SQL lexicon
 character names 3–6
 delimiters 3–23
 graphic constants 3–31
 hexadecimal constants 3–32
 Japanese character names 3–3, 3–6
 keywords 3–22
 lexical separators 3–37
 object names 3–6
 operators 3–35
 request terminator 3–41
 statement separator 3–39
 SQL requests
 multi-statement 1–13
 single-statement 1–13
 SQL responses 1–52
 failure 1–55
 success 1–53
 warning 1–54
 SQL return codes 1–49
 SQL statements
 basic structure 1–9
 executable 1–11
 invoking 1–11
 name resolution 1–32
 nonexecutable 1–12
 partial names, use of 1–31
 punctuation 1–9
 SELECT, dynamic SQL 1–26
 structure 1–30
 SQL statements, macros and 2–86
 SQL. See also Embedded SQL
 SQL-99 keywords B–1
 SQLCA 1–49
 SQLCODE 1–49
 SQLSTATE 1–49
 Statement processing. See Query processing
 Statement separator 3–39
 Stored procedures
 ACTIVITY_COUNT 1–49
 creating 2–93
 deleting 2–96
 elements of 2–91
 executing 2–95
 modifying 2–94
 privileges 2–92
 renaming 2–96
 Subquery, defined 2–84
 Syntax, how to read A–1

T

Table
 cardinality of 2–6
 creating indexes for 2–6
 defined 2–6
 degree of 2–6
 dropping 2–56
 full table scan 2–38
 global temporary 2–9
 maximum number of columns C–3
 maximum number of rows C–3
 tuple and 2–6
 volatile temporary 2–15
 Table access summary 2–48
 Table structure, altering 2–54
 Table, change structure of 2–54
 Teradata DBS, session management 1–46
 Teradata Index Wizard 2–24
 Teradata RDBMS
 database specifications C–3
 session specifications C–5
 system specifications C–2
 Teradata SQL, ANSI SQL and 1–3
 Terminator, request 3–41
 TIME literals 3–28
 TIMESTAMP literals 3–28
 TITLE phrase, column definition 3–5
 TPump
 join indexes and 2–42
 TPump, join indexes and 2–47
 Transaction mode, session control 1–42
 Transactions
 defined 1–16
 explicit, defined 1–18
 implicit, defined 1–18
 Trigger
 defined 2–52
 dropping 2–56
 process flow for 2–52
 Two-phase commit. See 2PC

U

Unicode, notation A–6
 Unique index. See Index, Primary index, Secondary index
 UPI. See Primary index, unique
 USI. See Secondary index, unique

V

View

- described 2-50
- dropping 2-56
- maximum expanded text size C-3
- maximum number of columns C-3
- restrictions 2-51