Teradata
a division of NCR

Teradata® RDBMS

SQL Reference - Volume 2
Statement and Transaction Processing

V2R5.0

B035-1101-122A
December 2002

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

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

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

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

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

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

or write:

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

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

Copyright © 1996-2002, NCR Corporation
All Rights Reserved

Preface

Supported Software Release

This book supports Teradata RDBMS V2R5.0.

Changes to This Book

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

Date	Description
December 2002	• Upgraded parser diagram to indicate security module. • Added new material on revised QCD to chapter 4 to comply with enhanced Query Capture Facility features. • Added new chapter 5 on Teradata Index Wizard feature. • Added material to chapter 2 about when and why to collect statistics. • Added material to chapter 2 about random table statistical samples, comparing them with random AMP statistical samples and full table scan statistics in terms of relative likelihood of their accuracy. • Moved Target Level Emulation DIAGNOSTIC statements to *Teradata RDBMS SQL Reference, Volume 4*. • Performed DR fixes and enhancement requests.
June 2001	• Moved some material on query optimization from Chapter 1 to Chapter 2. • Added new material on query optimization and join planning to Chapter 2. • Revised material in Chapter 4 to comply with enhanced Query Capture Facility features. • Performed DR fixes and enhancement requests.
September 2000	• Usability enhancements.
June 2000	• General reorganization and rewriting of material to enhance usability. • Addition of new material on the Query Capture Facility. • Addition of new material on Target Level Emulation feature. • Addition and extensive rewriting and reorganization of embedded SQL material formerly found in Teradata RDBMS Application Programming with C, COBOL and PL/I.

About This Book

Purpose

Teradata RDBMS SQL Reference, Volume 2 describes the Teradata SQL parser, including its component parts, the Query Capture Database, the database components of the Teradata Index Wizard and related utilities, and the basics of the Teradata SQL transaction processing environment.

This preface describes the organization of *Teradata RDBMS SQL Reference, Volume 2* 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

This book is intended for database administrators, SQL programmers, and other users who interface with the Teradata RDBMS.

How This Book Is Organized

This manual has seven chapters and one appendix.

- Chapter 1: "Statement Parsing," describes SQL statement parsing, including the components of the SQL parser.
- Chapter 2: "Query Optimization," describes query optimization.
- Chapter 3: "Join Optimizations," describes join optimization processes, including star and snowflake joins.
- Chapter 4: "Query Capture Facility," describes a feature used to capture the Optimizer query plans expressed by EXPLAIN statements for additional offline analysis.
- Chapter 5: "Database Foundations for the Teradata Index Wizard," describes the server functionality underlying the Teradata Index Wizard client utility.
- Chapter 6: "Target Level Emulation," describes a feature used to capture the environmental cost factors from a production system. The captured data can then be exported to a test system and used to examine the query plans the test system Optimizer generated as if they had been processed on the target system.
- Chapter 7: "Transaction Processing," describes SQL transaction semantics, processing, and database object locking.
- Appendix A: "Notation Conventions," explains how to read the syntax diagrams, character shorthand notation, and predicate calculus notation used in this book.

Prerequisites

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

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

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

List of Acronyms

This book uses acronyms, which the following table lists in alphabetical order:

ACID	Atomicity, Consistency, Isolation, Durability
AMP	Access Module Processor vproc
ANSI	American National Standards Institute
API	Application Programming Interface
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
OLTP	On-Line Transaction Processing
PDE	Parallel Database Extensions
PE	Parsing Engine vproc
PI	Primary Index
PK	Primary Key
PPI	Partitioned Primary Index
QCD	Query Capture Database

QCF	Query Capture Facility
RDBMS	Relational Database Management System
RI	Referential Integrity
SQL	Structured Query Language
TDQM	Teradata Query Manager
TLE	Target Level Emulation
TPA	Trusted Parallel Application
TSET	Teradata System Emulation Tool
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:
Statement Parsing

The SQL Parser 1–2
Dictionary Cache 1–6
Syntaxer 1–8
Resolver 1–11
Security Checking 1–13
Optimizer 1–14
Generator 1–21
Statement Cache 1–23
Purging the Statement Cache 1–27
Apply 1–29
Dispatcher 1–31

Chapter 2:
Query Optimization

Query Optimizers 2–2
Teradata Optimizer Processes 2–6
Translation to Internal Representation 2–8
Query Rewrite 2–14
Why Are Statistics Necessary? 2–20
Interval Histograms 2–25
All-AMPs Sampled Statistics 2–33
Random AMP Sampling 2–35
Relative Accuracy of Residual Statistics Versus Random AMP Sample Statistics 2–37
Comparing the Relative Accuracies of Various Methods of Collecting Statistics 2–40
When Should Statistics Be Collected Or Recollected? 2–42

How the Optimizer Uses Statistical Profiles ... 2–44
Examples of How Cardinality Estimates Are Made for Simple Queries Using Interval Histograms ... 2–46
Cost Optimization ... 2–56
Environmental Cost Factors ... 2–62
Optimizer Join Plans ... 2–64
Evaluating Join Orders ... 2–72
Lookahead Join Planning ... 2–76
Join Methods ... 2–80
Product Join ... 2–82
Merge Join ... 2–86
Hash Join ... 2–93
Nested Join ... 2–97
Local Nested Join ... 2–98
Slow Path Local Nested Join ... 2–100
Fast Path Local Nested Join ... 2–103
Remote Nested Join ... 2–104
Nested Join Examples ... 2–107
Join Plan Without Nested Join ... 2–108
Join Plan With Nested Join ... 2–109
Exclusion Join ... 2–110
Exclusion Merge Join ... 2–111
Exclusion Product Join ... 2–114
Inclusion Join ... 2–116
RowID Join ... 2–117
Correlated Joins ... 2–119
Minus All Join ... 2–121
Relational Query Optimization References ... 2–122

Chapter 3:
Join Optimizations

Star and Snowflake Join Optimization ... 3–3
LT/ST-J1 Indexed Joins ... 3–8
LT-ST-J2 Unindexed Joins ... 3–9
Miscellaneous Considerations for Star Join Optimization ... 3–10
Selecting Indexes for Star Joins ... 3–12
Star Join Examples ... 3–15
Cardinality and Uniqueness Statistics for the Reasonable Indexed Join Examples ... 3–17
Reasonable Indexed Join Plan Without Star Join Optimization ... 3–18
Reasonable Indexed Join Plan With Star Join Optimization: Large Table Primary Index Joined to Small Tables ... 3–20

Reasonable Indexed Join Plan With Star Join Optimization: Large Table USI Joined to Small Tables ... 3–22
Reasonable Indexed Join Plan With Star Join Optimization: Large Table NUSI Joined to Small Tables ... 3–24
Join Plan With Star Join Optimization: Large Table Subquery Join ... 3–26
Cardinality and Uniqueness Statistics for the Reasonable Unindexed Join Examples ... 3–28
Reasonable Unindexed Join Without Join Optimization ... 3–29
Reasonable Unindexed Join With Join Optimization ... 3–31
Join Indexes ... 3–33
Maintenance of Join Index for DELETE, INSERT, and UPDATE ... 3–39
General Method of Maintaining Join Index During DELETE ... 3–41
Optimized Method of Maintaining Join Index During DELETE ... 3–43
Maintaining Join Index During INSERT ... 3–44
General Method of Maintaining Join Index During UPDATE ... 3–46
Optimized Method of Maintaining Join Index During UPDATE ... 3–48

Chapter 4:
Query Capture Facility

Compatibility Issues With Prior Teradata Releases ... 4–2
Functional Overview of the Query Capture Facility ... 4–3
Query Capture Database ... 4–5
Creating the QCD Tables ... 4–7
Dropping the QCD Tables ... 4–11
Querying QCD ... 4–13
QCD Query Macros and Views ... 4–14
QCD Table Definitions ... 4–17
AnalysisLog ... 4–18
DataDemographics ... 4–20
Field ... 4–22
Index_Field ... 4–25
IndexColumns ... 4–26
IndexRecommendations ... 4–27
IndexTable ... 4–31
Predicate ... 4–35
Predicate_Field ... 4–37
QryRelX ... 4–38
Query ... 4–40
QuerySteps ... 4–44
Relation ... 4–51
SeqNumber ... 4–57
TableStatistics ... 4–59

User_Database 4–62
UserRemarks 4–63
ViewTable 4–65
Workload 4–67
WorkloadQueries 4–68
WorkloadStatus 4–69

Chapter 5:
Database Foundations for the Teradata Index Wizard
Teradata Index Wizard Overview 5–3
Workload Identification 5–5
Workload Definition 5–8
Index Analysis 5–11
Index Validation 5–16
Index Application 5–18

Chapter 6:
Target Level Emulation
An Overview of Target Level Emulation 6–2
Procedure to Enable Target Level Emulation 6–6
Procedures to Capture Target System Optimizer Cost Values and Random AMP Statistical Samples for a Test Environment 6–7

Chapter 7:
Transaction Processing
Database Transactions 7–2
Locking Levels and Severities 7–3
Lock Manager 7–6
Default Lock Assignments and Lock Upgradeability 7–8
Example: No Deadlock 7–12
Example: Deadlock 7–13
Example: Two Serial Transactions 7–15
DDL and DCL Statements, Data Dictionary Access, and Locks 7–17
DML Statements and Locks 7–18
Cursor Locking Modes 7–19
Blocked Requests 7–21
Deadlock Detection and Resolution 7–24
Preventing Deadlocks 7–26
Locking and Transaction Processing 7–32
Upgrading System Locks 7–33
Transaction Semantics: Operating in ANSI or Teradata Transaction Modes 7–34

ANSI Session Mode 7–37
Teradata Session Mode 7–38
Comparison of Transaction Rules in ANSI and Teradata Transaction Modes 7–39
Rollback Processing 7–41
Locking Issues With Tactical Queries 7–43

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

Index Index–1

Chapter 1:

Statement Parsing

This chapter describes SQL statement parsing, including the components of the SQL Parser that deal with query processing.

Topics include:

- Data Dictionary Cache
- Syntaxer
- Resolver
- Security
- Optimizer
- Generator
- Statement Cache
- Purging the Statement Cache
- Apply
- Dispatcher

The SQL Parser

Introduction

The SQL Parser is a component of the parsing engine (PE).

SQL requests are sent to the Parser in CLIv2 Request parcels. Request parcels consist of the following elements.

- One or more SQL statements
- Control information
- Optional USING data

Major Components of the Parsing Engine

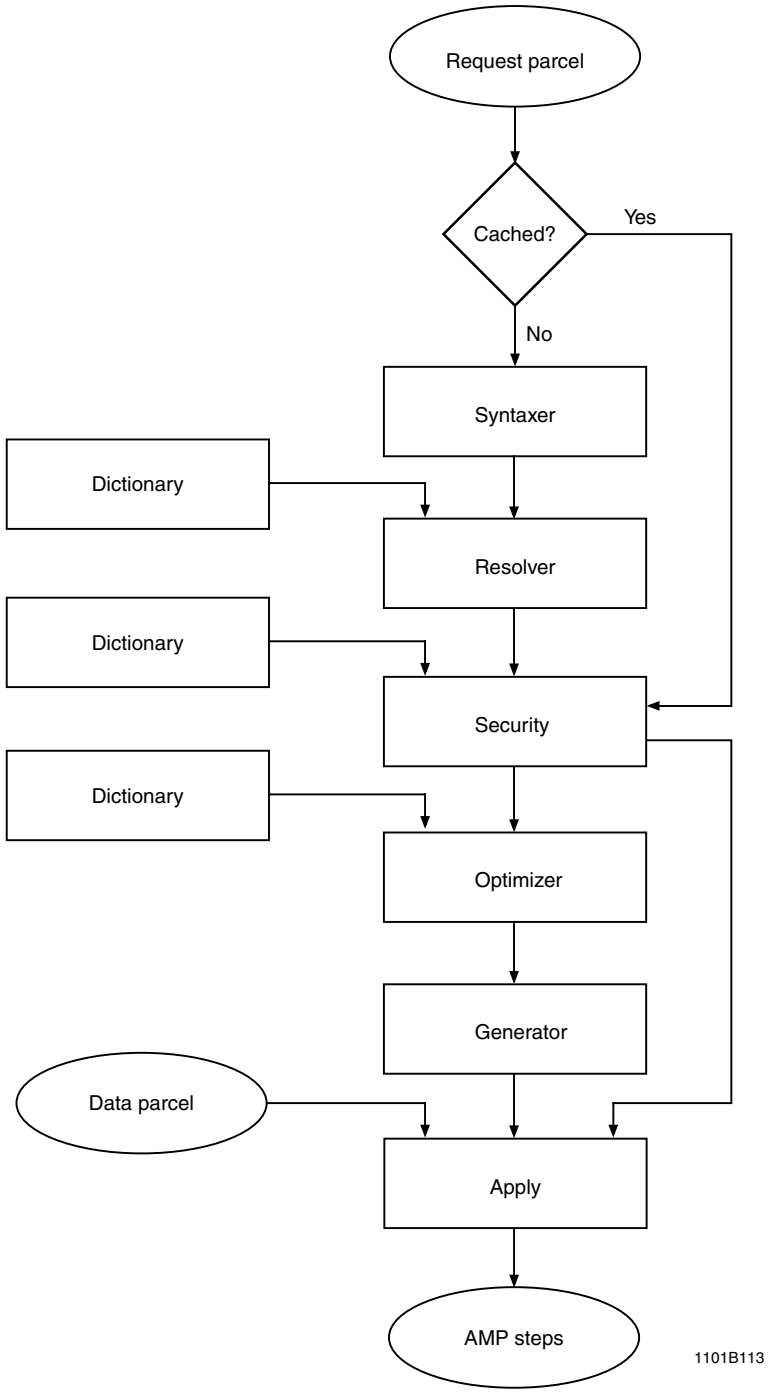
The parsing engine has four major and several lesser components related to query processing:

- The Parser
  - Syntaxer
  - Resolver
  - Security Checking
- The Optimizer
  - Query rewrite
  - Access planning
  - Join planning
  - Index Wizard
- The Generator
  - Steps generation
  - Steps packaging
- The Dispatcher
  - Execution control
  - Response control
  - Transaction and request abort management

The Parser also manages sessions between client applications and the Teradata server.

Block Diagram of Parsing Engine Activity

The parsing engine generates AMP steps from Request parcels, as shown in the following illustration:



The diagram does not include the Teradata Index Wizard because it is not used to process queries in a production environment. For more information about the Teradata Index Wizard, see Chapter 5: "Database Foundations for the Teradata Index Wizard."

Parsing Engine Component Processes

Parsing engine components perform the following functions for each SQL request sent to the Teradata server from a client application:

Stage	Process
1	The Syntaxer analyzes the high-level syntax of the statement for errors. If the syntax passes the check, then the statement components are converted into a data structure called a parse tree. This structure is an exact mapping of the original query text (see "Parse Tree Representations of a Query" on page 2-8). This skeletal parse tree is called a SynTree, which the Syntaxer then passes on to the Resolver.
2	The Resolver takes the SynTree and fleshes it out with information about any required data conversions and security checks, adds column names and notes any underlying relationships with other database objects, and then passes the more fleshed out tree, now known as a ResTree, to the Optimizer.
3	The Optimizer analyzes the ResTree to see if it can factor and combine the SQL statement components into a simpler, faster-performing form. It does this using various database statistics and configuration data to determine the optimum plans to access and join the tables specified by the request. The Optimizer then examines any locks placed by the SQL statement and attempts to optimize their placement to enhance performance and avoid deadlocks. The optimized parse tree, now transformed from a simple statement tree to a complex operation tree, is then passed to the Steps Generator for further processing.
4	The Steps Generator creates plastic steps. Plastic steps are, except for statement literals, a data-free skeletal tree of AMP directives derived from the optimized parse tree. The completed plastic steps tree is then passed to Steps Packaging for further processing.

Stage	Process
5	Steps Packaging adds context to the plastic steps by integrating various user- and session-specific information. If any data parcels were passed to the Parser via a USING modifier, then that data is also added to the steps tree. The final product of this process is referred to as concrete steps.
6	Steps Packaging passes the concrete steps to the Dispatcher for assignment.
7	The Dispatcher sequentially, incrementally, and atomically transmits the concrete steps, called AMP steps at this point in the process, across the BYNET to the appropriate AMPs for processing.
8	The Dispatcher manages any abort processing that might be required.
9	The Dispatcher receives the results of the AMP steps from the BYNET and returns them to the requesting application.

Dictionary Cache

Introduction

To transform a Teradata SQL statement into the steps needed to process the query, the Parser needs current information from the data dictionary about tables, columns, views, macros, triggers, stored procedures, and other objects.

Definition: Dictionary Cache

The dictionary cache is a buffer in parsing engine memory that stores the most recently used dictionary information. These entries, which also contain statistical information used by the Optimizer (see "Interval Histograms" on page 2-25 and "Environmental Cost Factors" on page 2-62), are used to convert database object names to their numeric IDs.

Why Cache?

Caching the information reduces the I/O activity for the following items:

- Resolving database object names
- Optimizing access paths
- Validating access rights

When the Parser needs definitions not found in cache, it asks an AMP to retrieve the necessary information. When the information is received, the Parser stores it in the dictionary cache. If another SQL statement requires information about the same database object, the Parser retrieves it from cache rather than performing the more costly task of asking the AMP multiple times for the same information.

Keeping the Cache Contents Fresh

If an SQL statement changes the contents of the data dictionary, a spoil message is sent to every PE, instructing them to drop the changed definitions from their respective dictionary caches.

The dictionary cache is purged periodically, phased so that the cache for only one PE is purged at a time.

Use DBSControl to Fine Tune Cache

For each PE, the default size of the dictionary cache is 1 Mbyte. You can also control how many of its most recent 550 object references are retained in cache.

See "DBS Control Utility" in *Teradata RDBMS Utilities* for further information about fine tuning the dictionary cache.

Syntaxer

Introduction

The Syntaxer checks the request parcel for high level syntax. If no errors are detected, it converts the request parcels into a skeletal parse tree referred to as the SynTree, which it then passes on to the Resolver.

A parse tree is a data structure used by the Teradata SQL Parser to represent a Request parcel in a form that is simple to annotate with various descriptive and statistical information derived from the data dictionary. The Syntaxer does not transform the query text in any way.

The parse tree also permits a relatively simple transformation of the Request parcel into an execution plan.

Syntaxer Processing Rule

The larger the Request parcel, the longer the syntaxer takes to generate the parse tree.

A corollary to this rule is that using macros has a positive effect in reducing the time to generate a parse tree.

Views and macros can be nested up to 64 levels deep, but nesting adds processing time.

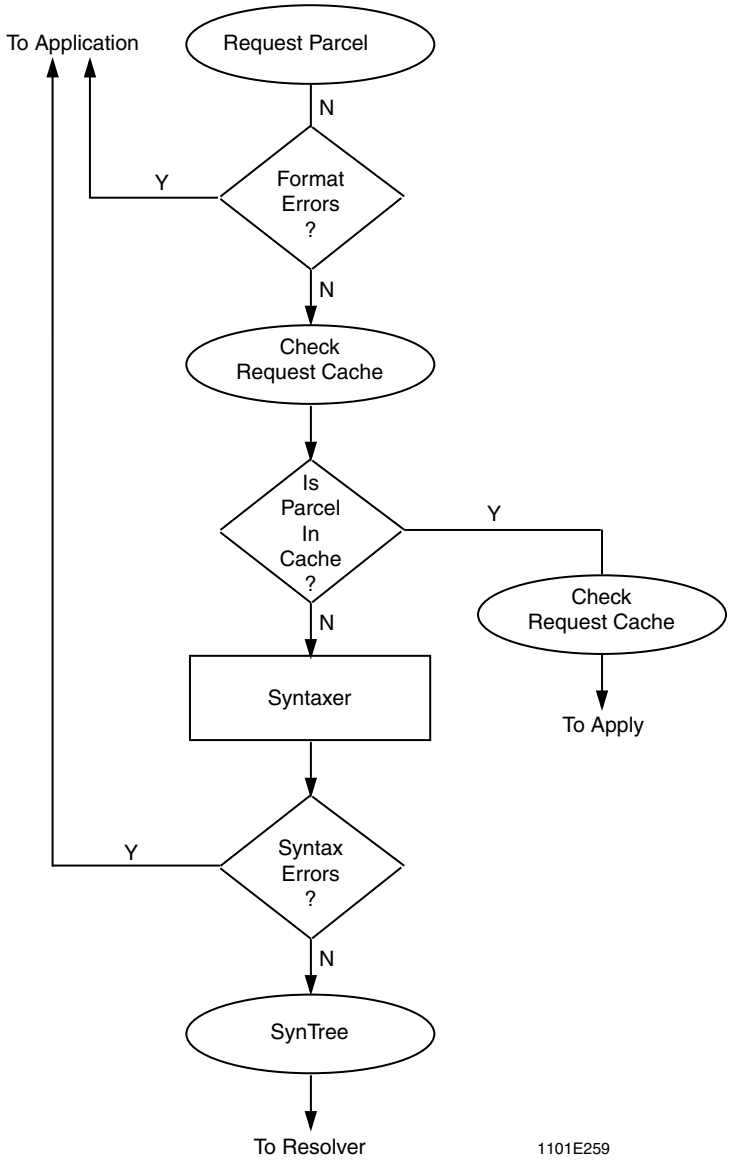
Syntaxer Components

The Syntaxer is composed of the Lexer and Synact.

The Lexer decomposes a Request parcel into its component tokens such as keywords, special characters, numeric data, and character strings.

Block Diagram of Syntaxer Activity

A block diagram of Syntaxer activity is shown in the following illustration:



Syntaxer Component Processes

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>The Syntaxer checks the request cache to determine if it contains an identical Request parcel.
<table>
<tr><th>IF an identical parcel is . . .</th><th>THEN the Syntaxer . . .</th></tr>
<tr><td>found</td><td>calls Steps Packaging and passes it the previously generated AMP processing steps. The steps, called plastic steps, are not yet bound with host variable data values.

Plastic steps are directives to the AMPs. The steps do not yet contain data values from the data parcel (from the USING modifier).

Plastic steps are flexible, thereby allowing different sets of values (from the USING clause) to be inserted during subsequent operations.</td></tr>
<tr><td>not found</td><td>produces an initial parse tree, then calls the Resolver.</td></tr>
</table></td></tr>
<tr><td>2</td><td>If no matching parcel is found in cache and if no syntax errors are detected, then the Syntaxer generates a skeletal parse tree called a SynTree and passes it to the Resolver.

If a syntax error is detected, the Request parcel is returned to the requesting application with an appropriate error message.

At later points in the parsing process, additional information are added to the parse tree, building toward an eventual set of concrete AMP steps.</td></tr>
</table>

SQL Flagger Activity

When the SQL Flagger is enabled, the Syntaxer is the component of the Parser that does most of the flagging.

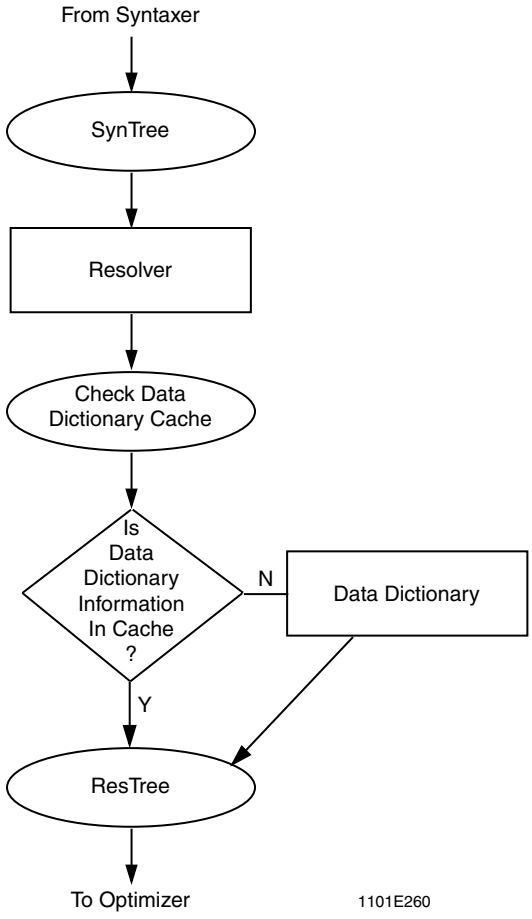
Resolver

Introduction

The Resolver annotates the SynTree with information about such things as data conversions, column names, security checks, and underlying relationships and then produces a more fleshed out parse tree called a ResTree, which it passes to the Optimizer. The Resolver does not transform the query text in any way.

Block Diagram of Resolver Activity

A block diagram of Resolver activity is shown in the following illustration:



Resolver Component Processes

Stage	Process
1	Each database or user, table, view, trigger, stored procedure, and macro is assigned a globally unique numeric ID. Each column and each index is assigned a numeric ID that is unique within its table. These IDs are maintained in the data dictionary.
2	The Resolver refers to the data dictionary to verify all names and access rights and to convert those names to their equivalent numeric IDs.
3	The Resolver takes available information from the data dictionary cache, which is used on a least-recently-used or most-recently-used basis. If the needed information is not cached, it is retrieved from the appropriate system tables.
4	If a Request parcel contains views or macros, then the Resolver retrieves the view or macro text from the data dictionary, resolves it, and then merges the elements of the resulting tree into the request tree.

SQL Flagger Activity

Some additional compliance checking is performed by the Resolver. The most significant of these compliance checks is a semantic check that detects instances in which data types are compared or converted and assigned implicitly.

Security Checking

After the request parcel passes its Resolver checks, the Parser interrogates several system tables in the data dictionary to ensure that the user making an SQL request has all the appropriate logon and database object access rights.

Because security privileges can be granted and revoked dynamically, Teradata validates user logon security rights for every request processed, including those that have been cached. Database object access rights for cached requests are not revalidated when requests are processed from cache.

For more information about Teradata security, refer to the following manuals:

- *Teradata RDBMS Security Administration*
- *Teradata RDBMS Database Administration*
- *Teradata RDBMS Data Dictionary*

Optimizer

Introduction

The Teradata SQL Query Optimizer determines the most efficient way to access and join the tables required to answer an SQL request.

For more detailed information about optimization, see the following topics in Chapter 2: "Query Optimization."

- "Query Optimizers" on page 2-2
- "Translation to Internal Representation" on page 2-8
- "Query Rewrite" on page 2-14
- "Interval Histograms" on page 2-25
- "How the Optimizer Uses Statistical Profiles" on page 2-44
- "Examples of How Cardinality Estimates Are Made for Simple Queries Using Interval Histograms" on page 2-46
- "Cost Optimization" on page 2-56
- "Environmental Cost Factors" on page 2-62
- "Optimizer Join Plans" on page 2-64
- "Evaluating Join Orders" on page 2-72

For information about the Teradata Index Wizard, which is also a component of the Optimizer, see Chapter 1: "Database Query Analysis."

What Sorts of Questions Does a Query Optimizer Ask?

The Optimizer performs its task of determining a best plan for joining and accessing tables using various demographic information about the tables and columns involved in the request and the configuration of the system, as well as numerous heuristic strategies, or rules of thumb.

Among the myriad possible optimization aids examined by the Optimizer are those addressed by the following questions.

- How can the query text be rewritten to make it more efficient without changing its semantics?
- What is the cardinality of the table?

  In this context, cardinality generally refers to the number of rows in a result or spool table, not the number of rows in a base table.
- What is the degree of the table?

  In this context, degree generally refers to the number of columns in a result or spool table, not the number of columns in a base table.
- Is the requested column set indexed?
- If the column set is indexed, is the index unique or nonunique?
- How many distinct values are in the column set?
- How many rows per column set value are expected to be returned?
- Can a base table be replaced by a hash or single-table join index?
- Is a join partly or completely covered by a join index or NUSI?
- Is an aggregate already calculated for a column by an existing join index?
- What strategies have tended to work best with queries of this type in the past?
- How many AMPs are there in the system?
- How many nodes are there in the system?
- How much and what kind of disk does each AMP have and what is the processor speed of the node it is running on?

How Does a Query Optimizer Answer the Questions It Asks?

Ideally, these questions are answered largely based on statistical data that you have generated using the SQL COLLECT STATISTICS statement. When database statistics are collected regularly, then you can always expect the Optimizer to make the best decisions possible.

If statistics have not been collected on a regular basis, then the Optimizer might not be able to make the best-informed decisions about how to proceed.

If no statistics have been collected, then the Optimizer makes a random snapshot sampling of data from a single AMP and uses that estimate to make a best guess about the optimum data retrieval path.

The degree that this random AMP sample approximates the population demographics for a column or table is directly proportional to the size of the table: the larger the table, the more likely a one-AMP sample approximates its true global demographics.

What Query Optimizers Do Not Do

Query optimizers do not do either of the following things:

- Guarantee that the access and join plans it generates are infallibly the best plans possible.

  A query optimizer always generates several optimal plans based on the population and environmental demographics it has to work with and the quality of code for the query it receives, then selects the best of the generated plan set to use to respond to the DML statement.

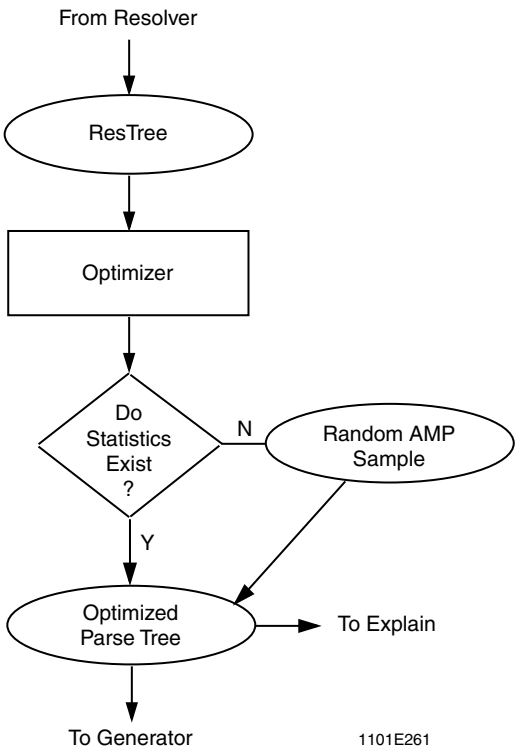
  You should not assume that any query optimizer ever produces the best query plan possible to support a given DML statement.

  You *should* assume that the query plan selected for use is more optimal than the otherwise unoptimized Resolver tree formulation would have been.
- Rationalize poorly formed queries in such a way as to make their performance as effective as a semantically equivalent well-formed query that returns the same result.

  A query optimizer always creates the most effective plan it can *for the query it is presented*; nevertheless, semantically identical queries can differ in their execution times by an order of magnitude or more depending on how carefully their SQL code is written.

Block Diagram of Query Optimizer Processes

A block diagram of Optimizer activity is shown in the following illustration and explained in the table that follows.



Optimizer Component Processes

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>The Optimizer examines an incoming Request parcel to determine if it is a DML statement or a DDL/DCL statement.
<table>
<tr><th>IF the Request parcel contains this type of SQL statement ...</th><th>THEN the Optimizer ...</th></tr>
<tr><td>DDL or DCL</td><td>deletes the original statement from the parse tree after it has been replaced with specific data dictionary operations.

No access planning is required for DDL and DCL statements, so the Optimizer only converts the request parcel into work steps involving dictionary writes, locking information, and so on.</td></tr>
<tr><td>DML</td><td>produces access paths and execution plans.

The Optimizer then uses whatever statistical information it has to determine which access paths or plans are to be used.

If no column or index statistics exist in the data dictionary, then the Optimizer uses dynamic AMP sampling to estimate the population statistics of the data.</td></tr>
</table></td></tr>
<tr><td>2</td><td>The Optimizer determines if the steps are to be executed in series or in parallel, and if they are to be individual or common processing steps.</td></tr>
<tr><td>3</td><td>The parse tree is further fleshed out with the optimized access paths and join plans and the Optimizer selects the best access path based on the table statistics it has to work with.</td></tr>
<tr><td>4</td><td>The Optimizer places, combines, and reorders locks to reduce the likelihood of deadlocks and then removes any duplicate locks it finds.</td></tr>
<tr><td>5</td><td>Finally, the Optimizer passes its fully optimized parse tree on to the Generator for further processing.</td></tr>
</table>

Definition: Steps

A step is a Teradata data structure that describes an operation to be processed by one or more AMPs in order to perform a task in response to a request parcel.

There are several types of steps, the most important of which are the plastic and concrete steps.

More information about plastic steps is in "Generator" on page 1-21.

More information about concrete steps is in "Apply" on page 1-29.

Definition: Parallel Steps

Parallel steps are steps from the same request parcel that can be processed concurrently by the AMPs or a single step in a request parcel that can be processed simultaneously by multiple AMPs, taking advantage of the parallelism inherent in the Teradata architecture. Each parallel step has an independent execution path, running simultaneously with other steps, but on different AMPs.

The Optimizer determines which steps of a task can be run in parallel and groups them together. These parallel steps, which make the best use of the BYNET architecture, are generated by the Optimizer whenever possible.

The EXPLAIN facility explicitly reports any parallel steps specified by the Optimizer.

Definition: Common Steps

Common steps are processing steps common to two or more SQL statements from the same request parcel or macro. They are recognized as such and combined by the Optimizer.

For example, consider the following multistatement request parcel.

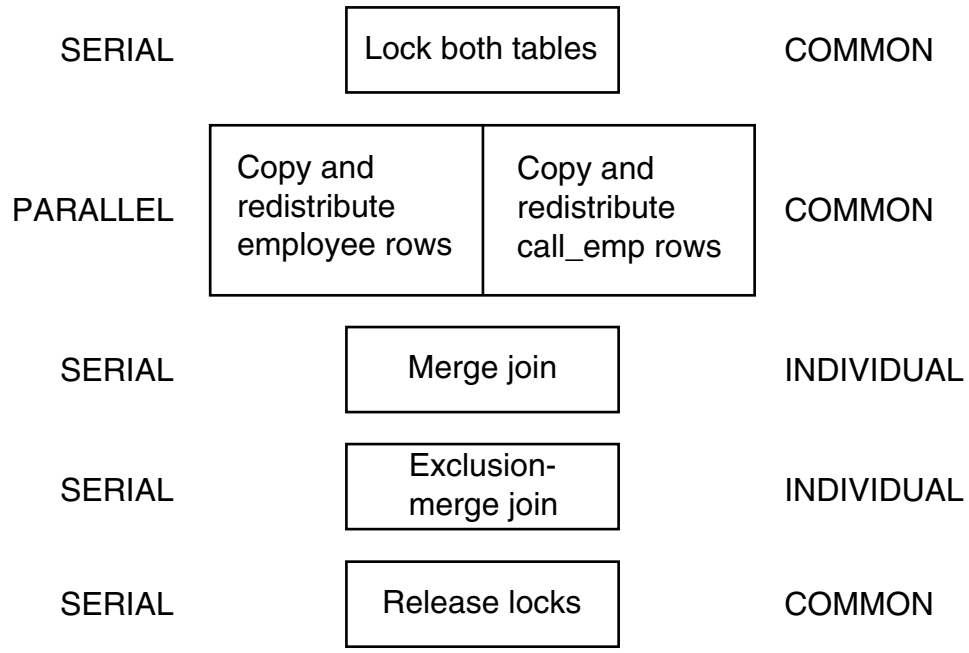
```
SELECT employee_number, last_name, 'Handling Calls'
FROM employee
WHERE employee_number IN
 (SELECT employee_number
  FROM call_employee)

;SELECT employee_number, last_name, 'Not Handling Calls'
FROM employee
WHERE employee_number NOT IN
 (SELECT employee_number
  FROM call_employee);
```

The Optimizer processes these requests in parallel using a common steps approach as illustrated by the following table:

Stage	Process	Processing Mode	Step Type
1	Both tables (employee_number and call_emp) are locked.	Serial	Common
2	The rows from the employee_number table are copied and redistributed. The rows from the call_emp table are copied and redistributed.	Parallel	Common
3	Results are merge joined.	Serial	Individual
4	Results are exclusion merge joined.	Serial	Individual
5	The table locks are released.	Serial	Common

The Optimizer generates the parallel and common steps for the parcel as shown in the following illustration:



FF03A008

Generator

Introduction

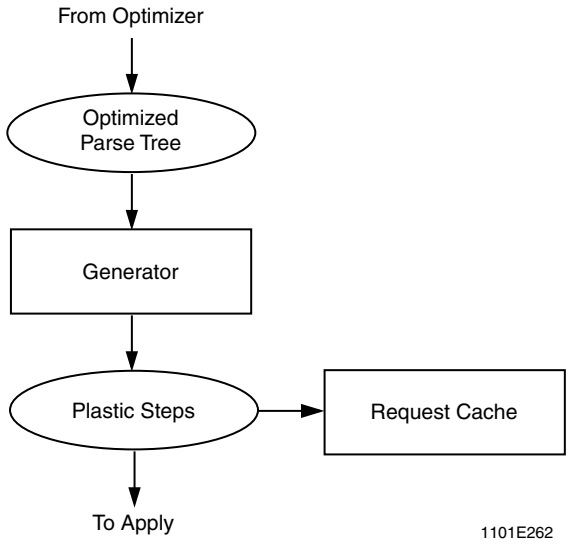
The Generator formulates the AMP processing steps, called plastic steps, based on the optimized parse tree plan it receives as the output of the Optimizer.

Except for any hard-coded literal values that may be used, the plastic steps do not have any data values associated with them.

A single request parcel can generate many plastic steps.

Block Diagram of Generator Activity

The following block diagram illustrates the workings of the Generator. The Generator is also the home of the Teradata hashing algorithm.



Generator Component Processes

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>The Generator receives the optimized parse tree from the Optimizer and uses it to build plastic steps.</td></tr>
<tr><td>2</td><td>Each step is created with a step header containing fixed information about the step and the component of the step, which is the actual context-free AMP directive.</td></tr>
<tr><td>3</td><td>Depending on the nature of the request, one of the following outcomes occurs.
<table>
<tr><th>IF the plastic steps were generated as the result of ...</th><th>THEN they are sent to ...</th></tr>
<tr><td>a real data-containing request parcel</td><td>Apply and dispatched across the BYNET to the AMPs.</td></tr>
<tr><td>an EXPLAIN modifier request</td><td>the user in the form of a report on steps taken and the projected speed of their execution.</td></tr>
</table>
</td></tr>
</table>

Definition: Plastic Steps

The Generator creates a series of directives for the AMPs that contain column and row information, but no literal data values. The steps, called plastic steps, are later transformed into concrete steps by Apply. Plastic steps allow sets of values taken from the USING modifier to be inserted during subsequent operations.

The Generator writes the plastic steps into the request cache, then sends them to Apply. The plastic steps contain all the context-free information there is to know about a request and so are cached for potential reuse.

Parser logic assumes that requests with USING modifiers will be submitted repeatedly, and will thus benefit from reusing the plastic steps.

Statement Cache

Definition

The statement, or request, cache stores successfully parsed Teradata SQL requests so they can be reused, thereby eliminating the need for reparsing the same request parcel. The cache is a PE buffer that stores the steps generated during the parsing of a DML statement.

The Value of the Statement Cache

The statement cache is particularly useful for a batch update program that repeatedly issues the same request with different data values. This is so because all requests entered during a given session are routed to the same PE and thus access the same statement cache.

The statement cache is also useful in a transaction processing environment where the same DML statements are entered by a number of users from the same application program.

The Role of the Statement Cache in Statement Parsing

The statement cache is checked at the start of the parsing process, before the Syntaxer step, but after the Request parcel has been checked for format errors. If a matching cache request is found, the Parser bypasses the Syntaxer, Resolver, Optimizer, and Generator steps, does a security check (if required), and goes to Apply.

Because cached requests can be shared across logons, the Parser always makes a security check on cached requests. The first time a request is parsed, the Resolver builds a list of required access rights, and stores that list with the request. When the cached request is reused, the list of access rights is checked for the new user.

Immediate Caching

Not all requests are cached, and not all cached requests are cached immediately. If a request has a data parcel, it is cached immediately. Data parcels contain user-supplied information for a USING modifier.

The following examples show statements that produce requests with data parcels:

```
USING (A INTEGER, B INTEGER, C SMALLINT)
INSERT INTO TableX VALUES (:A, :B, :C);

USING (D SMALLINT, E INTEGER)
EXEC MacroY (:D, :E);
```

Non-Immediate Caching

Because they do not have data parcels, parameterized macros without USING modifiers are not cached immediately. The following macro does not have a data parcel because it does not have a USING modifier.

```
EXEC MacroZ (100, 200, 300);
```

The Parser considers parameter values provided at execution time to be a part of the request parcel, not a part of the data parcel.

If a request does not have a data parcel, its plastic steps are not immediately cached. Instead, a hash value derived from the request text is stored in one of the 100 first-seen entries in the cache management data structure.

If the same request comes through the Parser a second time, the request is cached, and its entry is moved from the first-seen area into one of the 300 cache entries in the data structure.

The Cache Management Structure

Each statement cache consists of two major entities:

- A data structure to manage the cached requests

  This structure consists of 300 slots, each of which contains information about one of the cached requests, plus 100 first-seen entries containing information about requests that have not yet been cached.

  The structure uses 32 kilobytes of RAM and is always memory-resident. Each cached request can be as large as 64 kilobytes (two 32 kilobyte segments), but is only as large as necessary.
- The individual requests

  The cached requests consist of the text of the SQL statements and their plastic steps.

  Individual cached requests are stored in PE memory or aged out to the PE swap area on its buddy AMP.

Although the statement cache is not shared among PEs, the data management structure is shared among all Parser tasks on the same PE, and cached requests can be shared across sessions and logons for a PE.

Criteria For Caching a DML Statement

If a DML statement has ...	THEN its steps are ...
a USING modifier	cached the first time the statement is parsed.
no USING modifier	not cached the first time the statement is parsed. If it is parsed a second time, then the steps are cached.

Criteria for Matching New Requests to Cached Requests

A new request is considered to match an existing cached request when all of the following criteria are met.

The new request must:

- Match the candidate cached request character-for-character.

  An exception is made for values in the data parcels of USING clauses.
- Be in the same mode (field, record, or indicator) as the candidate cached request.
- Have been submitted by the same type of application (batch or interactive) as the candidate cached request.
- Use the same default database as the candidate cached request.

  This criterion is required only if the database is needed in order to parse the request.
- Have been submitted from a host with the same host format as the candidate cached request.
- Use the same character set as the candidate cached request.
- Use the same collation sequence as the candidate cached request.

Statement Cache Matching Process

This table outlines the process used to determine whether a new request matches a cached request.

By using a combination of the hashed value derived from the request text, length information, and other flags, the Parser can make a preliminary identification of matching requests without comparing the new request to each cached request.

Stage	Process
1	The Parser creates a preliminary list of variables derived from the new request that consists of the following items: • SQL text hash • Request length • Other flags

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>2</td><td>The Parser tests the preliminary variable list for the new request against the same information for the requests in the statement cache.
<table>
<tr><th>IF the preliminary information ...</th><th>THEN the Parser ...</th></tr>
<tr><td>matches a cached request entry in the cache management structure</td><td>fetches the full cached request for comparison.</td></tr>
<tr><td>does not match a cached request entry in the cache management structure</td><td>compares a hash of the new request to the hash values of its first-seen entries.
<table>
<tr><th>IF ...</th><th>THEN ...</th></tr>
<tr><td>a match is found</td><td>the entry is moved to one of the slots in the data structure and its SQL text and plastic steps are stored in the request cache.</td></tr>
<tr><td>no match is found</td><td>the entry is marked as “first-seen” and cached for future use.</td></tr>
</table></td></tr>
</table></td></tr>
</table>

Aging Cached Requests to Buddy AMPs

Cached requests are aged to a swap area on the buddy AMP according to a least-recently-used algorithm. Aged requests are recorded as swap I/Os in ResUsage. Swapping cached requests back into the statement cache is done at the request of the Parser, and such swaps are *not* recorded as swap I/Os in ResUsage.

Maintaining Swap Area Free Space

If the free space in the buddy AMP swap area drops below 15 percent, the cache manager purges cached requests according to a least-recently-used algorithm until the free space exceeds its 15 percent threshold.

Purging the Statement Cache

Introduction

Requests remain in the statement cache until they are marked as spoiled, or no longer valid. Whenever the Parser receives a DDL statement, it broadcasts a spoil message to all PEs.

The spoil message causes those PEs to delete any cached requests (and their corresponding entries in the cache management data structure) that reference the database or table changed by the DDL statement.

Nonexempt Requests

Cached requests that are not marked as being exempt are purged periodically. The purge times are phased among the PEs so that all are not purged simultaneously.

Exempt Requests

An exempt request is one that would not be optimized differently if the demographics of the table were to change (assuming that table demographics might change over the period between cache purges).

If a request is exempt, it remains in statement cache until space is required, or until the system is restarted. Exempt requests include primary index requests that are independent of demographic changes, some types of requests that use USIs, and some types of nested joins.

Purging a Full Cache

When all 300 slots are full, the Parser uses a least-recently-used algorithm to determine which statements to delete from the cache management structure. When an entry is deleted from the data structure, its corresponding cached request is also deleted.

Purging Statistics-Bound Statement Cache Entries

Periodically, the statement cache is purged of all entries whose access or join plans are dependent on statistics. Entries that use only unique indexes for access are not affected by this periodic purge.

Purging Individual Statement Cache Entries

Statement cache entries are purged individually under the following conditions:

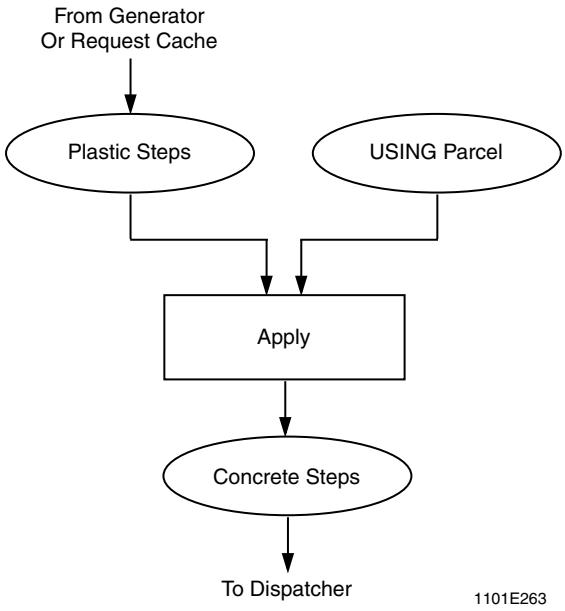
- The cache becomes full, and space is needed for a new entry. In this case, the steps for the least-recently used statement are discarded.
- A data definition statement (for example, ALTER TABLE) is entered for a table that has been specified by a cached statement. In this case, the steps for the statement are discarded.

Apply

Introduction

Apply places data from a USING data parcel, if present, into the plastic steps, thus producing concrete steps. Concrete steps are passed to the Dispatcher, which distributes them to the AMPs via the BYNET.

Block Diagram of Apply Activity



Apply Component Processes

Stage	Process
1	Apply receives the plastic steps from the Request Cache or from the Generator.
2	The plastic steps are compiled into executable machine code.
3	Apply retrieves any Data parcels associated with the Request parcel and applies them to the plastic steps, creating concrete steps.
4	The concrete steps are sent to the Dispatcher, which sends them across the BYNET to the AMPs.

Definition: Concrete Steps

Concrete steps are context- and data-laden AMP directives, containing user- and session-specific information in addition to Data parcels.

Each concrete step is composed of four parts.

Component	Description
System message header	Contains the message class and kind, length, logical host identifier, session and request number, version number, and character set code.
Common step header	Contains account numbers and routing information. Specifies who is to receive the step, what should be done with the responses generated by the step, how the AMPs signal task completion, and what is the context for the step (for example, a transaction number).
Step components	A series of operation-dependent fields including step flags and the step mode.
Data	Contained in a varying length data area.

Dispatcher

Introduction

The Dispatcher has three primary functions:

- To route AMP steps to the appropriate AMPs.
- To return the results of a request to the user application.
- To notify client applications and server processors of aborted transactions and requests.

Execution Control

Execution control is the term applied to Dispatcher handling of concrete steps.

The Dispatcher routes the Optimizer plan for a request (in the form of concrete steps) to the appropriate AMPs for processing.

Once a concrete step has been placed on the BYNET, it is referred to as an AMP step.

Part of the routing function of the Dispatcher is the sequencing of step execution. A new step is never dispatched until the previous step has completed its work, which is signalled by a completion response from the affected AMPs.

Depending on the nature of the request, an AMP step might be sent to one, several, or all AMPs (termed point-to-point, multicast, and broadcast, respectively).

Execution control also monitors the status reports of individual AMPs as they process the steps the Dispatcher has sent to them and forwards the results to the response control function once the AMPs complete processing each step.

Response Control

Response control is the term applied to Dispatcher handling of results. The second most important function of the Dispatcher is to return the (possibly converted) results of a request to the requesting application.

Transaction and Request Abort Management

The Dispatcher monitors transactions for deadlocks. When a deadlock occurs, the Dispatcher resolves it by managing the locked resources consistently and resolving the deadlock in the most optimal manner. This often means that one of a pair of transactions must be aborted.

The Dispatcher process request and transaction aborts by notifying both the client application and all affected server virtual processors.

Transactions and requests can abort for several reasons, both normal and abnormal.

Typical normal aborts are caused by the following actions.

- TDP logoff command
- Client application terminates normally

Typical abnormal aborts are caused by the following actions.

- Syntax and semantic errors in a query
- Internal Parser errors such as memory overflow
- Internal AMP errors such as primary index conflicts
- Transaction deadlocks

Chapter 2:

Query Optimization

This chapter describes query optimization. The information provided is designed to help you to interpret EXPLAIN reports more accurately and to explain and emphasize the importance of maintaining accurate statistical and demographic[1] profiles of your databases, not to describe query optimization as an abstraction.

Topics described include why relational systems need query optimizers, what query optimization is, and an overview of how query optimization is done. Special attention is paid to statistics, cost optimization, and join planning. Joins typically consume significantly more optimization time than other query processes with the exception of Cartesian products.

Among the topics covered are the following:

- Query optimizers
- Why column and index statistics are necessary for query optimization
- Interval histograms
- All-AMPs and random AMP sampled statistics
- When statistics should be collected or recollected
- How the optimizer uses statistical profiles
- Cost optimization
- Environmental cost factors
- Optimizer join plans
- Evaluating join orders
- Join methods

This chapter does not describe the Teradata Index Wizard, which is also a component of the Optimizer. For information on the Teradata Index Wizard, see Chapter 5: “Database Foundations for the Teradata Index Wizard.”

1 A statistic is a single number that represents an entire sample or population of data. For example, the mean of a distribution is a measure of its central tendency calculated by summing the data and then dividing that sum by the number of items summed. The actual data might not have a value identical to its mean, but the statistic provides a very valuable means for describing the data.

Demographics are raw, uncomputed characteristics of an entire set of data. For example, the highest and lowest values that occupy an interval are demographic measures, not statistics. Similarly, the cardinality of a table is the actual count of its rows: it is demographic information.

Query Optimizers

Introduction

Query optimizers have been likened to a travel agent: they book the most efficient route through the many joins, selections, and aggregations that might be required by a query.

Continuing the analogy, a Presidential candidate scheduled to make campaign speeches in a dozen cities across the country over a three-day period must rely on a travel agent who is sensitive to the time limitations, sequence of stops, and distance to be traveled when composing an itinerary. Query optimizers must be every bit as sensitive.

Why Relational Systems Require Query Optimizers

Optimization is required in relational systems to handle ad hoc queries. Unlike prerelational database management systems, the data in relational systems is independent of its physical implementation.

Queries in prerelational database systems do not require optimization because their access paths are determined in advance by the application programmer. The IBM hierarchical database management system IMS, for example, uses data structures called database descriptor blocks (DBD) and program control blocks (PCB) to define a database and how to access its records. Ad hoc queries are not possible in such a system because every possible way to access database records would have to be determined and specified in advance.

On the other hand, because any query that meets the syntactical and lexical rules of SQL is valid, a relational system must be able to access and join tables in many different ways depending on the different conditions presented by each individual query. This is particularly true for decision support and data mining applications, where the potential solution space is very large and the penalty for selecting a bad query plan is high.

If the demographics of the database change significantly between requests, the identical SQL query can be optimized in a radically different way. Without optimization, acceptable performance of queries in a relational environment is not possible.

What Is a Query Optimizer?

A query optimizer is an intricate software system that performs the following actions:

1. Translates an SQL query into some internal representation.
2. Rewrites the translation into a canonical form.
3. Assesses and evaluates candidate procedures for accessing and joining database tables.
4. Generates several candidate query plans[2] and selects the least costly plan from the generated set for execution.

SQL query optimizers are based on principles derived from compiler optimization and from artificial intelligence.

Parallels With Compiler Optimization

The parallels between SQL query optimization and compiler optimization are striking and direct. Any transformation of an input source code file into an optimized object code file by an optimizing compiler must meet the following criteria:

- Preserve the semantics of the original code.
- Enhance the performance of the program by a measurable amount over unoptimized code.
- Be worth the effort.

The following table indicates how these criteria generalize to the optimization of SQL queries.

This statement about compiler code optimization . . .	Generalizes to SQL query optimization as follows . . .
The transformation must preserve the semantics of the original code.	The transformed query must return the identical result as the original query.
The transformation must enhance execution of the program by a measurable amount over unoptimized code.	The response time for the transformed query must be significantly less than the response time for an unoptimized query.
The transformation must be worth the effort.	If the time taken to optimize the query exceeds the savings gained by reducing response time, there is no reason to undertake it.

2 A query plan is a set of low-level retrieval instructions called AMP steps that are generated by the Optimizer to produce a query result in the least expensive manner (see "Optimizer Cost Functions" and "Statistics and Cost Estimation" on page 2-59 for a description of query plan costing).

Parallels With Artificial Intelligence

SQL query optimization draws from two areas of artificial intelligence:

- Expert systems

  The Optimizer uses its knowledge base of column and index statistics to determine how best to access and join tables.

  Similarly, it consults its server configuration knowledge base (environmental cost variables) to determine how its statistical picture of the database relates to the partitioning of table rows across the AMPs.

  The Optimizer does not simply query these knowledge bases to return a result; it makes intelligent decisions on how to act based on the results it receives.
- Heuristics

  A heuristic is a rule of thumb: a method that generally produces optimum results through a series of successive approximations, but is not based on formal, provable rules.

  Heuristics do not necessarily produce the *best* solution to a query optimization problem, but they produce a reliably *better* solution than taking no action.

  The Optimizer uses heuristics at many different decision points. For example, heuristics are used to judge whether an intermediate join plan using single-table lookahead is good enough, or whether a five-table lookahead would generate a less costly plan. Similarly, heuristics are used to decide when to stop generating access plans for cost evaluation.

Objectives of Query Optimization

The principal objectives of SQL query optimization are the following:

- Maximize the output for a given number of resources
- Minimize the resource usage for a given output

The overriding goal of optimizers is to minimize the response time for a user query. Another way of viewing this is to rephrase the goal of query optimization in terms of cost: the object is to produce a query plan that has a minimal execution cost.

This goal can be achieved reasonably well only if user time is the most critical resource bottleneck; otherwise, an optimizer should minimize the cost of resource usage directly. These goals are not orthogonal; in fact, they are largely complementary.

Types of Query Optimizers

There are two basic types of query optimizers:

- Rule-based

  The determination of which candidate access plan to use is based on a set of ironclad rules.
- Cost-based

  The determination of which candidate access plan to use is based on their relative environmental and resource usage costs coupled with various heuristic devices. The least costly plan is always used.

Although the behavior of both types converges in theory, in practice, rule-based optimizers have proven to be less performant in decision support environments than cost-based optimizers. The Teradata query optimizer is cost-based.

What are the specific costs that a cost-based optimizer seeks to minimize? The Teradata Optimizer examines the following three cost criteria when it determines which of the candidate plans it has generated shall be used:

- Interconnect (communication) cost
- I/O costs, particularly for secondary storage
- CPU (computation) cost

Note that these costs are always measured in terms of time, and cost analyses are made in millisecond increments.

The plans generated by the Teradata optimizer are based entirely on various statistical data, data demographics, and environmental demographics. The Optimizer is not order-sensitive and its parallelism is both automatic and unconditional.

To summarize, the goal of a cost-based optimizer is not to unerringly choose *the* best strategy; rather, it is to find *a* best strategy from a set of candidate strategies and to avoid strategies that are clearly worse.

Teradata Optimizer Processes

Introduction

This topic provides a very high level survey of the stages of query optimization undertaken by the Teradata query optimizer. The information is provided only to help you to understand what sorts of things the Optimizer does and the relative order in which it does them.

Query Optimization Processes

The following table shows the logical sequence of the processes undertaken by the Teradata optimizer as it optimizes a DML statement. The input to this process is the Resolver ResTree (see "Resolver" on page 1-11).

Stage	Process
1	Marshal the predicates (see "Translation to Internal Representation" on page 2-8).
2	Perform preliminary processing of outer joins. **Stage** / **Process** 1 — When possible, convert outer joins to inner joins. 2 — Push down predicates. 3 — Build outer join terms. 4 — Generate information about dependencies between joined tables and their connecting conditions.
3	Process correlated subqueries by converting them to non-nested SELECTs or simple joins.
4	Search for a relevant join or hash index.
5	Materialize subqueries to spool files.
6	Analyze the materialized subqueries for optimization possibilities. **Stage** / **Process** 1 — Separate conditions from one another (see "Predicate Marshaling" on page 2-14). 2 — Push down predicates (see "Predicate Push Down and Pullup" on page 2-16). 3 — Generate connection information. 4 — Locate any complex joins.
7	Generate size and content estimates of spool files required for further processing (see "How the Optimizer Uses Statistical Profiles" on page 2-44).

Stage	Process
8	Generate an optimal single table access path.
9	Simplify and optimize any complex joins identified in stage 6.4.
10	Map join columns from a join (spool) relation to the list of field IDs from the input base tables to prepare the relation for join planning.
11	Generate information about local connections. A connecting condition is one that connects an outer query and a subquery. A direct connection exists between two tables if either of the following conditions is found. • ANDed bind; miscellaneous terms such as inequalities, ANDs, and ORs; cross, outer, or minus join term that satisfies the dependent information between the two tables • A spool file of an uncorrelated subquery EXIST predicate that connects with any outer table
12	Generate information about indexes that might be used in join planning, including the primary indexes for the relevant tables and pointers to the table descriptors of any other useful indexes.
13	Use a recursive greedy 1-table lookahead algorithm to generate the best join plan (see "One-Join Lookahead Processing of n-Way Joins" on page 2-76).
14	If the join plan identified in stage 13 does not meet the heuristics-based criteria for an adequate join plan, generate another best join plan using an *n*-table lookahead algorithm (see "Five-Join Lookahead Processing of n-Way Joins" on page 2-79).
15	Select the better plan of the two plans generated in stages 13 and 14.
16	Generate a star join plan (see "Star and Snowflake Join Optimization" on page 3-3).
17	Select the better plan of the selection in stage 15 and the star join plan generated in stage 16.
18	Generate plastic steps for the plan chosen in stage 17 (see "Definition: Plastic Steps" on page 1-22).

Translation to Internal Representation

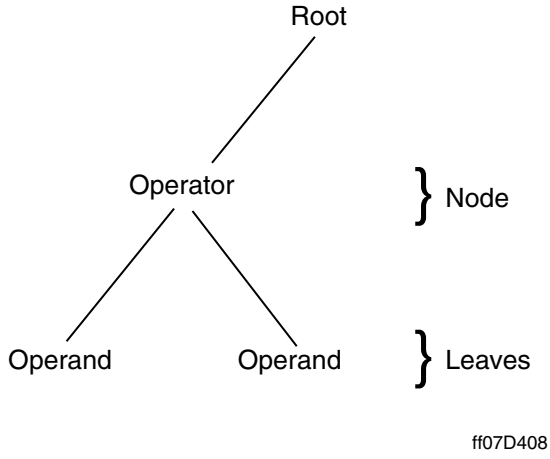
Parse Tree Representations of a Query

Like most cost-based optimizers, the Teradata optimizer represents queries internally as parse trees. A parse tree is a particular type of acyclic graph: a type of data structure that represents an input code set in terms of nodes and edges, with each arc in the tree specified by a pair of nodes. The foundation element for any parse tree is referred to as the root. Each node of the tree has 0 or more subtrees.

The elements of a parse tree are text strings, mathematical operators, parentheses, and other tokens that can be used to form valid expressions. These elements are the building blocks for SQL statements, and they form the nodes and leafs of the parse tree.

Parse trees are traversed from the bottom up, so the term "push down" means that an expression is evaluated earlier in the process than it would have otherwise been. Each node in the tree represents a database operation and its operands are the nodal branches, which are typically expressions of some kind, but might also be various database objects such as tables.

The following graphic illustrates a minimal parse tree having one node and two leaves.



Tables Used for the Examples

Consider how part of the following query could be represented and optimized using a parse tree. The tables used for the query example are defined as follows.

Parts

PartNum	Description	MfgName	MfgPartNum
PK		FK	
PI			
ND, NN			

Manufacturer

MfgNum	MfgName	MfgAddress	MfgCity
PK			
PI			
ND, NN			

Customer

CustNum	CustName	CustAddress	CustCity
PI			
PK			
ND, NN			

Order

OrderNum	CustNum	MfgPartNum	OrderDate
PI	FK		
PK			
ND, NN			

Example Query

Consider the following example query:

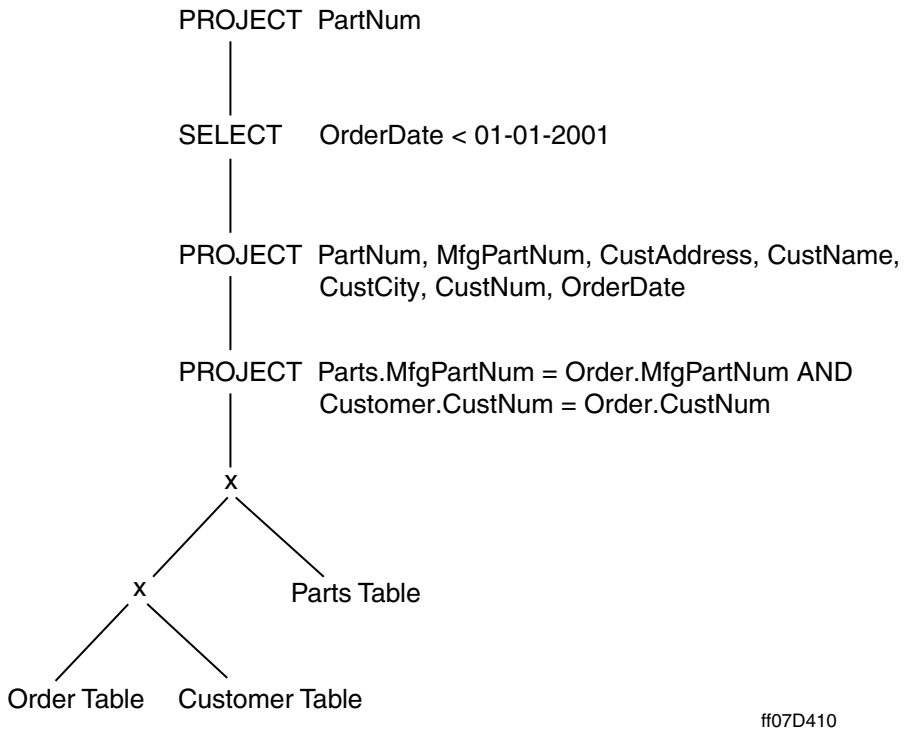
```
SELECT PartNum, Description, MfgName, MfgPartNum, CustName,
 CustAddress, CustNum, OrderDate
FROM Order JOIN Customer JOIN Parts
WHERE Customer.CustNum = Order.CustNum
AND Parts.MfgPartNum = Order.MfgPartNum
AND OrderDate < DATE '2001-01-01' ;
```

The initial translation of the SQL query into a parse tree is performed by the Syntaxer after it finishes checking the query text for syntax errors. The Optimizer receives a processed parse tree as input from the Resolver. This statement tree is just a parse tree representation of the original query text.

The Optimizer further transforms the tree by rewriting the query, determining an optimal access plan, and, when appropriate, determining optimal table join orders and join plans before passing the resulting parse tree on to the Generator.

At this point, the original statement tree has been discarded and replaced by an entirely new parse tree that contains instructions for performing the DML statement. The parse tree is now an operation tree. It is this tree, also referred to as a white tree, that the Optimizer passes to you as EXPLAIN text when you explain a query.

Assume the Optimizer is passed the following simplified parse tree by the Resolver (this tree is actually an example of a simple SynTree, but an annotated ResTree would needlessly complicate the explanation without adding anything useful to the description).



The Cartesian product operator is represented by the symbol x in the illustration.

The first step in the optimization is to marshal the predicates (which, algebraically, function as relational select, or restrict, operations) and push all three of them as far down the tree as possible. The objective is to perform all SELECTION and PROJECTION operations as early in the retrieval process as possible. Remember that the relational SELECT *operator* is an analog of the WHERE clause in SQL because it restricts the rows in the answer set, while the SELECT clause of the SQL SELECT *statement* is an analog of the algebraic PROJECT operator because it limits the number of columns represented in the answer set.

The process involved in pushing down these predicates is indicated by the following table. Some of the rewrite operations are justified by invoking various rules of logic. You need not be concerned with the details of these rules: the important thing to understand from the presentation is the general overall process, not the formal details of how the process can be performed.

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>Split the compound ANDed condition into separate predicates. The result is the following pair of SELECT operations.

SELECT Customer.CustNum = Order.CustNum

SELECT Parts.MfgPartNum = Order.MfgPartNum</td></tr>
<tr><td>2</td><td>By commutativity, the SELECTION Order.OrderDate < 2001-01-01 can be pushed the furthest down the tree, and it is pushed below the PROJECTION of the Order and Customer tables.

This particular series of algebraic transformations, which is possible because OrderDate is an attribute of the Order table only, is as follows.

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>Begin with the following predicate.

SELECT OrderDate < 2001-01-01((Order X Customer) X Parts)</td></tr>
<tr><td>2</td><td>Transform it to the following form.

SELECT (OrderDate < 2001-01-01(Order X Customer)) X Parts</td></tr>
<tr><td>3</td><td>Transform it further to the following form.

SELECT ((OrderDate < 2001-01-01(Order)) X Customer) X Parts

This is as far as the predicate can be transformed, and it has moved as far down the parse tree as it can be pushed.</td></tr>
</table>
</td></tr>
<tr><td>3</td><td>The optimizer examines the following SELECT operation to see if it can be pushed further down the parse tree.

SELECT Parts.MfgPartNum = Order.MfgPartNum

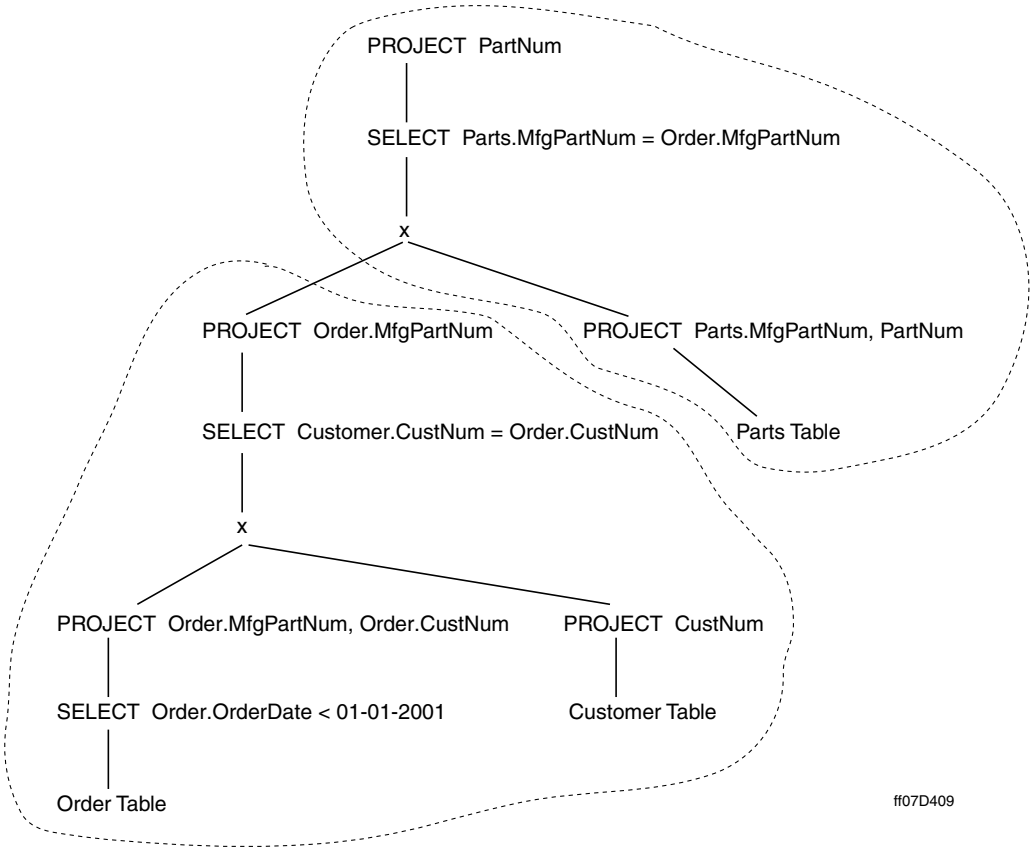
Because this SELECTION contains one column from the Parts table and another column from a different table (Order), it cannot be pushed down the tree any further than the position it already occupies.</td></tr>
<tr><td>4</td><td>The optimizer examines the following SELECT operation to determine if it can be pushed any further down the parse tree.

SELECT Customer.CustNum = Order.CustNum

This expression can be moved down to apply to the product OrderDate < 2001-01-01 (Order) X Customer.

Order.CustNum is an attribute in SELECT Date < 2001-01-01(Order) because the result of a selection accumulates its attributes from the expression on which it is applied.</td></tr>
<tr><td>5</td><td>The optimizer combines the two PROJECTION operations in the original parse tree into the single PROJECTION PartNum.

The structure of the parse tree after this combination is reflected in the following illustration.</td></tr>
</table>



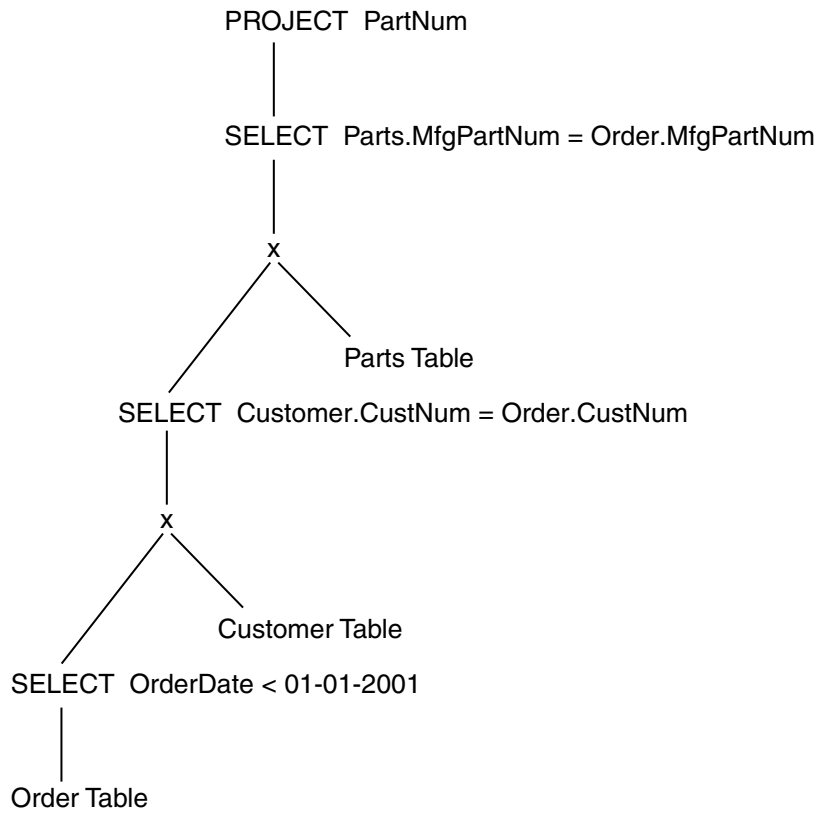
Stage	Process

6	This intermediate stage of the parse tree can be further optimized by applying the rules of commutation for SELECT and PROJECT operations and replacing `PROJECT PartNum` and `SELECT Customer.CustNum = Order.CustNum` by the following series of operations.
	`PROJECT Parts.PartNum`
	`SELECT Parts.MfgPartNum = Order.MfgPartNum`
	`PROJECT Parts.PartNum, Parts.MfgPartNum, Order.MfgPartNum`

7	Using the rules of commutation of a PROJECTION with a Cartesian product, replace the last PROJECTION in Stage 6 with the following PROJECTION. `PROJECT PartNum, Parts.MfgPartNum`

8	Similarly, apply `PROJECT Order.MfgPartNum` to the left operand of the higher of the two Cartesian products. This projection further interacts with the SELECT operation immediately below it (`Customer.CustNum = Order.CustNum`) to produce the following series of algebraic operations.
	`PROJECT Order.MfgPartNum`
	`SELECT Customer.CustNum = Order.CustNum`
	`PROJECT Order.MfgPartNum, Customer.CustNum, Order.CustNum`

Stage	Process
9	The last expression from Stage 8 bypasses the Cartesian product by commutation of PROJECTION with a UNION operation and partially bypasses the SELECTION operation `SELECT Order.OrderDate < 01-01-2001` commutation.
10	The optimizer sees that the resulting expression `PROJECT Order.MfgPartNum, Order.CustNum, OrderDate` is redundant with respect to its PROJECTION term, so it is removed from further consideration in the query. The transformed parse tree that results from all these transformations is shown in the following illustration.



The two Cartesian product operations are equivalent to equijoins when they are paired with their higher selections (where *higher* indicates that the operations are performed later in the execution of the query). Note that operators are grouped in the graphic, illustrated by boundary lines. Each bounded segment of the parse tree would become an AMP worker task.

As noted at the beginning of this topic, parse trees are always presented upside down, so query execution begins with the lower cluster of operations and terminates with the upper cluster. In an EXPLAIN of this query, the expressions in the upper cluster would be referred to as residual conditions.

Query Rewrite

Introduction

Query rewrite is arguably the most fundamental process undertaken by a query optimizer, and is the first piece of true optimization performed. The fundamental motivation behind query rewrite is to "flatten" the code in order to make optimization of query components such as views, correlated subqueries, and aggregates more tractable for optimization.

No query optimizer can consistently rewrite a poorly coded query to make it as efficient as a well-coded, semantically equivalent query, but a good optimizer can certainly close the gap between the two.

With many SQL queries now being created by query generator tools that do not write optimally efficient code, query rewrite has become more crucial than ever. The Teradata Optimizer rewrites queries aggressively in order to produce query plans that are not exceeded in the commercial relational database market.

Typical Query Rewrite Techniques

Because relational systems are based on rigorous mathematical principles, there is a good deal of algebraic simplification and term reduction that can be made for most queries. Similarly, there are often semantically equivalent operations that can be substituted for analytically more difficult operations in order to simplify processing. Most of these semantically equivalent rewrites cannot be specified in the query itself, an example being the substitution of a join index for an explicitly specified join operation.

The following sequence lists many of the more important methods used by query optimizers to rewrite SQL code:

- Predicate marshaling
- Predicate push down and pullup
- Correlated subquery decorrelation
- Redundant join elimination
- Aggregate sharing
- View materialization and other database object substitutions

Predicate Marshaling

The first step in the process of query rewrite is to marshal the predicates, both ordinary and connecting, for the DML operations presented to the Optimizer by the vanilla Resolver ResTree. This means that first that the query predicates are, where possible, isolated from the parse tree, converted to conjunctive normal form, or CNF (see “Path Selection” on page 2-57 for a definition of

CNF), and then they might be combined with other predicates or eliminated from the query altogether by a process analogous to factoring and cancellation in ordinary algebra.

Conditions on multiple relations are converted by expansion. For example,

```
A + CD → (A+C) (A+D)
AB + CD → (A+C) (A+D) (B+C) (B+D)
```

Conditions on a single relation are *not* converted because they are needed to generate efficient access paths for that relation. Consider the following single-relation condition set.

```
(NUSI_1 = 1 AND NUSI_2 = 2) OR (NUSI_1 = 3 AND NUSI_2 = 4)
```

In this case, each ORed expression can be used to read the NUSI subtable to generate a RowID spool.

Where possible, new conditions are derived using transitive closure[3]. For example,

```
A = B AND A = C → B = C
A = 5 AND A = B → B = 5
A = 5 AND B = 5 → A = B
```

where the symbol → indicates transformation.

Transitive closure is *not* applied to the conditions specified in the WHERE clause and ON clause of an outer join because of the possible semantic ambiguities that could result. As a general rule, a single relation condition on an outer table cannot be combined with terms from an outer block or WHERE clause.

---

3 Transitive closure is an algebraic method the Optimizer uses to solve questions of the sort "can I get to point b from point a"?

For example, if x>1 and y>x, then the implication is that y>=3. Similarly, if x is in the set (1,2,3) and y=x, then the implication is that y is also in (1,2,3).

Consider a simple SQL example. The following SELECT statement implies that x<1.

```
SELECT *
FROM t1
WHERE x IN
 (SELECT y
  FROM t2 WHERE y<1);
```

Similarly, the following SELECT statement implies that x < 3 and y is in (1,4):

```
SELECT *
FROM t1
WHERE EXISTS
 (SELECT *
  FROM t2
  WHERE y<3
  AND x=y)
  AND x IN (1,4);
```

In the current context, the conditions under analysis are referred to as connecting conditions.

A connecting predicate is one that connects an outer query with a subquery. For example, consider the following transformation.

```
(table_1.x, table_1.y) IN
 (SELECT table_2.a, table_2.b
  FROM table_2)
→
(table_1.x IN spool_1.a) AND (table_1.y IN spool_1.b)
```

Similarly, the following transformation deals with a constant by pushing it to spool.

```
(table_1.x, constant) IN
 (SELECT table_2.a, table_2.b
  FROM table_2)
→
(table_1.x IN spool_1.a)
```

The term `(table_2.b = constant)` is pushed to spool_1.

The following transformation is more complex.

```
(table_1.x, table_1.y) IN
 (SELECT table_2.a, constant
  FROM table_2)
→
(table_1.x IN spool_1.a) AND (table_1.y = spool_1.constant)
```

The more connecting conditions available, the more flexible the plan.

The next step is to push the marshaled predicates down the parse tree as far as possible. See "Translation to Internal Representation" on page 2-8 and "Predicate Push Down and Pullup" on page 2-16 for additional information on predicate push down and pullup.

Predicate Push Down and Pullup

The concept of parse trees was introduced in "Translation to Internal Representation" on page 2-8, as was predicate push down.

The justification for predicate push down is to move operations as far to the beginning of query processing as possible in order to eliminate as many rows (using SELECTION) and columns (using PROJECTION) as possible to reduce the cost of the query to a minimal value. This manipulation is particularly important in the Teradata parallel environment because it reduces the number of rows that must be moved across the BYNET to other AMPs. Because query processing begins at the bottom of the parse tree, these predicates are said to be pushed *down* the tree.

Predicate pullup is less frequently used than push down in query optimization. Its typical purpose is to move more expensive operations further back in the processing queue so they have fewer rows and columns on which to operate. Because query processing begins at the bottom of the parse tree, these predicates are said to be pulled *up* the tree.

Correlated Subquery Decorrelation

Correlated subqueries present a potentially expensive problem in a parallel environment (depending on how the relevant tables are partitioned) because all rows from one of the tables would have to be broadcast to all the AMPs to fulfill the request. The performance problems introduced by most correlated subqueries can be reduced, if not eliminated entirely, by converting them to simple joins. This conversion makes correlated subqueries more performant in general, but particularly when specially optimized join algorithms exist to deal with them. The Teradata optimizer can choose from among six different join algorithms when rewriting correlated subqueries. To read more about them, see "Correlated Joins" on page 2-119.

Consider the following example of decorrelating a correlated subquery. Keep in mind that this example is an SQL text representation of how the Optimizer rewrites the query by optimizing elements of the parse tree. It is not meant to portray a precise representation of how the Optimizer actually rewrites the query.

The query probes two tables: an Order table and a Shipper table. Because the Order table is partitioned by order_number, brute force performance of this query requires first that the relevant Shipper table rows be broadcast to all AMPs, and second that the correlated subquery be performed multiple times: once for every cancelled order in the Order table that was to be shipped by truck in the Central States region.

```
SELECT SUM(order.order_amount), COUNT(*)
FROM order, shipper
WHERE order.shipperID = shipper.shipperID
AND shipper.region = 'central states'
AND order.order_type = 'ready'
AND order.shipper_type = 'truck'
AND order.order_amount >
  (SELECT 2.5 * AVG(o_amount)
   FROM order o
   WHERE o.shipperID = shipper.shipperID
   AND o.order_type = 'ready'
   AND o.shipper_type = 'rail';
```

The rewrite of this query uses an aggregate derived table named shipper_aggregate to decorrelate the query. In the rewritten query, the average value of rail orders for each shipper is computed once and broadcast across the AMPs.

```
SELECT SUM(order.order_amount), COUNT(*)
FROM order, shipper
WHERE order.shipperID = shipper.shipperID
AND shipper.region = 'central states'
AND order.order_type = 'ready'
AND order.shipper_type = 'truck'
AND order.order_amount >
  (SELECT shipper_aggregate.order_trigger_amount
   FROM
     (SELECT o.shipperID, 2.5 * AVG(o.o_amount) order_trigger_amount
      FROM order o, shipper s
      WHERE o.shipperID = s.shipperID
      AND o.shipper_region = 'central states'
      AND o.order_type = 'ready'
      AND o.shipper_type = 'rail'
      GROUP BY s.shipperID) AS shipper_aggregate
   WHERE shipper_aggregate.shipperID = shipper.shipperID;
```

Redundant Join Elimination

Joins are among the most frequently specified operations in SQL queries. They are also among the most demanding resource consumers in the palette of an SQL coder. Because of its many optimized join algorithms, Teradata processes joins with relative ease. Nevertheless, when joins can be eliminated or made less complex, the result is always better performance. An excellent example of how the Teradata optimizer recognizes and eliminates unnecessary or redundant joins is provided in "Outer Join Case Study" and its subordinate topics in the chapter "Join Expressions" in *Teradata RDBMS SQL Reference, Volume 6*. The topics "First Attempt" and "Third Attempt" are particularly good examples of how the Optimizer can streamline poorly written joins.

Aggregate Sharing

Wherever possible, a query optimizer reduces the number of aggregate computations by consolidating and sharing aggregations.

Consider the following simple query. It performs two summations and two counts (the AVG function performs a count and a sum internally).

```
SELECT SUM(order.order_amount), AVG(order.order_amount),
       COUNT(*)
FROM order ;
```

The rewritten query, which looks more complex than its predecessor, uses a derived table named order_aggregate to perform SUM and COUNT once each and then pools the result. The transformation accelerates performance enormously.

```
SELECT order_aggregate.order_amount_sum,
(order_aggregate.order_amount_sum /
order_aggregate.order_count), order_aggregate.order_count
FROM
  (SELECT SUM(order.order_amount) AS order_amount_sum,
   COUNT(*) AS order_count
   FROM order)
AS order_aggregate ;
```

Like the example used to illustrate decorrelation of a correlated subquery, this is an SQL text representation of how the Optimizer rewrites the query by optimizing elements of the parse tree. It is not meant to portray a precise representation of how the Optimizer actually rewrites the query.

View Materialization[4] and Other Database Object Substitutions

Perhaps the most obvious query rewrites concern instantiating virtual database objects and replacing specified structures with more performant substitutes when possible.

All views referenced by a query must be resolved into their underlying base tables before the query can be performed.

The Optimizer also replaces base tables and table joins with hash, join, or even secondary indexes whenever the substitution makes the query more performant. These substitutions also apply to view materialization if base tables referenced by the view can be profitably replaced.

4 The term materialized view is sometimes used to describe database objects like snapshot summary tables, and even hash or join indexes. For purposes of the current discussion, the term literally refers to materializing a view in terms of its base tables or base table substitutions.

Why Are Statistics Necessary?

Introduction

The Optimizer uses statistics for several different purposes. Without full or all-AMPs sampled statistics, query optimization must rely on a random AMP sample estimate of cardinality and the number of distinct index values plus a small set of system-defined heuristics for various predicate selectivities.

Statistics provide the Optimizer with information it uses to reformulate queries in ways that permit it to produce the least costly access plan. The critical issues you must evaluate when deciding whether to collect statistics are not whether query optimization can or cannot occur in the face of inaccurate statistics, but the following pair of antagonistic questions:

- How accurate must the available statistics be in order to generate the best possible query plan?
- How poor a query plan you are willing to accept?

Purposes of Statistics

The following list is a very high-level description of the most important purposes for column and index statistics.

- The Optimizer uses statistics to decide whether it should generate a query plan that use a secondary, hash, or join index instead of performing a full-table scan.
- The Optimizer uses statistics to estimate the cardinalities of intermediate spool files based on the qualifying conditions specified by a query.

  The estimated cardinality of intermediate results is critical for the determination of both optimal join orders for tables and the kind of join method that should be used to make those joins.

  For example, should 2 tables or spool files be redistributed and then merge joined, or should one of the tables or spool files be duplicated and then product joined with the other. Depending on how accurate the statistics are, the generated join plan can vary so greatly that the same query can take only seconds to complete using one join plan, but take hours to complete using another.

In many cases, random all-AMPs (see “All-AMPs Sampled Statistics” on page 2-33) or random AMP sampled statistics (see “Random AMP Sampling” on page 2-35) are not accurate enough for the Optimizer to generate an optimal, or even a good, join plan. However, it is sometimes true that statistics collected by sampling small subpopulations of table rows can be as good as those collected with a full-table scan. The value of collecting full-table statistics is that they provide the Optimizer with the most accurate information that can be gathered for making the best possible query plan cost estimates.

Statistical accuracy is fundamentally important for any query plan because the effect of suboptimal access and join plans generated from inaccurate statistics, of which there can be many in the optimization of a complex query, is multiplicative.

A query plan generated using full-table statistics is guaranteed to be at least as good as a query plan generated using any form of sampled statistics[5].

Is It Better To Collect Full-Table Statistics or Sampled Statistics?

When viewed in isolation, the decision between full-table and all-AMPs sampled statistics is a simple one: always collect full-table statistics because they provide the best opportunity for producing optimal query plans.

While statistics collected from a full-table scan are an accurate representation of the entire domain, an all-AMPs sample estimates statistics based on a small sample of the domain, and a random AMP sample is not only based on an even smaller, more cardinality- and skew-sensitive sample of the domain, it also generates far fewer, more granular statistics.

Unfortunately, this decision is not so easily made in a production environment. Other factors must be accounted for, including the time required to collect the statistics and the resource consumption burden the collection of full-table statistics incurs.

In a production environment running multiple heavy query workloads, the problem concerns multiple levels of optimization:

Level	Type of Optimization	Questions Asked
Bottom	Query	If collecting full-table statistics makes queries run faster, what reasons could there be for collecting less accurate statistical samples?
Middle	Workload	If the act of collecting full-table statistics makes the system run slower, is there any way to collect statistical samples of table populations that are reasonably accurate and that will produce reasonably good query plans?
Top	Mix	What mix of query and workload optimization is best for overall system performance?

5 There is a high probability that a query plan based on full-table statistics will be better, and sometimes significantly better, than a plan based on any form of sampled statistics.

Time and Resource Consumption As Factors In Deciding How To Collect Statistics

The elapsed time to collect statistics and the resources consumed in doing so are the principal factors that mitigate collecting statistics on the entire population of a table rather than collecting what are probably less accurate statistics from a sampled subset of the full population which, in turn, will probably result in less optimal query plans.

The elapsed time required to collect statistics varies as a function of the following factors:

- Base table cardinality
- Number of distinct index or non-indexed column values
- System configuration

For a 32-node production system, collecting statistics on a single column of a 100 million row table might take an hour. If you collect statistics on multiple columns and indexes of the same table, the process can easily take 4 hours to complete. When you add the time required to collect statistics for numerous smaller tables in the database to the mix, the time required to collect statistics for all the tables can be surprisingly large. You might even decide that the necessary time for collection is excessive for your production environment, particularly if you have a narrow time window for collecting statistics.

Collecting full-table statistics is not just time-consuming; it also places a performance burden on the system, consuming CPU and disk I/O resources that would otherwise be devoted to query processing. You might decide that collecting statistics places too many burdens on system resources to justify recollecting statistics on a daily, weekly, or even monthly basis.

After examining all these considerations, you might even conclude that *any* recollecting of statistics is an unacceptable burden for your production environment.

If, and only if, you find yourself in this situation, you should consider collecting sampled full-table statistics rather than not refreshing your statistics at all. You can do this by specifying the USING SAMPLE option of the COLLECT STATISTICS statement (see “COLLECT STATISTICS (Optimizer Form)” in the chapter “Query and Workload Analysis Statements” in *Teradata RDBMS SQL Reference, Volume 4*). Collecting sampled full-table statistics should be approximately one to two orders of magnitude faster than collecting full-table statistics, though the accuracy of the statistics collected this way is never better, and usually is not as good.

If you decide to take this path, it is important to examine the appropriate EXPLAIN reports carefully to ensure that the query plans generated with these less accurate statistics are acceptable for your application environment. If your query plans are as good as those generated with full-table statistics, or at least are reasonably good, then the sampled statistics are probably adequate. On the other hand, if the Optimizer does *not* generate good query plans using sampled statistics, then you should strongly consider finding a way to collect full-table statistics.

An Example of How Old Statistics Can Produce a Poor Query Plan

The following example is admittedly extreme, but is instructive in demonstrating how negatively bad statistics can affect the query plan the Optimizer generates.

Consider the following two tables:

Table Name	Statistics Collected?	Cardinality When Statistics Were Collected	Current Cardinality
A	Yes	1 000	1 000 000
B	No		75 000

If a product join between table A and table B is necessary for a given query, and one of the tables must be duplicated on all AMPs, then the Optimizer will select Table A to be duplicated because, as far as it knows from the available statistics, only 1 000 rows must be redistributed, as opposed to the far greater 75 000 rows from table B.

In reality, table A currently has a cardinality of one million rows, not one thousand rows, and the Optimizer makes a very bad decision by duplicating the million rows of table A instead of the seventy five thousand rows of table B). As a result, the query runs much longer than necessary.

There are two general circumstances under which statistics can be considered to be stale:

- The number of rows in the table has changed significantly.
- The range of values for a index or column of a table for which statistics have been collected has changed significantly.

  Sometimes you can infer this from the date and time the statistics were last collected, or by the nature of the column. For example, if the column in question stores transaction dates, and statistics on that column were last gathered a year ago, it is almost certain that the statistics for that column are stale.

You can obtain the number of unique values for each statistic on a table, as well as the date and time the statistics were last gathered, using the following query:

```
HELP STATISTICS tablename;
```

For statistics on unique indexes, you can cross check values reported by HELP STATISTICS by comparing the row count returned by the following query:

```
SELECT COUNT(*)
FROM tablename;
```

For statistics on nonunique columns, you can cross check the HELP STATISTICS report by comparing it with the count returned by the following query:

```
SELECT COUNT(DISTINCT columnname)
FROM tablename;
```

Teradata Statistics Wizard

The Teradata Statistics Wizard client utility provides an objective method of determining when fresh statistics should be collected on a table, column, or index based on a set of user-provided SQL query workloads. The Statistics Wizard also advises you on which tables, columns, and indexes statistics should be collected as well as providing a means of verifying the enhanced efficiency of queries run against the workloads.

See *Teradata Statistics Wizard User Guide* for additional information about the uses and capabilities of this utility.

Interval Histograms

Introduction

A synopsis data structure is a data structure that is substantially smaller than the base data it represents. Interval histograms are a form of synopsis data structure: they provide a useful statistical profile of attribute values that characterizes the properties of that raw data.

Teradata uses interval histograms to represent the cardinalities and certain other statistical values and demographics of columns and indexes for all-AMPs sampled statistics and for full-table statistics. Each histogram is composed of a maximum of 100 intervals.

The statistical and demographic information maintained in the histograms is used to estimate various attributes of a query, most notably the magnitudes of the relations that are produced during the resolution of the request.

The use of interval histograms to make these estimates is a nonparametric method. Nonparametric statistical analyses are used whenever the population parameters of a variable, such as its distribution, are not known.

Interval Histograms Used By Teradata

Depending on the distribution of values (degree of skewness) in a column, any one of three possible histogram types is used to represent its cardinality and statistics.

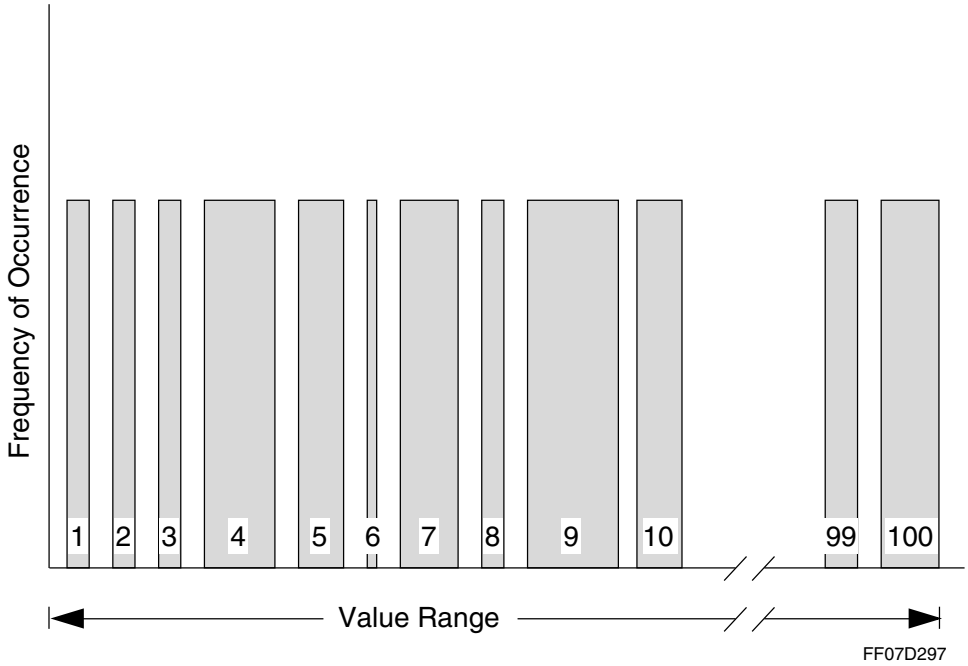
- Equal-height interval histogram

  In an equal-height interval histogram, each interval has the same number of values. To achieve this, the interval widths must vary.

  When a histogram contains 100 equal-height intervals, each interval effectively represents a single percentile score for the population of attribute values it represents.

  The statistics for a column are expressed as an equal-height interval histogram if the frequencies of its values are normally distributed (not skewed).

The following graphic illustrates the concept of an equal-height interval histogram:



- High-biased interval histogram

  In a high-biased interval histogram, each interval has at most two values.

  High-biased intervals are used only when there is significant skew in the frequency distribution of values for the column.

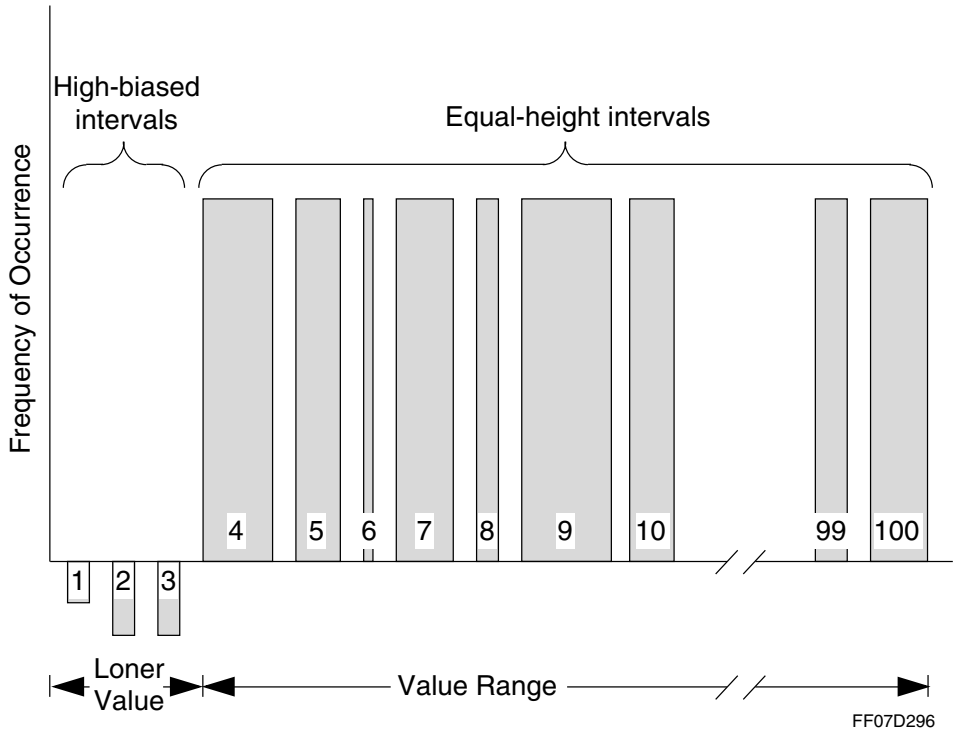
WHEN the interval stores this many loners . . .	THEN "values" is set to . . .
1	-1
2	-2

- Compressed histogram

  Compressed histograms contain a mix of 100 equal-height and high-biased intervals.

  Reading from left-to-right, high-biased intervals always precede equal-height intervals in the histogram.

  The following graphic illustrates the concept of a compressed histogram:



  Note the first three intervals, which define high-biased loners, have the values -1, -2, and -2 respectively.

Terminology

Term	Definition
Bucket	A synonym for an interval in an interval histogram.
Cardinality	The number of rows per AMP that satisfy a predicate condition. Note that in this context, cardinality generally is not the number of rows in the entire table, but the number of rows that qualify for a predicate filter. See "Examples of How Cardinality Estimates Are Made for Simple Queries Using Interval Histograms" on page 2-46 for a fuller explanation of what cardinality means in this context.

Term	Definition
Compressed interval histogram	A family of histograms that combines high-biased intervals with equal-height intervals. The high-biased intervals are always stored in buckets 1 through *n*, where *n* represents the highest numbered bucket containing high-biased interval information. Equal-height intervals begin at bucket *n* + 1.
Equal-height interval	An interval containing column statistics normalized across the distribution in such a way that the graph of the distribution of the number of values as a function of interval number is flat. This is achieved by varying the width of each interval so it contains the same number of values (but usually different attribute value ranges) as its neighbors.
Equal-height interval histogram	A family of histograms characterized by equal numbers of occurrences and non-constant attribute value ranges per bucket.
High-biased interval	An interval used to characterize a skewed value set for a column. Any attribute value that is significantly skewed (see the statistic defined under "Loner" on page 2-29) is summarized by a high-biased interval. Each high-biased interval contains statistics for at most two attribute values.
High-biased interval histogram	A family of histograms characterized by many loner buckets as seen, for example, with a multimodal or otherwise-skewed attribute value distribution.
Histogram	A graphic means for representing distributions as a function of the number of elements per an arbitrarily determined interval width. Histograms are often called bar charts. Each bar in a histogram represents the frequency of occurrence of the value or range of values for the defined interval. Histogram intervals are sometimes referred to as buckets because they contain a number of values that summarize the demographics for the values that fall into the range defined for the interval. In Optimizer theory, the term is used to describe the rows in a dictionary table that store the buckets defined by the particular intervals used to characterize the frequency distribution of the attribute values for a column. All histograms described in this topic are frequency histograms. Each bucket in a frequency histogram contains some number of tokens representing the frequency of occurrence of the attribute values belonging to its range.
Interval	A bounded, non-overlapping set of attribute values.

<table>
<tr><th>Term</th><th>Definition</th></tr>
<tr><td>Loner</td><td>A loner is an attribute value whose frequency in the sampled population deviates significantly from a defined criterion; an unusually frequent value indicating significant frequency skew. By definition, no more than two loners are stored per high-biased interval.

The formal definition of a loner is as follows:
$f \geq \frac{T}{200}$
where:<table><tr><th>This variable ...</th><th>Defines the ...</th></tr><tr><td>f</td><td>frequency of the loner.</td></tr><tr><td>T</td><td>cardinality of the base table.</td></tr></table>
By this definition, the maximum number of loners in a histogram is 200. In practice, a column might produce more than 200 loners, so the implemented upper bound on loners is 198 (or 99 intervals). This leaves one equal-height interval to account for the remaining values in the distribution.</td></tr>
<tr><td>Skew</td><td>A measure of the asymmetry of the distribution of a set of attribute values.

Skewness is the third moment of the probability density function for a population of attribute values.

The first two moments are the mean and the standard deviation, respectively.

With respect to skew in parallel databases, there are several possible types.
• Attribute value skew refers to skew that is inherent in the data. An example might be a column that can take on only two values.
• Partition skew refers to skew that results from an uneven distribution of data across the AMPs.

The difference is apparent from the context. As used in this manual, the term usually refers to the partition skew that occurs when the primary index for a table is defined on a column set that is highly nonunique.</td></tr>
</table>

Content and Storage of Statistical Histograms

Index and column histograms are stored in the FieldStatistics column of the DBC.TVFields data dictionary table.

The following table lists details of the data stored in this column:

Position	Field Length (Bytes)	Information
1	2	Length of this field.
2	8	Time the statistics were collected in the following format: Information Stored / Number of Bytes Year / 2 Month / 1 Day / 1 Hour / 1 Minute / 1 Second / 1 Centisecond / 1
3	2	Not used.
4	2	Statistics version.
5	8	Number of nulls in the column or index.
6	2	Number of intervals for the column or index.
7	1	Flag indicating whether stored statistics are numeric.
8	1	Not used.

<table>
<tr><th>Position</th><th>Field Length (Bytes)</th><th>Information</th></tr>
<tr><td>9</td><td>40</td><td>Interval 0 contains global statistics for the column or index. The following information is stored in each field of Interval 0:

<table><tr><th>Field</th><th>Statistic</th></tr><tr><td>1</td><td>Minimum value for the column or index.</td></tr><tr><td>2</td><td>Modal value for the column or index.</td></tr><tr><td>3</td><td>Number of occurrences of the modal value for the column or index.</td></tr><tr><td>4</td><td>Number of distinct values for the column or index.</td></tr><tr><td>5</td><td>Cardinality of the column or index.</td></tr></table></td></tr>
<tr><td>10-109</td><td>40 per interval</td><td>Intervals 1 - 100 contain information specific to their respective histogram.

<table><tr><th>Field</th><th>Statistic</th></tr><tr><td>1</td><td>Minimum value for the interval.</td></tr><tr><td>2</td><td>Modal value for the interval.</td></tr><tr><td>3</td><td>Number of occurrences of the modal value in the interval.</td></tr><tr><td>4</td><td>Number of distinct non-modal values in the interval.</td></tr><tr><td>5</td><td>Number of rows containing distinct non-modal values for the interval.</td></tr></table></td></tr>
</table>

Rows for each histogram are separated into three categories:

- Nulls
- Loners
- Non-loners

Recall that a histogram can be composed of any of the following types of intervals in addition to Interval 0:

- 99 high-biased (loner) intervals and one equal-height interval
- All equal-height (non-loner) intervals
- A mix of high-biased and equal-height intervals (compressed histogram).

Only the first 16 bytes of non-numeric values are stored, so loners whose first 16 bytes match are stored contiguously. To address this storage with respect to estimating cardinalities, the statistic for calculating cardinality must be adjusted.

The adjusted statistic for estimating the cardinality for equivalent loner values is given by the following equation.

$$C = \frac{\Sigma f^2}{\Sigma f}$$

where:

This variable ...	Represents the ...
C	estimated cardinality of the relation.
f	frequency of the loner value in question.

Consider the following very simple examples:

For this value of f ...	The cardinality estimate is ...	Comment
1	1	None.
1 and 3	2.5	This estimate is more meaningful than the simple arithmetic average, which is 2, because the loner having $f = 3$ should weight the estimate more heavily than the loner having $f = 1$.

All-AMPs Sampled Statistics

Introduction

There are occasions when it becomes difficult to spend the effort required to collect full-table statistics. For example, suppose you have a multi-billion row table that requires several hours collection time, and a narrow collection window. If you encounter situations like this, for which it is not possible to collect full-table statistics, you can opt to collect statistics on a sample of table rows instead.

Caution The quality of the statistics collected with full-table sampling is not as good as the quality of statistics collected on an entire table. Do not think of sampled statistics as an alternative to collecting full-table statistics, but as an alternative to not collecting statistics at all.

When you use sampled statistics rather than full-table statistics, you are trading time in exchange for less accurate statistics. The underlying premise for using sampled statistics is nothing more than the assumption that sampled statistics are better than no statistics.

Sampled Statistics Are More Accurate Than Random AMP Statistical Samples

Do not confuse statistical sampling with the random AMP samples that the Optimizer collects when it has no statistics on which to base a query plan. Statistical samples are likely to be much more accurate than random AMP samples. The following table describes some of the differences between the two methods of collecting statistics.

Statistical Sampling	Random AMP Sampling
Collects statistics on a small sample of rows from all AMPs.[a]	Collects statistics on a small sample of rows from a single AMP.
Collects full statistics (see "Content and Storage of Statistical Histograms" on page 2-30) and stores them in histograms in DBC.TVFields.	Collects estimates for cardinality and number of distinct index values only.
Expands sample size dynamically to compensate for skew.	Sensitive to skew.

Statistical Sampling	Random AMP Sampling
Provides fairly accurate estimates of all statistical parameters.	Provides fairly accurate estimates of base table and NUSI cardinality if the following conditions are met: • Table is large • Distribution of values is not skewed • Data is not taken from an atypical sample Other standard statistical parameters are less likely to be as accurate.

a. If the columns are not indexed, then the rows are organized randomly, so Teradata just scans the first *n* percent it finds, where the value of *n* is determined by the relative presence or absence of skew in the data.

If the columns are indexed, then more sophisticated sampling is performed to take advantage of the hash-sequenced row ordering.

A Note on Recollecting Statistics

When you recollect statistics on a column or index, the same method of collection, either full-table scan or sampling, is used to recollect the data as was used in the initial collection.

Random AMP Sampling

Introduction

When there are no statistics available to quantify the demographics of a column or index, then the Optimizer selects a single AMP to sample for its statistics using an algorithm based on the number of the current session. By inference, these numbers are then assumed to represent the global statistics for the column or index.

Assumptions Underlying the Method

The fundamental, closely related, assumptions underlying random AMP sampling are the following.

- The sampled table has enough rows that a snapshot of any randomly selected AMP in the system accurately characterizes its demographics for the entire table.
- The sampled table is distributed evenly, without skew.

Because these assumptions are usually valid, the statistics estimated using dynamic AMP sampling are generally accurate.

How Cardinality Is Estimated from a Random AMP Sample

The process used to gather statistics using a random AMP sample is described in the following table.

Stage	Process
1	Randomly select an AMP to be used to gather a statistical sample to represent an estimation of population demographics.
2	Read the Master Index to locate the cylinders containing data for the desired table, column, or index.
3	Count the number of cylinders that contain the desired data.
4	Randomly select one of those cylinders and read its Cylinder Index to locate the data blocks containing data for the desired table, column, or index.
5	Count the number of data blocks in the cylinder that contain the desired data.
6	Randomly select one of those data blocks and count the number of rows it contains.
7	Compute an estimate of the cardinality of the table using the following equation. $\text{Cardinality} = \sum \text{Rows in block} + \sum \text{Blocks in cylinder} + \sum \text{Cylinders with data} + \sum \text{AMPs in system}$

Table cardinality and, for indexes, the number of distinct values are the only statistics estimated by a random AMP sample:[6] no interval histograms (see "Interval Histograms" on page 2-25) are created as a result of random AMP sampling. The only other parameters available to the Optimizer after it collects a random AMP statistical sample is a set of default heuristics used to estimate the selectivity of various predicates[7].

As a result, the potential for error introduced by random AMP sampling from a small table (too few rows per AMP to provide an accurate measure) or a skewed value set is much higher than the estimates gained by the full-table statistics gathered using the Optimizer form of the COLLECT STATISTICS statement (see "List of the Statistics Collected and Computed" on page 2-44 for a list of those statistics).

Because the Optimizer understands that the information it collects using a random AMP sample is less reliable, it is less aggressive in its pursuit of optimal access and join strategies, meaning that in most cases, the resulting query plan is more costly than a plan developed from the more extensive, accurate statistics maintained in the interval histograms.

---

6 In the case of a NUSI, the cardinality estimate is for the number of index rows that correspond to the number of distinct values in the index.

7 For example, the following heuristics are used to evaluate simple WHERE clause predicates:

```
column_name = constant                                  10% selectivity
column_name > constant_1 AND column_name < constant_2 20% selectivity
column_name > constant_1                                33% selectivity
```

There are similar heuristics for estimating the selectivity of these conditions when connected in different ways by logical AND and OR operators.

Relative Accuracy of Residual Statistics Versus Random AMP Sample Statistics

Introduction

The relative accuracy of residual[8] statistics with respect to a random AMP sample depends on several factors.

For example, if the demographics for a column or index do not change, then residual statistics are a good representation of current demographics. It is possible for the cardinality of a table to grow by more than 10 percent, but for the relative proportion of its rows with particular values not to change. As a result, even a change in the table demographics of this magnitude might not affect the query plan generated by the Optimizer. When this is true, residual statistics could still be more accurate than a newly collected random AMP sample.

On the other hand, if the column in question is date-oriented, then rows added to the table with new dates cannot be accounted for by residual statistics, so the statistics collected with a random AMP sample are definitely more accurate than residual statistics.

Some Comparison Scenarios

Consider a more detailed scenario. Suppose the Optimizer evaluates statistics to determine if NUSIs are to be used in the query plan. The standard evaluation criterion for determining the usefulness of a NUSI in a query plan is that the number of rows that qualify per data block should be less than 1.0.

Assume an average data block size of 50 Kbytes and an average row size of 50 bytes. This produces an average of 1 000 rows per data block. Suppose the number of rows for a particular NUSI is 1 in 2 000, or one row in every 2 data blocks. The Optimizer determines that using the NUSI will save reading data blocks, so employing it in the query plan would result in fewer I/Os than doing a full-table scan.

Now, assume the table grows by 10 percent. The number of qualifying rows is now 1 in every 2 200 rows (a 10 percent increase in the cardinality of the table). For this particular case, the number of rows per data block is still less than 1.0, so the Optimizer does not need new statistics to produce a good query plan.

8 "Residual" meaning existing statistics that were not collected recently, implying the likelihood they no longer provide an accurate picture of column and index data.

On the other hand, consider a join scenario in which the Optimizer needs to know how many rows will qualify for spool. This can be critical, especially if the original estimate is near the cusp of the crossover point where a 10 percent increase in the number of rows makes the Optimizer change its selection from one join plan to another.

In this case, working with residual statistics could mean that the Optimizer chooses an old plan instead of a new, faster plan. Or, worse still, the residual statistics could produce a new join plan that is much slower that the previous plan.

Effects of Relative Number of Distinct Values on the Usefulness of Residual Statistics

Tables with a large[9] number of distinct values might see no significant negative performance effects with residual statistics, with the exception of new column values such as new dates. Tables with several orders of magnitude fewer distinct values can have a larger number of values nearer the crossover point that would affect their position within the statistics partitions if new statistics were collected.

Effects of Cardinality on the Usefulness of Residual Statistics

Base table cardinality is another critical determinant for the usefulness of residual versus randomly sampled or recollected statistics. The smaller the cardinality, the more sensitive the Optimizer is to inaccurate statistics. Consider the following scenario as an example.

Suppose a table has only 100 000 rows with 1 000 distinct values, an average of 100 rows per distinct value. The following list illustrates some of the potential issues the Optimizer faces when dealing with small tables, particularly with respect to generating query plans based on inaccurate statistics:

- For a NUSI question, there are too few rows per distinct value for the Optimizer to use the NUSI in its query plan.
- For estimating spool cardinalities for join processing, the variance from 100 is very important:
  - If some distinct values have only a few rows, the Optimizer could decide to duplicate this small spool and then do a product join.
  - If the number of rows per distinct value is 200, rather than 2, then a product join could easily be the wrong join plan.
  - If the spool cardinality is estimated to be a much larger value like 1 000 000 with 1 000 distinct values, then the average number of rows per distinct value is 1 000 and for the majority of cases, the Optimizer would probably pick a product join.

9 “Large” in this case is defined as greater than 100 000.

The problem with this sort of exercise is that it cannot possibly account for all the variables the Optimizer uses to evaluate and generate a query plan, so you really cannot know *a priori* how good a query plan it will create with less accurate statistics.

Comparing the Relative Accuracies of Various Methods of Collecting Statistics

Introduction

The important thing to understand when considering the comparative *a priori* likelihoods of the accuracy of statistics collected by the various available methods is the consistently higher accuracy of population statistics over all forms of sampled statistics.[10] Although this is undeniable, it is not possible to know *a priori* whether it is necessary to collect a full set of new statistics in order to ensure that the Optimizer produces the best query plans.

Ranking the Relative Accuracy Probabilities of the Various Methods of Collecting Statistics

The best comparative estimates of the relative accuracies of the statistics collected by the various methods are described by the following ranked list:

1. Random AMP samples are better than residual statistics in the majority of cases.
2. Random all-AMPs samples are better than random AMP samples in most cases.
3. Full-table population statistics are usually better than any sampled statistics.

The following table provides some details to support these rankings. Each successively higher rank represents an increase in the accuracy of the statistics collected and a higher likelihood that the Optimizer will produce a better query plan because it has more accurate information to use for its estimates.

10 It is always possible that statistics collected using a method with a lesser probability of accuracy will be *as* good as those collected at any given higher level of probable accuracy, but they will never be *more* accurate.

Collection Method	Relative Elapsed Time to Collect	Accuracy Rank (Higher Number = Higher Accuracy)	Comments
Use residual statistics	None.	1	• Impossible to know *a priori* if residual statistics will produce a good query plan. At worst, can produce a very poor query plan. • Because statistics exist, Optimizer does not collect fresh statistics using a random AMP sample.
Random AMP sample	Almost none.	2	• Data is collected from a subset of the rows on a single AMP and might not be representative of full table demographics. • Collects statistics for cardinality and number of distinct index values only. • Because the sample size is small, there is a high bias in the statistics. • Accuracy is very sensitive to skew. • Provides fairly accurate estimates of base table and NUSI cardinality if table is large, distribution of values is not skewed, and data is not taken from an atypical sample. Other standard statistical parameters are less likely to be as accurate. • Does not enable semantic query optimization.
All-AMPs sample	Approximately 5 percent of time to perform a full-table scan[a].	3	• Data is collected from a system-determined subset of the rows on all AMPs. • Collects identical statistics to full-table scan, including interval histogram creation. • Percentage of sampled rows is small. • Percentage of sampled rows is increased dynamically to enhance the accuracy of collected statistics if skew is detected in the samples. • Does not enable semantic query optimization.
Full-table scan	Approximately 195 percent of the time to perform sampled statistics.	4	• Data is collected from all rows on all AMPs so there is no sample bias. • Collects full statistics and creates interval histograms. • Skew is accounted for using high-biased intervals in the statistical histogram for a column or index. • Up to 100 histogram intervals are used, depending on the data. • Required to initiate rule production for semantic query optimization.

a. When the data is skewed, this percentage is larger, depending on how much the system dynamically increases its sample size.

When Should Statistics Be Collected Or Recollected?

Introduction

The choice of collecting full-table statistics, some form of sampled statistics, or no statistics is yours to make, as long as you understand that the method that always provides the best table statistics over the long run is collecting full-table statistics.

Teradata recommends *always* collecting full-table statistics on a regular basis. Just how frequently statistics should be collected is contingent on several factors, and depending of the various qualitative and quantitative changes in your column and index demographics, residual statistics can be just as good as freshly collected statistics (see "Relative Accuracy of Residual Statistics Versus Random AMP Sample Statistics" on page 2-37 for a description of some of the factors to consider when making this evaluation).

Of course, it is up to you to determine what methods work best in the various situations encountered in your production environment. You might decide that a single approach is not good enough for all your different tables. All-AMPs sampled statistics might provide sufficient accuracy for your very large tables, enabling you to avoid expending the system resources that collecting full-table statistics might consume.

Keep in mind that the operational definition of good statistics is those statistics that produce an optimal query plan. How and when you collect statistics on your table columns and indexes depends on your definition of an optimal query plan.

Teradata Statistics Wizard

The Teradata Statistics Wizard client utility provides an objective method of determining when fresh statistics should be collected on a table, column, or index based on a set of user-provided SQL query workloads. The Statistics Wizard also advises you on which tables, columns, and indexes statistics should be collected as well as providing a means of verifying the enhanced efficiency of queries run against the workloads.

See *Teradata Statistics Wizard User Guide* for additional information about the uses and capabilities of this utility.

Policies for Collecting Statistics

The following table lists some suggested policies for collecting statistics. Each policy is rated as recommended or strongly recommended.

Policy	Required or Recommended
Recollect all statistics when you upgrade to a new Teradata release.	Strongly recommended.
Recollect statistics whenever the population of a table changes by 10 percent or more. For high volumes of very nonunique values such as dates or timestamps, you should consider recollecting statistics when the population changes by as little as 7 percent.	Strongly recommended.
Collect statistics on newly created, empty tables to define the table columns and indexes as well as to create the synoptic data structures for subsequent collection of statistics.	Recommended.
Recollect statistics whenever the number of rows per distinct value is less than 100.	Recommended.

The following table provides more specific recommendations for collecting statistics. You should consider this set of recommendations to be minimal and essential.

FOR ...	You should collect statistics on ...
all tables	all columns used in join conditions.
large tables	all NUPIs.
small tables[a]	the primary index.

a. In this context, a small table is defined as a table whose cardinality is less than 5 times the number of AMPs in the system.

To ensure the best query plans, you should consider collecting statistics on the following, more general set of table columns:

- All indexes.
- High-access join columns.
- Columns referenced in WHERE clause predicates, particularly if those columns contain skewed data.

How the Optimizer Uses Statistical Profiles

Introduction

The COLLECT STATISTICS (Optimizer Form) statement gathers demographics about a specified column or index. It then uses this information to compute a statistical synopsis or profile of that column or multicolumn index to summarize its characteristics in a form that is useful for the Optimizer when it generates its access and join plans.

Sometimes it seems like every page you read in the Teradata manual set instructs you to collect statistics frequently. This topic explains why you should do so. The topic first describes some of the basic statistics calculated and then explains, at a very high level, how the Optimizer uses the computed statistical profile of your database.

List of the Statistics Collected and Computed

The following set of variables represents the essential set of column statistics that are computed each time you perform the Optimizer form of the COLLECT STATISTICS statement.

Different statistics are stored for a column depending on whether its values are highly skewed or not.

IF the distribution of column values is ...	THEN its statistics are stored in this type of interval histogram ...
Non-skewed	Equal-height
Highly skewed	High-biased

Note the use of the term *estimate* in the descriptions. The values for a column interval are exact only at the moment their demographics are collected. The statistics stored for a column are, at best, only a snapshot view of its value distributions.

Term	Description
Equal-Height Interval Statistics	
Number of rows	An estimate of the cardinality of the table.
Number of distinct values	An estimate of the number of unique values for the column.
Maximum number of rows per value	An estimate of the maximum number of rows having the particular value for the column.
Typical number of rows per value	An estimate of the most common number of rows having the particular value for the column.
Number of nulls	An estimate of the number of nulls for the column.
Maximum value for the interval	An estimate of the largest value for the interval.
Minimum value for the interval	An estimate of the smallest value for the interval.
Number of intervals	The number of intervals in the histogram containing the column statistics.
Modal value for the interval	An estimate of the most frequently occurring value or values in the interval.
Number of rows having the modal value	An estimate of the distinct number of rows in the interval having its modal value.
Number of rows not having the modal value	An estimate of the skewness of the distribution of column values within the interval.
High-Biased Interval Statistics	
Modal value	The value for the loner. If two values are stored in the interval, then Mode is the value for the first loner. No more than two modal values can be stored per interval for a skewed distribution.
Maximum value	The value for the second loner in a two-loner interval.

You can view the statistics for a column using the Statistics Collection program of the Teradata Manager product.

Examples of How Cardinality Estimates Are Made for Simple Queries Using Interval Histograms

Introduction

The following set of examples uses a small subset of the interval histograms for the column or index values evaluated. The examples are oversimplified to emphasize basic aspects of the process.

The data demographics of the tables examined in these examples are such that their data is stored in equal-height intervals. This is done to simplify the examples.

Interval Histogram Data

Recall that the principal information stored in a standard equal-height interval is as follows.

- Maximum value in the interval.
- Most frequent value in the interval.
- Instances of the most frequent value in the interval.
- Number of distinct values in the interval.
- Number of values in the interval *not* equal to the most frequent value.

  This number is constant across intervals when equal-height intervals are used. Its value is calculated as follows

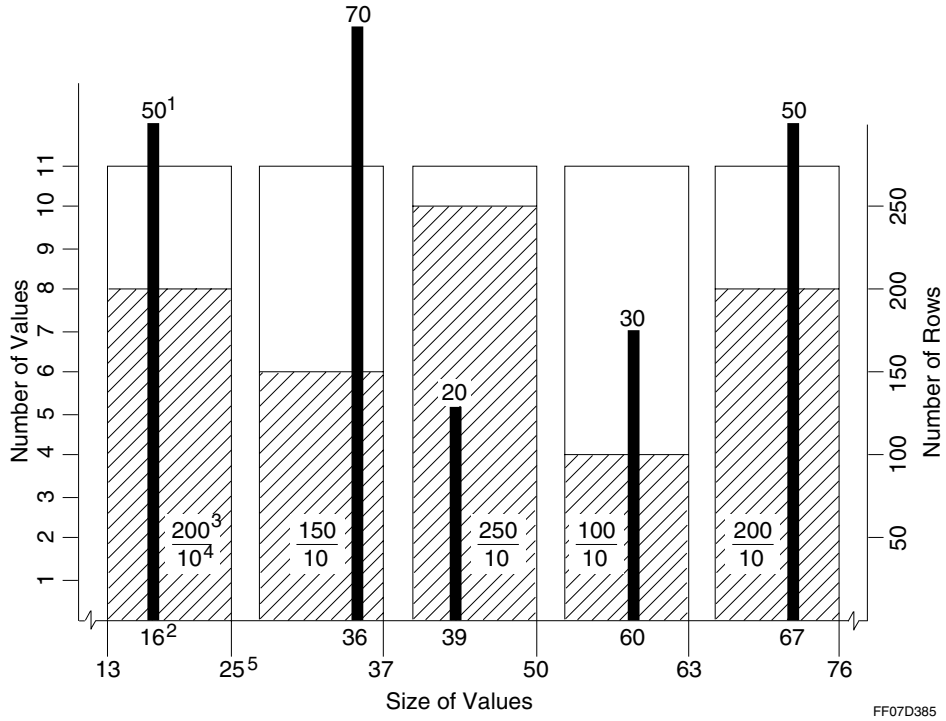
  ```
  Values not equal to most frequent  =  Number of values in interval - 1
  ```

  where 1 is the number of distinct values that occur most frequently in the interval.
- Number of rows containing the most frequent value in the interval.
- Number of rows *not* containing the most frequent value in the interval.

The data used for the examples is tabulated below. Notice that the total number of distinct values (the shaded cells of the table) is the same for all five intervals. This is definitive for equal-height interval histograms.

Variable	Interval Number				
	1	2	3	4	5
Instances of the most frequent value in the interval	16	36	39	60	67
Number of values not equal to the most frequent value in the interval	10	10	10	10	10
Number of distinct values in the interval	11	11	11	11	11
Number of rows in the interval having the most frequent value	50	70	20	30	50
Number of rows in the interval *not* having the most frequent value	200	150	250	100	200
Number of rows in the interval	250	220	270	130	250
Maximum value in the interval	25	37	50	63	76

The following picture illustrates these numbers graphically. The footnotes are defined in the table following the illustration.



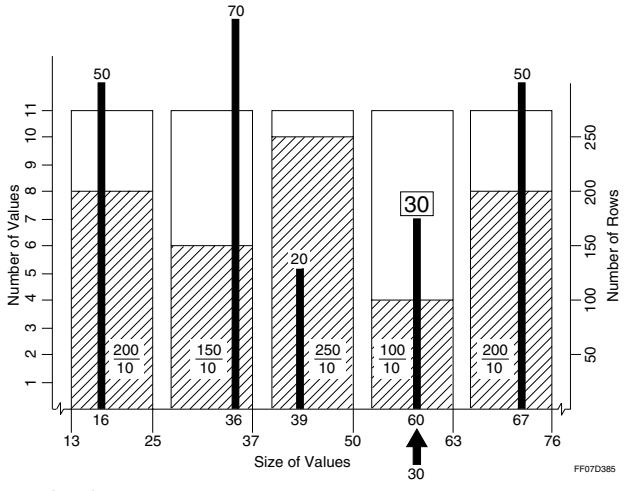
Note	Description
1	Number of rows that have the most frequent value in the interval as the value for the column or index.
2	Most frequent value for the column or index in the interval.
3	Number of rows that do not have the most frequent value in the interval as the value for the column or index.
4	Number of values in the interval that are not equal to the most frequent value for the column or index.
5	Maximum value in the interval.

The shaded portion of the interval represents the number of rows, while the height of the interval represents the number of distinct values it contains, which is constant for any equal-height interval.

Notice that the width of each of these intervals is 12 units of value. This is not a characteristic of equal-height histograms, which tend to vary widely in their width. The interval widths were chosen to make the intervals more tractable for demonstrating the concepts of how interval histograms are used by the Optimizer. The interval heights are, by definition, constant. In this set of examples, the constant interval height is 11 values. No examples using high-biased intervals are presented in this example set.

Example 1

The cardinality of the response set for this simple query need not be estimated because if the statistics on the column set are current, then the Optimizer knows exactly what the cardinality is. There are 30 instances of the value 60 in *table*. This value is known because it is the count of the number of rows in the interval having the most frequent value, 60, for the column named in *condition*.



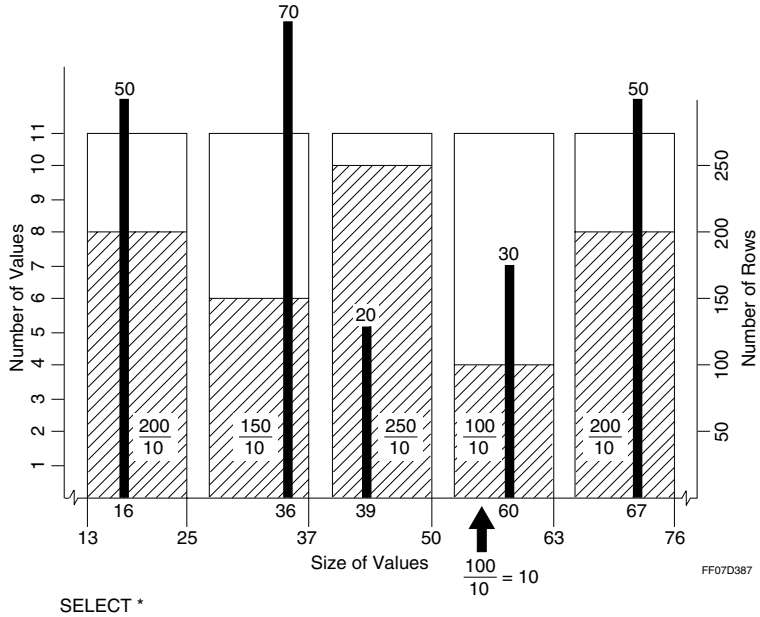

```
SELECT *
FROM table
WHERE condition = 60;

[ 30 rows ]
```

Example 2

This example illustrates a simple equality condition.

If there are any rows that satisfy the condition where the value for the named column is 55, they are found in the range between 51, the lower bound for the interval, and its upper bound of 63.




```
SELECT *
FROM table
WHERE condition = 55;

[ 10 rows ]
```

The Optimizer knows the following things about this condition.

- It is an equality.
- The equality ranges over a single interval.
- The most frequent value in the interval does not qualify its rows for inclusion in the response set.

The Optimizer has to estimate the cardinality of the response set for this example because unlike "Example 1" on page 2-48, there are no exact statistics to describe it. The heuristic used to estimate the response set cardinality is to divide the number of rows in the interval not having the most frequent value by the number of values in the interval not equal to the most frequent value.

$$\text{Estimated cardinality of the response set} = \frac{\text{Number of rows not having the most frequent value}}{\text{Number of values not the most frequent value}}$$

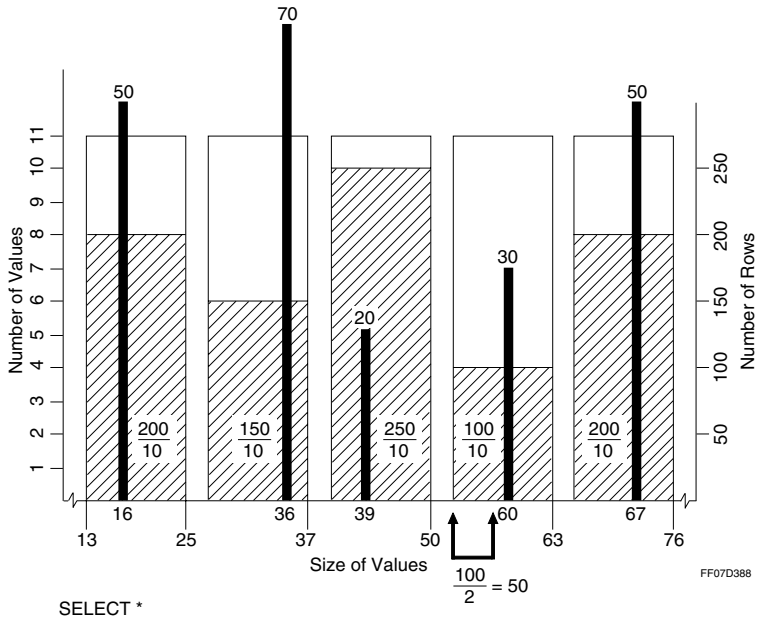
If the statistics are current, then there are 100 rows in this interval that do not have the value 30 for the column specified by *condition* and there are 10 values that are not equal to the most frequent value in the interval. The Optimizer divides the number of rows in the interval that do not have the value 30 by the number of values in the interval not equal to the maximum value, which is 10.

$$\text{Estimated cardinality of the response set} = \frac{100}{10} = 10 \text{ rows}$$

The cardinality of the response set is estimated to be 10 rows.

Example 3

This example specifies a range condition. Statistical histogram methods are a particularly powerful means for estimating the cardinalities of range query response sets.




```
SELECT *
FROM table
WHERE condition BETWEEN 51 AND 57;

[ 50 rows ]
```

The Optimizer knows that the quantity of rows having the *condition* column value between 51 and 57 must be found in the single interval bounded by the values 51 and 63.

The keyword BETWEEN is SQL shorthand for *value* ≥ *lower_limit* AND ≤ *upper_limit*, so it signifies an inequality condition.

The Optimizer knows the following things about this condition.

- It is an inequality.
- The inequality ranges over a single interval.
- The most frequent value in the interval does not qualify its rows for inclusion in the response set.

The Optimizer has to estimate the cardinality of the response set for this example because there are no exact statistics to describe it. The heuristic used to estimate the response set cardinality is to divide the number of rows not having the most frequent value in the interval in half.

$$\text{Estimated cardinality of the response set} = \frac{\text{Number of rows not having the most frequent value}}{2}$$

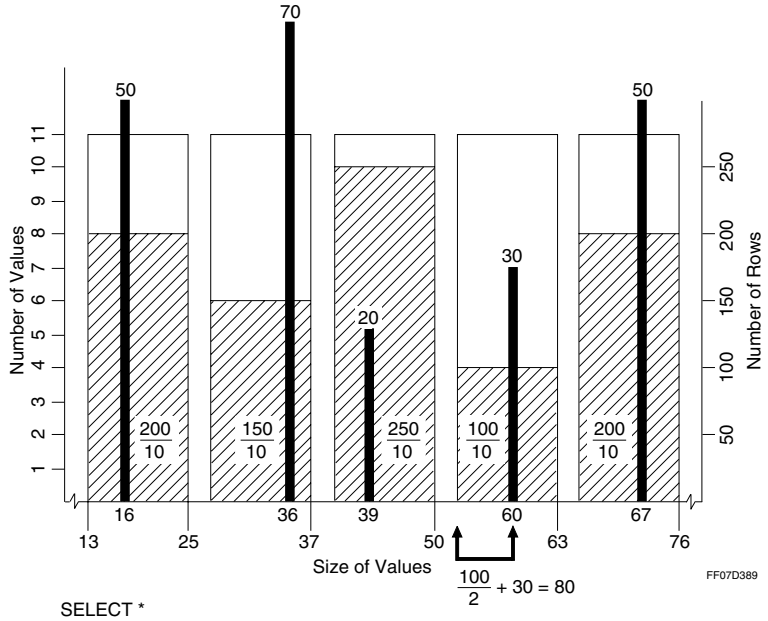
Assuming current statistics, there are 100 rows in the interval that do not have the value 30 for the column specified by *condition*, so the Optimizer divides the number of rows not having the value 60, which is 100, in half.

$$\text{Estimated cardinality of the response set} = \frac{100}{2} = 50 \text{ rows}$$

The cardinality of the response set is estimated to be 50 rows.

Example 4

This example is slightly more sophisticated than "Example 3" on page 2-50 because it specifies a range predicate that includes the most frequently found value in the interval.




```
SELECT *
FROM table
WHERE condition BETWEEN 51 AND 60;

[ 80 rows ]
```

The Optimizer knows that the quantity of rows having their *condition* column value between 51 and 60 must be found in the single interval bounded by the values 51 and 63.

The Optimizer knows the following things about this condition.

- The condition is an inequality.
- The inequality ranges over a single interval.
- The most frequent value in the interval qualifies its rows for inclusion in the response set.

The Optimizer has to estimate the cardinality of the response set for this example because there are only partial exact statistics to describe it. The heuristic used to estimate the response set cardinality is to divide the number of rows not having the most frequent value in the interval in half and then add that number to the number of rows in the interval having the most frequent value.

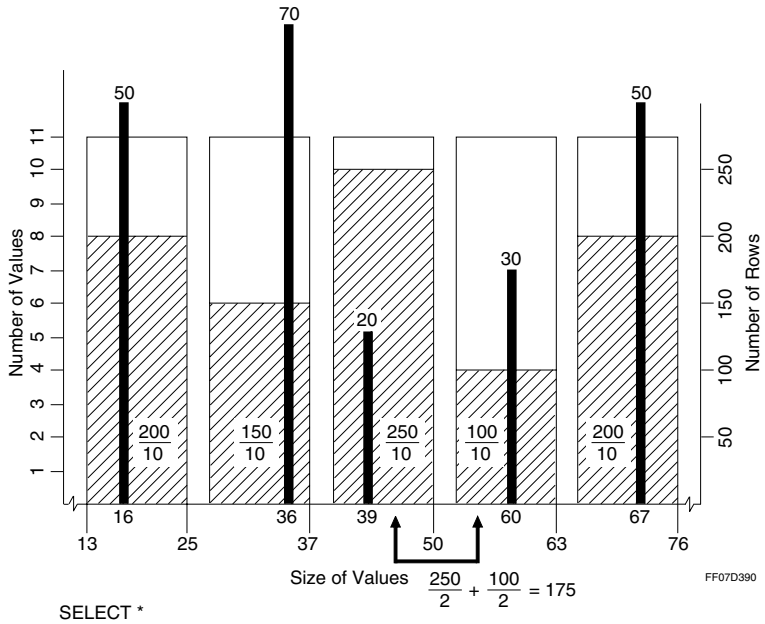
Because this quantity is known exactly if the statistics are current, the estimated cardinality of the response set should be more accurate than in the previous example.

There are 100 rows in the interval that do not have a value 30 for the column specified by *condition*, so the Optimizer divides the number of rows not having the value 60, which is 100, in half and then adds the 30 rows known to exist where *condition* = 60.

$$\text{Estimated cardinality of the response set} = \frac{100}{2} + 30 = 80 \text{ rows}$$

Example 5

This example is a slightly more complicated range query than "Example 4" on page 2-51 because the response set spans two histogram intervals.




```
SELECT *
FROM table
WHERE condition BETWEEN 45 AND 55;

[ 175 rows ]
```

The Optimizer knows that the quantity of rows having the *condition* column value between 45 and 55 must be found in the two adjacent intervals bounded by the values 38 and 63.

The Optimizer knows the following things about this condition.

- It is an inequality.
- The inequality ranges over two intervals.
- The most frequent value in neither interval qualifies its rows for inclusion in the response set.

The Optimizer has to estimate the cardinality of the response set for this example because there are no exact statistics to describe it. The heuristic is to estimate the response set cardinalities for each interval individually by dividing the number of rows not having the most frequent value in the interval by half and then summing those two cardinalities.

There are 250 rows in the lower interval that do not have a value of 20 for the column specified by *condition*, so the Optimizer divides the number of rows not having the value 20, which is 250, in half, producing an estimate of 125 rows satisfying the condition for that interval.

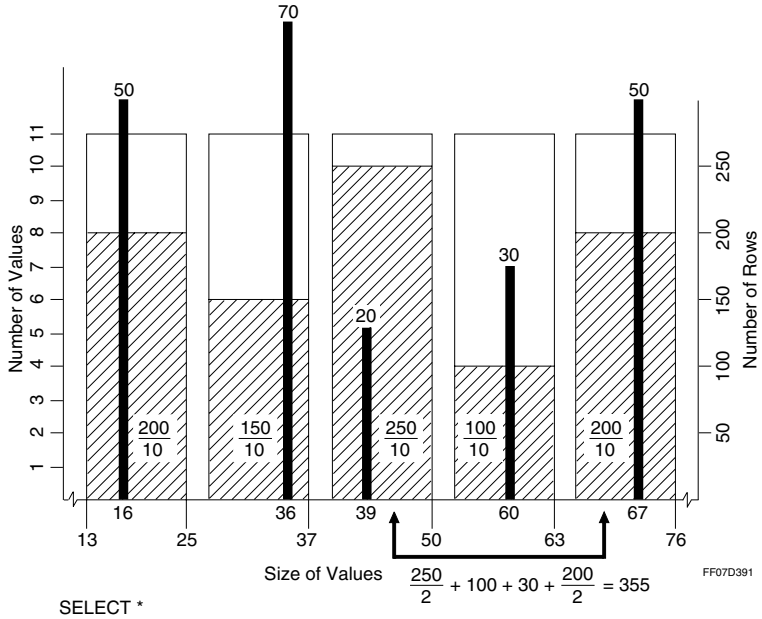
There are 100 rows in the higher interval that do not have a value 30 for the column specified by *condition*, so the Optimizer divides the number of rows not having the value 60, which is 100, in half, producing an estimate of 50 rows satisfying the condition for that interval.

The total estimate is obtained by adding the estimates for each of the two intervals.

$$\text{Estimated cardinality of the response set} = \frac{250}{2} + \frac{100}{2} = 175 \text{ rows}$$

Example 6

The final example specifies a range query that spans three intervals and includes the most frequent value in the middle interval.




```
SELECT *
FROM table
WHERE condition BETWEEN 45 AND 65;

[ 355 rows ]
```

The Optimizer knows that the quantity of rows having the *condition* column value between 45 and 50 must be found in the interval bounded by the values 38 and 50.

The Optimizer also knows that all rows in the next higher interval, which is bounded by the values 51 and 63, are included in the response set. The estimate is computing by summing the number of rows with values that are not the most frequently found within the interval with the number of rows having the most frequent value, or 100 + 30. If the statistics are current, this number is exact.

The number of rows having *condition* column values in the range 64 through 65 must be estimated by using half the number of values that are not the most frequently found within the interval. The estimate is half of 200, or 100 rows.

The Optimizer knows the following things about this condition.

- It is an inequality.
- The inequality ranges over three intervals.
- The most frequent values in the lowest and highest intervals do not qualify their rows for inclusion in the response set.
- The most frequent value in the middle interval does qualify its rows for inclusion in the response set.

The Optimizer has to estimate the cardinality of the response set for this example because there are only partial exact statistics to describe it. The heuristic used to estimate the response set cardinality is the following.

Stage	Process
1	Estimate the cardinality of the response set returned from the first interval by dividing the number of rows not having the most frequent value in half.
2	Read the cardinality of the rows having the most frequent value from the second interval.
3	Read the cardinality of the rows not having the most frequent value from the second interval.
4	Estimate the cardinality of the response set returned from the third interval by dividing the number of rows not having the most frequent value in half.
5	Add the numbers derived in steps 1 through 4 to provide an overall estimate of the cardinality of the response set.

The total estimate is obtained by adding the estimates for each of the three intervals: 125 + 130 + 100, or 355 rows.

$$\text{Estimated cardinality of the response set} = \frac{250}{2} + 100 + 30 + \frac{200}{2} = 355 \text{ rows}$$

Cost Optimization

Introduction

When processing requests, the Optimizer considers all of the following factors and attempts to choose the least costly access method and, when appropriate, the least costly join path.

- Row selection criteria
- Index references
- Available statistics on indexes and columns

The column and index demographics gathered by the COLLECT STATISTICS (Optimizer Form) statement and the statistics it computes permit the Optimizer to better identify skew in tables so it has more finely-tuned knowledge for generating plans using its cost estimator functions.

The number of rows selected is based on the number of rows in the table and the selectivity of the constraint. If there is no constraint, all rows are selected.

Example

Consider the following table fragment:

Table_X

State	SerialNo
28	12345
28	23456
51	12345
51	23456

The following query resulted in a full file scan:

```
SELECT *
FROM Table_X
WHERE State IN (28, 51)
AND SerialNo IN (12345, 23456);
```

Now modify the query as follows:

```
SELECT *
FROM Table_X
WHERE (State = 28 AND SerialNo = 12345)
OR (State = 51 AND SerialNo = 23456)
OR (State = 28 AND SerialNo = 23456)
OR (State = 51 AND SerialNo = 12345);
```

The semantically identical modified query resulted in four primary index accesses, which is a marked performance enhancement over the full table scan used to solve the first query.

Path Selection

Expressions involving both AND and OR operators can be expressed in either of the following logically equivalent forms.

- `(A OR B) AND (C OR D)`
- `(A AND C) OR (A AND D) OR (B AND C) OR (B AND D)`

The first form is known as conjunctive normal form, or CNF. In this form, operand pairs are ORed within parentheses and ANDed between parenthetical groups. The advantage of CNF is that as soon as any individual condition in the expression evaluates to FALSE, the entire expression evaluates to FALSE and can be eliminated from further consideration by the Optimizer.

The second is called disjunctive normal form, or DNF. Different access paths might be selected based on these two forms; depending on the expression, one form may be better than the other.

If A, B, C and D refers to columns of a single table, the Optimizer generates the access path based on the form specified in the query; there is no attempt to convert from one form to another to find the best path. On the other hand, if A, B, C or D specifies a join condition, the second form is converted to the first.

Example 1

Consider the following expression:

```
(NUSI = A OR NUSI = B) AND (X = 3 OR X = 4)
```

In this case, CNF is more performant because the access path consists of two NUSI SELECTs with values of A and B. The condition (X=3 OR X=4) is then applied as a residual condition. If DNF had been used, then four NUSI SELECTs would be required.

Example 2

In the following expression the collection of (NUSI_A, NUSI_B) comprise a NUSI:

```
(NUSI_A = 1 OR NUSI_A = 2) AND (NUSI_B = 3 OR NUSI_B = 4)
```

In this case, DNF is better because the access path consists of four NUSI SELECTs, whereas the access path using CNF would require a full table scan.

Example 3

Consider an expression that involves a single field comparison using IN, such as the following.

```
Field IN (Value_1, Value_2, …)
```

Internally, that expression is converted to CNF.

```
Field = Value_1 OR Field = Value_2 OR …
```

Therefore, the same access path is generated for either form.

Example 4

Assume an expression involves a multiple-field comparison using IN, such as in the following statement:

```
    Field_1 IN (Value_1, Value_2, …)
AND Field_2 IN (Value_3, …)
```

The Optimizer converts this syntax to conjunctive normal form:

```
    (Field_1 = Value_1 OR Field_1 = Value_2 OR …)
AND (Field_2 = Value_3 OR …)
```

Example 5

Note how the converted form, conjunctive normal form (or CNF) differs from the second form, which is formulated in disjunctive normal form (DNF) as shown in the following example:

```
   (Field_1 = Value_1 AND Field_2 = Value_3)
OR (Field_1 = Value_2 AND Field_2 = Value_4)
OR …
```

The point is that semantically equivalent queries, when formulated with different syntax, often produce different access plans.

Optimizer Cost Functions

The Optimizer has a set of internal functions that return various cost factors used to compute optimal access and join plans. The cost factors returned to the Optimizer by these functions include the calculated number of blocks in a relation, the cost of various disk operations, the number of AMPs used for an operation, the number of AMPs configured per node, the cost of sorting a spool file, the cost of duplicating rows across AMPs, hashing costs, and so on.

Statistics and Cost Estimation

The following properties apply to Optimizer cost estimates when statistics are available:

- The Optimizer evaluates access selectivity based on demographic information.

  If statistics have recently been collected on the primary index for a table, then an accurate estimate of the total number of rows is available to the query Optimizer.
- The Optimizer uses primary index statistics to estimate the total number of rows in a table.

  Therefore, you should keep those statistics current to ensure optimum access and join planning. Collect new statistics on primary indexes when more than 10% of the total number of rows are added or deleted.

The following properties apply to Optimizer cost estimates when no statistics are available:

The Optimizer estimates the total number of rows in a table based on information sampled from a randomly selected AMP (see “Random AMP Sampling” on page 2-35). For the following reasons, this might not be an accurate estimate of table statistics.

- The table is relatively small and not partitioned evenly.
- The nonunique primary index selected for a table causes its rows to be distributed unevenly across the AMPs.

When either of these properties is true, then there might be AMPs on which the number of rows is highly unrepresentative of the average population of rows in the other AMPs.

If such an unrepresentative AMP is selected for the row estimate, the Optimizer might generate a bad estimate, and thus might not choose a plan (join, access path, and so on) that would execute the request most efficiently.

Outdated statistics often cause the Optimizer to produce a worse join plan than one based on estimated information from a randomly sampled AMP.

The only difference is that a plan based on outdated statistics is consistently bad, while a plan based on a random AMP sample has a likelihood of providing a reasonably accurate statistical estimate.

Ensuring Indexed Access

To guarantee that an index is used in processing a request, specify a WHERE constraint on a value from an indexed column.

If multiple nonunique secondary indexes (NUSIs) apply to such a WHERE constraint, and if the subject table is very large, then bit mapping provides the most efficient retrieval. For smaller tables, the Optimizer selects the index estimated to have the highest selectivity (fewest rows per index value).

If statistics are not available for a NUSI column, then the Optimizer assumes that indexed values are distributed evenly. The Optimizer estimates the number of rows per indexed value by selecting an AMP at random and dividing the total number of rows from the subject table on that AMP by the number of distinct indexed values on the AMP. When distribution of index values is uneven, such an estimate can be unrealistic and result in poor access performance.

Guidelines for Indexed Access

Always use the EXPLAIN modifier or the Visual EXPLAIN utility to determine the plan the Optimizer will generate to perform a query.

The Optimizer follows these guidelines for indexed access to table columns:

- UPIs are used for fastest access to table data.
- USIs are used only to process requests that employ equality constraints.
- The best performance in joins is achieved when the following is possible:
  - Matching UPI values in one table with unique index values, either UPI or USI, in another table.
  - Using *only* a primary index when satisfying an equality or IN condition for a join.
- NUPIs are used for single-AMP row selection or join processing to avoid sorting and redistribution of rows.

- When statistics are not available for a table, the estimated cost of using an index is based on information from a single AMP. This estimate assumes an even distribution of index values. An uneven distribution impacts performance negatively.
- Composite indexes are used only to process requests that employ equality constraints for *all* fields that comprise the index.

  Note that you can define an index on a single column that is also part of a multicolumn index on the same table.
- Bit mapping is used only when equality or range constraints involving multiple nonunique secondary indexes are applied to very large tables.
- Use either a nonuniquely indexed column or a USI column in a nested join to avoid redistributing and sorting a large table.

  For example, consider the following condition:

  ```
  table_1.field_1 = 1 AND table_1.field_2 = table_2.NUSI
  ```

  Without using a nested join, where table_1 is duplicated and joined with table_1.NUSI, both table_1 and table_2 might have to be sorted and redistributed before a merge join. This join is not performant when table_2 is large.

Environmental Cost Factors

Introduction

Teradata makes a substantial store of system configuration and performance data, referred to as environmental cost factors, available to the Optimizer so it can tune its plans appropriately for each individual system. This environmental cost data is what enables the Teradata optimizer to operate fully and natively in parallel mode.

Types of Cost Factors

There are two basic types of environmental cost factors, as defined in the following table.

Cost Factor Type	Definition
External cost parameters	External cost parameters are culled from various system tables and files at system startup. They consist of various hardware and configuration details about the system.
Performance constants	Performance constants specify the data transfer rates for each type of storage medium and network interconnection supported by Teradata. The values of these constants can be statically modified to reflect new hardware configurations when necessary.

External Cost Parameters

External cost parameters are various families of weights and measures, including the parameters listed in the following table.

External Cost Parameter Group	Definition
Optimizer weights and scales	Weightings of CPU, disk, and network contributions to optimizing a request plus scaling factors to normalize disk array and instruction path length contributions. Unitless decimal weighting factors.
CPU cost variables	CPU costs for accessing, sorting, or building rows using various methods.
Disk delay definitions	Elapsed times for various disk I/O operations. Measured in milliseconds.
Disk array throughput	Throughputs for disk array I/O operations. Measured in I/Os per second.
Network cost variables	Message distribution and duplication costs and overheads. Measured in milliseconds.
Optimizer environment variables	Hardware configuration information such as the number of CPUs and vprocs per node, maxima, minima, and average numbers of MIPS per CPU, nodes per clique, disk arrays per clique, and so on. Unitless decimal values.
Miscellaneous cost parameters	DBSControl information such as free space percentages, maximum number of parse tree segments, dictionary cache size, and so on. Measured in various units, depending on the individual cost parameter.
Performance constants	Various values used to assist the Optimizer to determine the best method of processing a given query. Measured in various units, depending on the individual performance constant.
PDE system control files	Lists various system control files used by PDE. Used to initialize the appropriate GDOs with the appropriate values at startup.
DBS control files	Lists various database control files found in the DBSControl record. Used to initialize the appropriate GDOs with their respective appropriate values at startup.
TPA subsystem startup initializations	Initializes or recalculates various Optimizer parameters, including the optimizer target table GDO used by the target level emulation software (see Chapter 6: "Target Level Emulation").

Optimizer Join Plans

Introduction

Selecting the optimum method to join tables is as critical to the performance of a query as the selection of table access methods. Selecting an optimal join method, or mode, is a separate issue from that of determining the optimal join order for a query. Join order optimization is described in "Evaluating Join Orders" on page 2-72.

There are many methods for joining tables, and the method ultimately chosen to make a join depends on multiple factors, including the comparison on which the join is made (vanilla equijoin or θ join, where θ is any valid nonequality condition), the estimated cardinality of the tables being joined, the configuration of the system on which the join is to be made, and so on.

This section of the chapter describes join processing at several levels including the important topic of join methods.

Join Processing Methods

Depending on the indexes defined for the tables involved and whether statistics are available for the indexes, the Optimizer processes a join using one of the following join algorithms.

- Product Join
- Hash Join
- Merge Join
- Nested Join (local and remote)
- Exclusion Join (merge and product)
- Inclusion Join (merge and product)
- RowID Join
- Self-Join
- Correlated Join
- Minus All Join

See Chapter 3: "Join Optimizations," for more information about each of these join modes.

Limit on the Number of Tables That Can Be Joined

Excluding self-joins, as many as 64 tables can be joined per query block.

Loosely defined, a query block is a unit for which the Optimizer attempts to build a join plan. The following list notes a few of the more frequently occurring query blocks.

- Uncorrelated subqueries
- Derived tables
- Complicated views
- Portions of UNION and INTERSECT operations

Each reference to a table, including those using correlation names, counts against the limit of 64 tables. This limit includes implicit joins such as the join of a hash or single-table join index to a base table to process a partial cover.

For example, consider a query with a single uncorrelated subquery. The subquery is limited to 64 tables and the outer query is limited to *63* tables, the 64th table for the outer query being the spooled result of the inner query that must be joined with it.

If the uncorrelated subquery were an outer query to an additional uncorrelated subquery, then the deepest subquery would be limited to referencing 64 tables, its outer query limited to 63 (63 plus the result of its inner query), and the parent outer query to 63 plus the result of *its* inner query.

In summary, while the number of tables, including intermediate spool relations, that can be joined is limited to 64 per query, the cumulative number of tables referenced in the course of optimizing the query can be considerably greater than 64.

There is no way to determine a priori how many query blocks will be created and processed by the Optimizer in the course of producing a join plan, but the factors listed here are all candidates to evaluate if your queries terminate because they exceed the 64 table join limit.

Join on the Same Domain

Always join tables on columns that are defined on the same domain.

If the join columns are not defined on the same domain, then the system must convert their values to a common data type before it can process the join. The conversion process is resource-intensive and thus a performance burden.

Ensure You Have Allocated Enough Parser Memory for Large Joins

If you plan to join more than 32 tables, use the DBS Control Utility to increase the value for the DBSControl Record MaxParseTreeSegs performance field to at least 200 to allocate additional Parser memory for code generation. The upper limit on MaxParseTreeSegs is 1 000.

MaxParseTreeSegs defines the number of 64 Kbyte tree memory segments the Parser allocates when parsing a Teradata SQL statement.

When NUSIs and USIs Are Used by the Optimizer

Nonunique secondary indexes (NUSIs) and unique secondary indexes (USIs) are used in a join operation only if the join plan used by the Optimizer calls for a nested join or a rowID join.

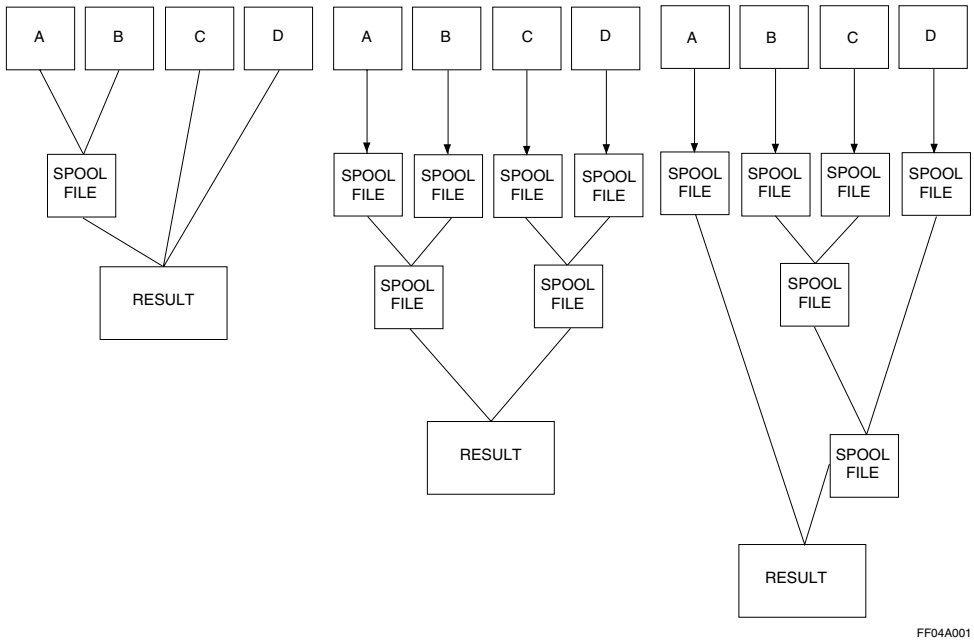
How the Optimizer Plans Multitable Joins

If multiple tables are involved in a join, the Optimizer reduces that join to a series of two-table joins and then attempts to determine the most cost effective join order.

For example, assume you submitted the following multitable query:

```
SELECT …
FROM A, B, C, D
WHERE …;
```

The Optimizer generates join plans like the three diagrammed in the following graphic. Because the Optimizer uses column statistics to choose the least costly join plan from the candidate set it generates, the plans it generates are extremely dependent on the accuracy of those numbers.



Column projection and row selection are done prior to doing the join. If selection criteria are not supplied, then all rows and columns participate in the join process.

Obviously, it is always advantageous to reduce the number of rows and columns that must be duplicated or redistributed; however, column projections are done only when it is more efficient to do so. If a join can be done *directly* with the base table, column projection is *not* done before the join.

Join Order Search Trees

Query optimizers use trees to build and analyze optimal join orders. The join search tree types used most frequently by commercial relational database optimizers are the left-deep tree and the bushy tree.

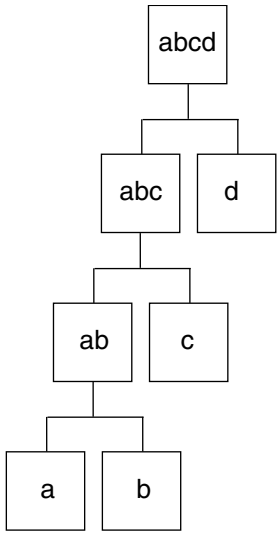
Left-Deep Search Trees

When a left-deep search tree is used to analyze possible join orders, the number of possibilities produced is relatively small, as characterized by the following equation, where *n* is the number of tables being joined.

Number of join orders = n!

For a four-table join, the number of join orders using a left-deep search tree is only 4!, or 24. Left-deep join trees are used by many commercial relational systems, but there are other methods that can produce many more possible join sequences, making it more likely to find a better join plan.

The following graphic illustrates a left-deep join tree for a four-table join.



ff07D413

Bushy Search Trees

Bushy trees are an optimal method for generating more join order combinations. At the same time, the number of combinations generated can be prohibitive (see "Possible Join Orders as a Function of the Number of Tables To Be Joined" on page 2-70), so the Optimizer uses several heuristics to prune the search space. For example, with the exception of star joins, unconstrained joins are not considered.

The following equation calculates the total number of join orders (before pruning) for *n* tables based on a bushy join tree search.

$$\text{Number of join orders} = n! \cdot \binom{2(n-1)}{(n-1)} \cdot \frac{1}{n}$$

The term $\binom{2(n-1)}{(n-1)}$ in this equation represents the number of combinations. More formally, it is the number of (n - 1) subsets of a set of 2(n - 1) elements.

To solve for it, you must substitute the term $\frac{n!}{k!\,(n-k)!}$, where *n* represents the term 2(n - 1) and *k* represents the term (n - 1).

Given the same four table case used for the left-deep tree case, the result is as follows.

$$\text{Number of join orders} = 4! \cdot \frac{(2(4-1))!}{(4-1)!\,(4-1)!} \cdot \frac{1}{4} = 120$$

Ignoring the pruning of unconstrained joins, this method produces an order of magnitude more join possibilities that can be evaluated than the left-deep tree method. Bushy trees also provide the capability of performing some joins in parallel. For example, consider the following four table case.

```
((A JOIN B) JOIN (C JOIN D).
```

`(A JOIN B)` and `(C JOIN D)` can be dispatched for processing at the same time. A system that uses only left-deep trees cannot perform this kind of parallel join execution.

Consider the case of four tables. The number of permutations of four tables is 4! or 24. If the tables are named a, b, c, and d, then those 24 permutations are as follows:

```
abcd, abdc, acbd, acdb ... dcba
```

Because the Optimizer can only join two tables at a time, these 24 permutations must be further partitioned into all their possible binary join orders as follows:

```
abcd => (((ab)c)d)
        ((ab)(cd))
        (a(b(cd)))
        ((a(bc))d)
        ((a(b(cd))
             .
             .
             .
```

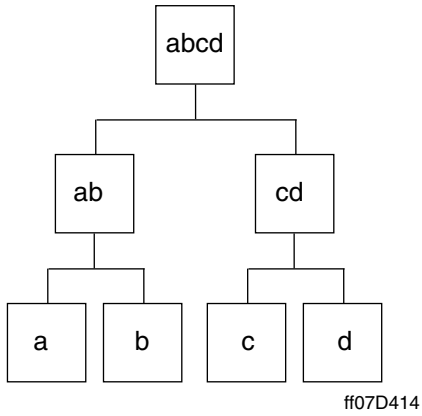
and so on.

The following graphic illustrates one possible sequence of binary joins of tables a, b, c, and d. Note the following about this description:

- The graphic is read from the bottom up.
- Intermediate join results are referred to as relations, not tables.

The illustrated process can be described as follows.

Stage	Process
1	Join table a with table b to create the intermediate relation ab.
2	Join table c with table d to create the intermediate relation cd.
3	Join relation ab with relation cd to create the final joined result, relation abcd.



Depending on table demographics and environmental costs, the Optimizer might just as likely produce the join sequence presented below.

Stage	Process
1	Join table a with table b to create the intermediate relation ab.
2	Join relation ab with table c to create the intermediate relation abc.
3	Join relation abc with table d to create the final joined result, relation abcd.

The Teradata query optimizer uses various combinations of join plan search trees, sometimes mixing left-deep, bushy, and even right-deep branches within the same tree. The Optimizer is very intelligent about looking for plans and uses numerous field-proven heuristics to ensure that more costly plans are eliminated from consideration early in the costing process in order to optimize the join plan search space.

Possible Join Orders as a Function of the Number of Tables To Be Joined

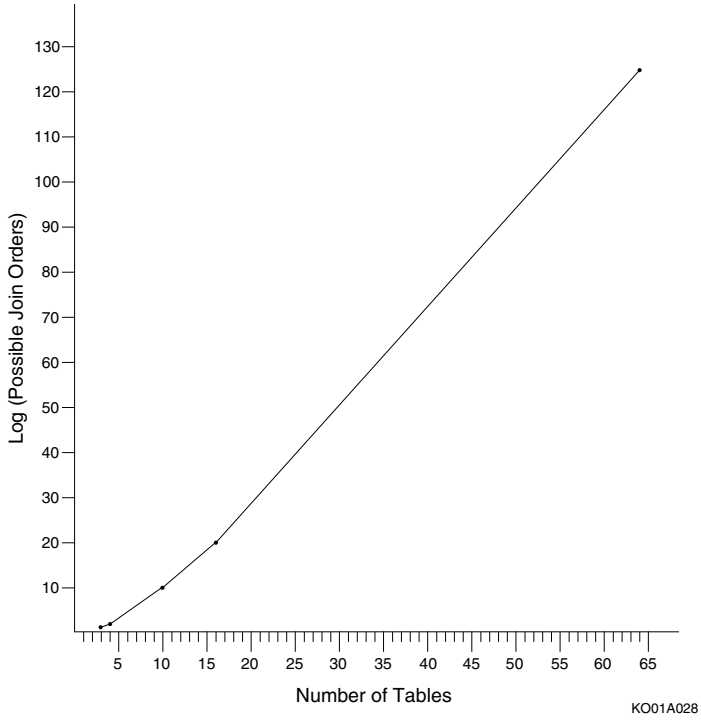
The following table illustrates the dilemma the Optimizer faces when it selects the order in which to join tables in a join plan. The table demonstrates how rapidly the possibilities for ordering binary joins escalates as the total number of tables joined increases.

To help paint a picture of how vast the number of combinations becomes as the number of tables increases, this table uses the metaphor of grains of sand.

Number of Tables Joined	Number of Possible Join Orders	log(Possible Join Orders)	Grains of sand
3	12	1.079	12
4	120	2.079	120
10	$1.8*10^{10}$	10.255	6 sand boxes
16	$2.0*10^{20}$	20.301	All the grains of all the beaches and all the deserts in the world
64	$1.2*10^{124}$	124.079	All the grains required to fill the known universe

The following graph shows the log-linear plot of the data in this table: the expansion in the number of possible join orders as a function of the number of tables being joined.

Because the plot is essentially a straight line, the growth in the number of possible join orders can be characterized as exponential.



Reduce the Number of Participating Rows With Careful Query Coding

The following example illustrates one way the number of participant rows in a join can be reduced.

In this example, the Parts table has 4 000 000 rows. Note that there can be up to 3 990 000 rows per value (probably due to a large number of SHIP# nulls).

Key			
Abbreviation	**Meaning**	**Abbreviation**	**Meaning**
PK	Primary Key	UPI	Unique Primary Index
FK	Foreign Key	SA	System Assigned

Statistics	Shipment Table	
500 Rows	Ship#	…
PK/FK	PK,SA	
	UPI	
Dist values	500	
Max rows/val	1	
Typ rows/val	1	

Statistics	Parts Table		
4 000 000 Rows	Part#	Ship#	…
PK/FK	PK,SA		
	UPI		
Dist values	$4.0*10^6$	501	
Max rows/val	1	$3.99*10^6$	
Typ rows/val	1	20	

The projected columns of all 4 000 000 rows of the Parts table participate in the join if the following query is submitted.

```
SELECT …
FROM Shipment, Part
WHERE Shipment.Ship# = Part.Ship#;
```

However, only the projected columns of 10 000 rows participate in the join if the query is modified to eliminate the nulls.

```
SELECT …
FROM Shipment, Part
WHERE Part.Ship# IS NOT NULL
AND Shipment.Ship# = Part.Ship#;
```

Evaluating Join Orders

Introduction

While it is possible to select an optimum join order when a small number of tables is to be joined, the exponentially escalating choices of binary join orders available to the Optimizer rapidly reach a point at which various algorithms and heuristics must replace the brute force method of evaluating all possible combinations against one another to determine the lowest cost join plan.

Note that the enumeration of join orders is a separate process from the costing of alternatives.

Process Overview

The following table provides a logical explanation of the process the Optimizer follows when it evaluates join orders in the process of generating a join plan.

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>Use a recursive greedy algorithm with a one-join lookahead to evaluate the existing join conditions.
<table>
<tr><th>IF the following conditions are found ...</th><th>THEN ...</th></tr>
<tr><td>• The cost of the one-join lookahead plan exceeds a threshold value.
• There exists one table larger than the threshold.
• The query has no outer join requirement.</td><td>Generate a five-join lookahead plan and evaluate its cost.</td></tr>
</table></td></tr>
<tr><td>2</td><td>Keep the less costly plan generated.</td></tr>
<tr><td>3</td><td>Evaluate the new existing join conditions.
<table>
<tr><th>IF the following conditions are found ...</th><th>THEN ...</th></tr>
<tr><td>• The cost of the current plan exceeds some threshold.
• A star join might be required.</td><td>Generate a better plan that uses star join optimization if such a plan can be found.
Otherwise, go to Stage 4 on page 2-73.</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>4</td><td>Find the best 3-way or n-way join plans among the combinations considered.
Follow these rules to find them:
• Use a depth-first search.
• Skip a join plan if any one of the following conditions is detected:
  • A less costly plan exists that delivers a result with similar or better attributes.
  • The accumulated cost exceeds that of the current candidate join plan.
• Only consider joins between connected tables.
• Join a nonconnected table only after all connected tables have been joined.</td></tr>
<tr><td>5</td><td>Create a temporary join relation before doing the next lookahead.
<table><tr><th>IF a unique ID is . . .</th><th>THEN do the following . . .</th></tr><tr><td>not used</td><td>1 Build the assignment list for the temporary relation.
2 Use it to estimate the following things:
• Spool size
• Output row cost
3 Map the term information of the residual term list to refer to the new relation.
When this temporary relation is removed, the assignment list is reused and the term list is remapped to refer to the inputs.</td></tr><tr><td>used</td><td>Use existing map list information to estimate the following things:
• Spool size
• Output row cost</td></tr></table></td></tr>
</table>

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>6</td><td>Evaluate the best current join plan.

<table><tr><th>IF ...</th><th>THEN ...</th></tr><tr><td>the following conditions are found for the first two joins of the best join plan:
• The joins involve one table connected with two tables not connected to each other
• Neither join is a Product join</td><td>try a compromise join where the two unconnected tables are Product-joined in the first join and that result is then joined with a third table.</td></tr><tr><td>the compromise join plan is less costly than the best current join plan</td><td>replace the first two joins on the best join plan with the compromise plan.</td></tr></table>
The compromise plan is designed to enable a star join plan.</td></tr>
<tr><td>7</td><td>Materialize the first join of the best N joins into a relation and evaluate the materialized relation for the following conditions:
• The number of lookaheads is sufficient to generate the complete join plan.
• The compromise join plan is not used.

<table><tr><th>IF the conditions are ...</th><th>THEN the ...</th></tr><tr><td>met</td><td>remaining joins of the best plan are also committed.</td></tr><tr><td>not met</td><td>second join is also committed if it joins another table with the result of the first join via the primary index for the newly joined table.</td></tr></table></td></tr>
<tr><td>8</td><td>Generate connection information for new relations based on the predicates.</td></tr>
<tr><td>9</td><td>Iterate Stages 1 through 8 until only one active relation remains.</td></tr>
</table>

Example

Consider the following query.

```
SELECT *
FROM t1, t2, t3, t4
WHERE x1=x2
AND y1=y3
AND z1 = y4 ;
```

The first picture in the following graphic illustrates the connections among the tables in this query.

The explicit connections are these.

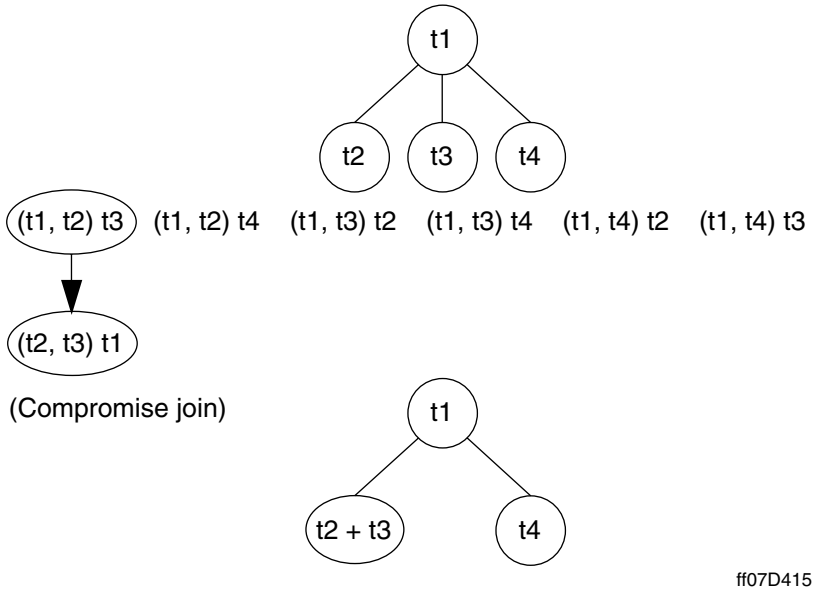
- t1 is connected to t2 because of the condition x1 = x2
- t1 is connected to t3 because of the condition y1 = y3
- t1 is connected to t4 because of the condition z1 = y4

We know from stage 6 in the process table on page 2-74 that when the following conditions are true, the Optimizer tries to make a compromise join where the two unconnected tables are product-joined and the result is joined with the third table.

- The first two joins of the best plan involve a table connected with two tables *not* connected to one another
- Neither join is a product join.

The second picture in the graphic indicates that the join of t1 with t2 followed by a join of the resulting relation t1t2 with t3 is not as good as the compromise join of the unconnected tables t2 with t3 followed by a join of the resulting relation t2t3 with t1.

Because of this, the compromise join is selected for the join plan, as indicated by the third picture in the graphic.



Lookahead Join Planning

Introduction

When it is planning an optimal join sequence for a query, the Optimizer uses a "what if?" method to evaluate the possible sequences for their relative usefulness to the query plan. This "what if?" method is commonly referred to as lookahead in join optimization because the evaluation looks several joins forward to examine the ultimate cost of joining relations in various orders.

Depending on certain well-defined circumstances (see "Five-Join Lookahead Processing of n-Way Joins" on page 2-79), Teradata uses both one-join and five-join lookahead methods to determine optimal join plans.

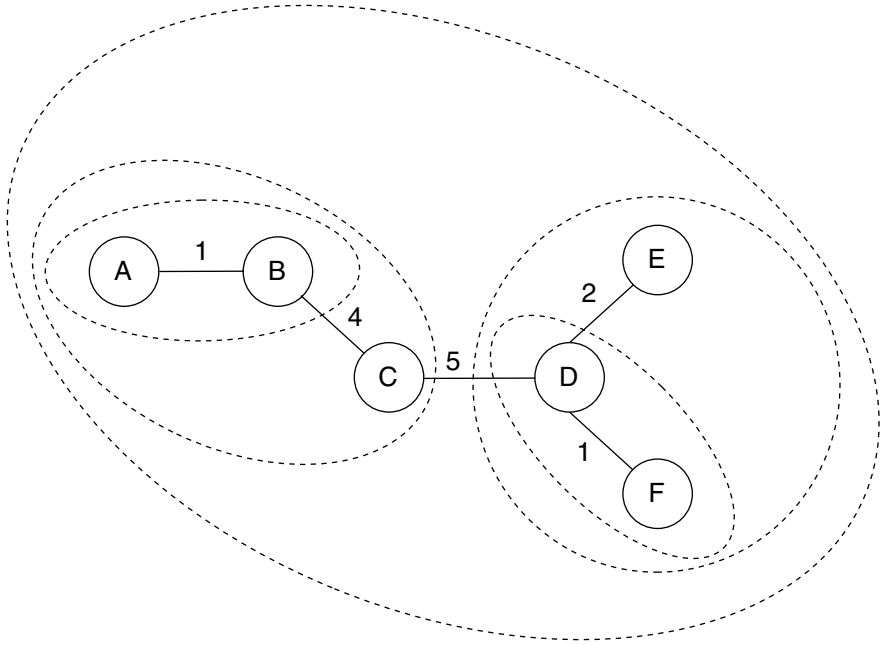
One-Join Lookahead Processing of *n*-Way Joins

The goal of any *n*-way join analysis is to reduce the number of join possibilities to a workable subset. The Teradata optimizer search of join space is driven by join conditions. To this end, is uses a recursive greedy algorithm to search and evaluate connected relations. In this context, the term *greedy* means the search generates the next logical expression for evaluation based on the result of the costing determined by the previous step. From the perspective of search trees, a greedy algorithm always begins at the *bottom* of the tree and works its way toward the top.

The Optimizer pursues the following strategy in evaluating join sequences for *n*-way joins using a one-join lookahead algorithm.

Stage	Process
1	Evaluate 3-way joins for the least costly join.
2	Commit the first join of the least costly 3-way join.
3	Replace the two source relations used in the join with their resulting joined relation.
4	Loop through the process until only one relation remains.
5	Use a single product join when possible.

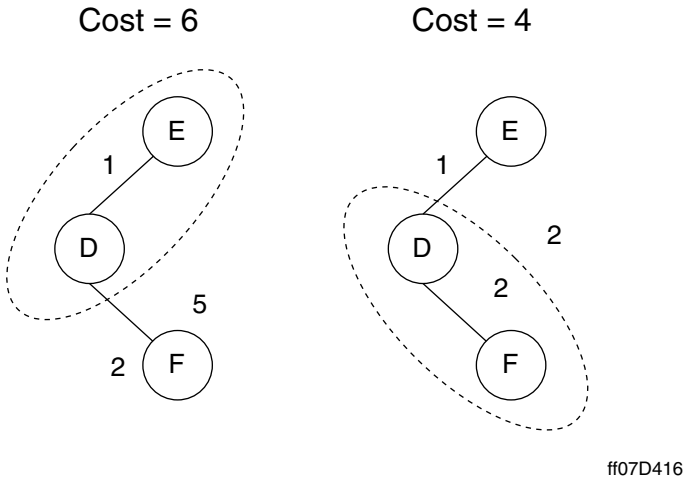
The following graphic illustrates the result of an Optimizer analysis of the join space for a 6-way join. The relative cost of each join is given by an integer within the selected join sequence.



These relationships are explicitly stated in the following table.

This binary join ...	Has this cost ...	And produces this relation as a result ...
A - B	1	AB
AB - C	4	ABC
D - F	1	DF
DF - E	2	DFE
ABC - DFE	5	ABCDEF

The following graphic illustrates one-join lookahead processing of an *n*-way join and how the Optimizer might select one join sequence over another based on their relative costs.



The cost relationships among the tables are explicitly stated in the following table.

This binary join …	Has this cost …	And produces this relation as a result …
D - E	1	DE
D - F	2	DF
DE - F	5	DEF
DF - E	2	DEF
ABC - DFE	5	ABCDEF

The Optimizer sums the individual binary join costs to produce an overall join cost which it then evaluates to determine the least costly join sequence. For this example, the following join sequence costs are determined.

```
((DE)F) = 1 + 5 = 6
((DF)E) = 2 + 2 = 4
```

As a result, the second sequence, joining tables D and F first following by joining the resulting relation DF with table E, is selected for the join plan because it is the less costly join.

Five-Join Lookahead Processing of *n*-Way Joins

Under some circumstances, the Optimizer pursues a second join plan derived from a 5-join lookahead analysis of the join space. This analysis occurs only when any of the following conditions are found to be true.

- The cost of the join plan generated by a 1-join lookahead analysis exceeds a heuristically-defined threshold.
- The cost of one of the tables being joined exceeds a heuristically-defined threshold.
- No outer joins are required to process the query.

In such a case, the second join plan is generated using a 5-join lookahead. The relative costs of the two join plans are evaluated and the better plan is kept.

Join Methods

General Remarks

The Optimizer has many join methods, or modes, to choose from to ensure that any join operation is fully optimized. Each type of join method is described in the sections that follow.

The particular processing described for a given type of join (for example, duplication or redistribution of spooled data) might not apply to all joins of that type.

Guidelines

The following two procedures are key factors for optimizing your SQL queries.

- Collect statistics regularly on all your regular join columns. Accurate table statistics are an absolute must if the Optimizer is to consistently choose the best join plan for a query.

  For further information on the Optimizer form of the COLLECT STATISTICS statement, see "COLLECT STATISTICS (Optimizer Form)" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.
- To obtain an accurate description of the processing that will be performed for a particular join, always submit an EXPLAIN statement or perform the Visual EXPLAIN utility for the query containing the join expression.

  You can often reformulate a query in such a way that resource usage is more highly optimized.

  For further information on the EXPLAIN statement modifier, see "EXPLAIN Modifier" in the chapter "SQL Data Manipulation Language Statement Syntax" of *Teradata RDBMS SQL Reference, Volume 6*.

  For further information on the Visual EXPLAIN utility, see *Teradata Visual Explain User Guide*.

Summary of the Most Commonly Used Join Algorithms

The following table summarizes some of the major characteristics of the most commonly used join algorithms:

Join Method	Important Properties
Product	• Always selected by the Optimizer for WHERE clause inequality conditions. • High cost because of the number of comparisons required.
Merge/Hash	• When done on matching primary indexes, do not require any data to be redistributed. • Hash indexes are often better performers and should be used whenever possible.
Nested	• Only join expression that generally does not require all AMPs. • Preferred join expression for OLTP applications.

Product Join

Definition

A product join compares every qualifying row from one table to every qualifying row from the other table and saves the rows that match the WHERE condition. This operation is called a product join because the number of comparisons needed is the product of the number of qualifying rows in the two tables.

How a Product Join is Processed

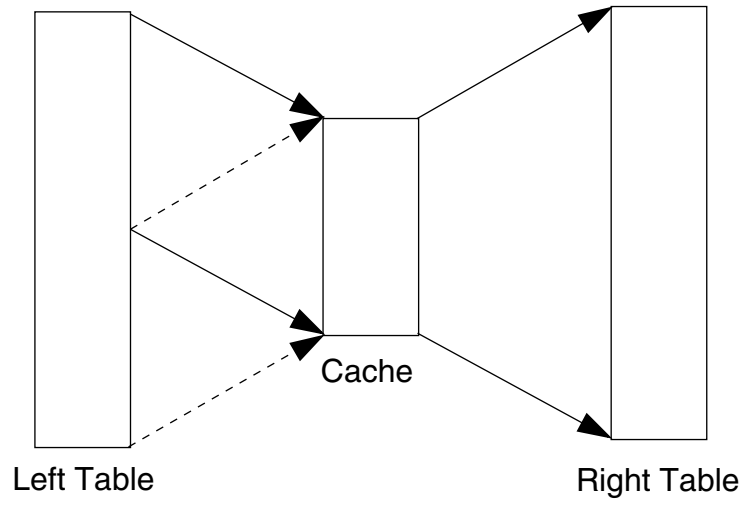
The following table outlines the product join process:

Stage	Process
1	Cache the left table rows.
2	Join each row of the right table with each row from the cached left table.

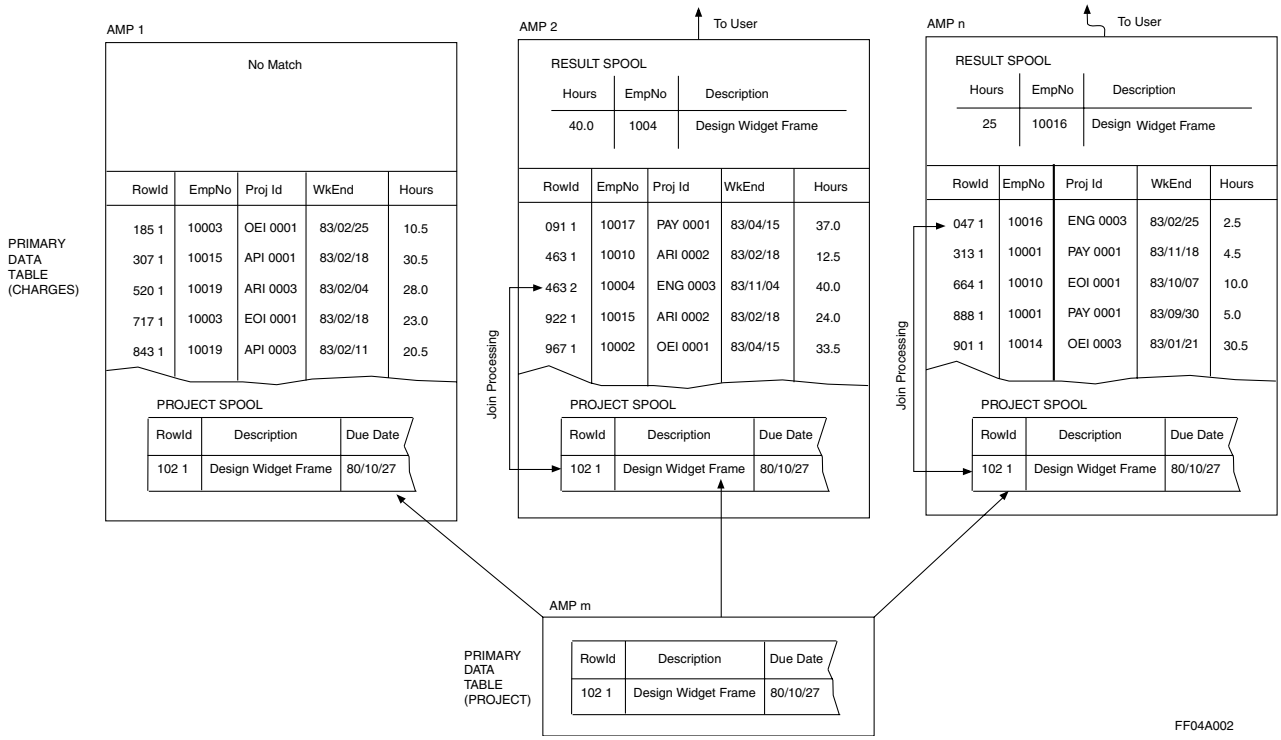
An overall join condition is a WHERE constraint that links tables to be joined on a column that is common to each, as shown in the following example:

```
WHERE Employee.DeptNo = Department.DeptNo
```

The following graphic illustrates the generic product join process.



The following graphic illustrates a concrete product join.



Cost of a Product Join

Product joins are relatively more time consuming than other types of joins because of the number of comparisons that must be made. The Optimizer uses product joins under the following conditions.

- The join condition is not based on equality
- The join conditions are ORed
- It is less costly than other join forms

The product join is usually the most costly in terms of system resources, and is used only when there is no more efficient method, such as a merge join or a nested join. However, a product join is useful because it can resolve any combination of join conditions.

Example 1

For example, consider the following join request, which identifies hours that have been charged time to a particular project for a week-ending date that is later than the project due date:

```
SELECT Hours, EmpNo, Description
FROM Charges, Project
WHERE Charges.Proj_Id = 'ENG-0003'
AND Project.Proj_Id = 'ENG-0003'
AND Charges.WkEnd > Project.DueDate ;
```

To process this request, the UPI on Proj_Id is used to access the qualifying row in the Project table directly. The row is copied into a spool file, which is then replicated on every AMP on which the Charges table is stored.

On each of these AMPs, the resident rows of the Charges table are searched one at a time for the qualifying value in the Proj_Id field.

When a row is found, it is joined with a copy of the Project row.

When the product join operation is complete, each AMP involved returns its results to the PE via a BYNET merge.

Note that blocks from one table may have to be read several times if all of the rows of a product join cannot be held in AMP memory.

This can make product joins very costly, especially if there are many comparisons. If the system is doing a costly join, it may be because of a poorly written or nonexistent WHERE clause.

Example 2

Consider the following table definitions:

Employee

Enum	Name	Dept
PK		FK
UPI		
1	Brown	200
2	Smith	310
3	Jones	310
4	Clay	400
5	Peters	150
6	Foster	400
7	Gray	310
8	Baker	310

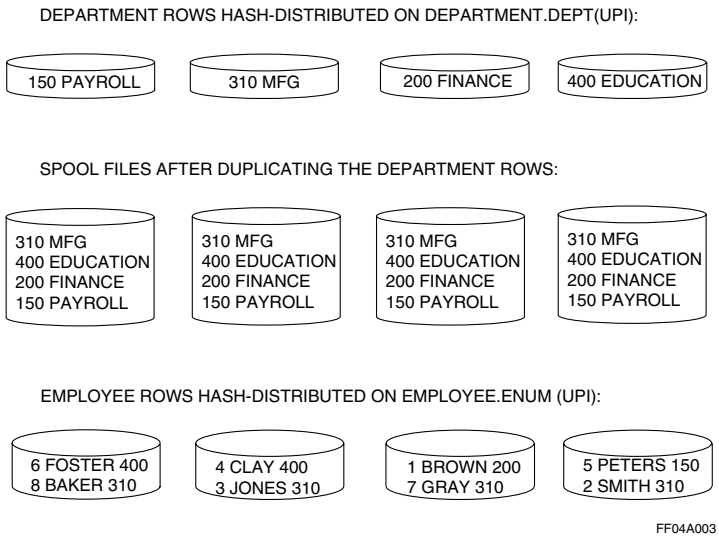
Department

Dept	Name
PK	
UPI	
400	Education
150	Payroll
200	Finance
310	Mfg

Assume that a join query such as the following is executed against these tables:

```
SELECT *
FROM Employee, Department
WHERE Employee.Dept > Department.Dept;
```

The rows are redistributed as shown in the graphic.



Notice that the rows of the smaller table, Department, are duplicated on all AMPS.

The smaller table is determined by multiplying the number of required column bytes by the number of rows.

Merge Join

Definition

A merge join retrieves rows from two tables and then puts them onto a common AMP based on the row hash of the columns involved in the join. The system sorts the rows into join column row hash sequence, then joins those rows that have matching join column row hash values.

In a merge join, the columns on which tables are matched are also the columns on which both tables, or redistributed spools of tables, are ordered. A merge join is generally more efficient than a product join because it requires fewer comparisons and because blocks from both tables are read only once.

Two different merge join algorithms are available:

- Slow path

  The slow path is used when the left table is accessed using a read mode other than an all-rows scan. The determination is made in the AMP, not by the Optimizer.
- Fast path

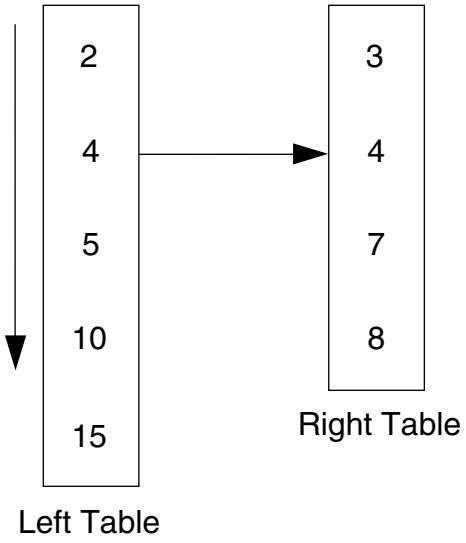
  The fast path is used when the left table is accessed using the all-row scan reading mode.

Slow Path Merge Join

The process applied by the slow path merge join algorithm is provided in the following table:

Stage	Process
1	Read each row from the left table.
2	Join each left table row with the right table rows having the same hash value.

The following graphic illustrates the generic slow path merge join process.

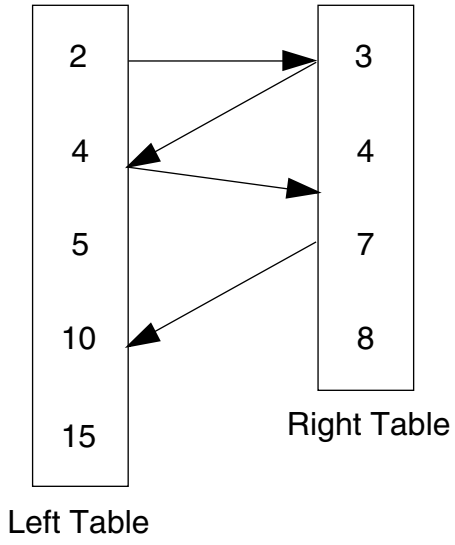


Fast Path Merge Join

The process applied by the fast path merge join algorithm is provided in the following table:

Stage	Process
1	Read a row from the left table and record its hash value.
2	Read the next row from the right table that has a row hash >= to that of the left table row. **IF the row hash values are ...** / **THEN ...** equal / join the two rows. not equal / use the larger row hash value to read the row from the other table.

The following graphic illustrates the generic fast path merge join process.



Merge Join Strategies

Merge joins use one of the following distribution strategies:

Strategy Number	Strategy Name	Stage	Process
1	Hash Redistribute	**1**	Hash redistribute one or both sides (depending on the primary indexes used in the join)
		2	Sort the rows into join column row hash sequence.
2	Duplicate Table	**1**	Duplicate the smaller side on all AMPs
		2	Sort the rows into the row hash sequence of the join column.
		3	Locally copy the other side and sort the rows into row hash sequence of the join column.
3	Matching Indexes	**1**	No redistribution is required if the primary indexes are the join columns and if they match.

FOR a representation of this strategy . . .	See this illustration . . .
Hash redistribution	"Merge Join Row Distribution (Hash Redistribution)" on page 2-91.
Table duplication	"Merge Join Row Distribution (Duplicate Table)" on page 2-91.
Index matching	"Merge Join Row Distribution (Matching Indexes)" on page 2-92.

Example 1

The following SELECT statement determines who works in what location:

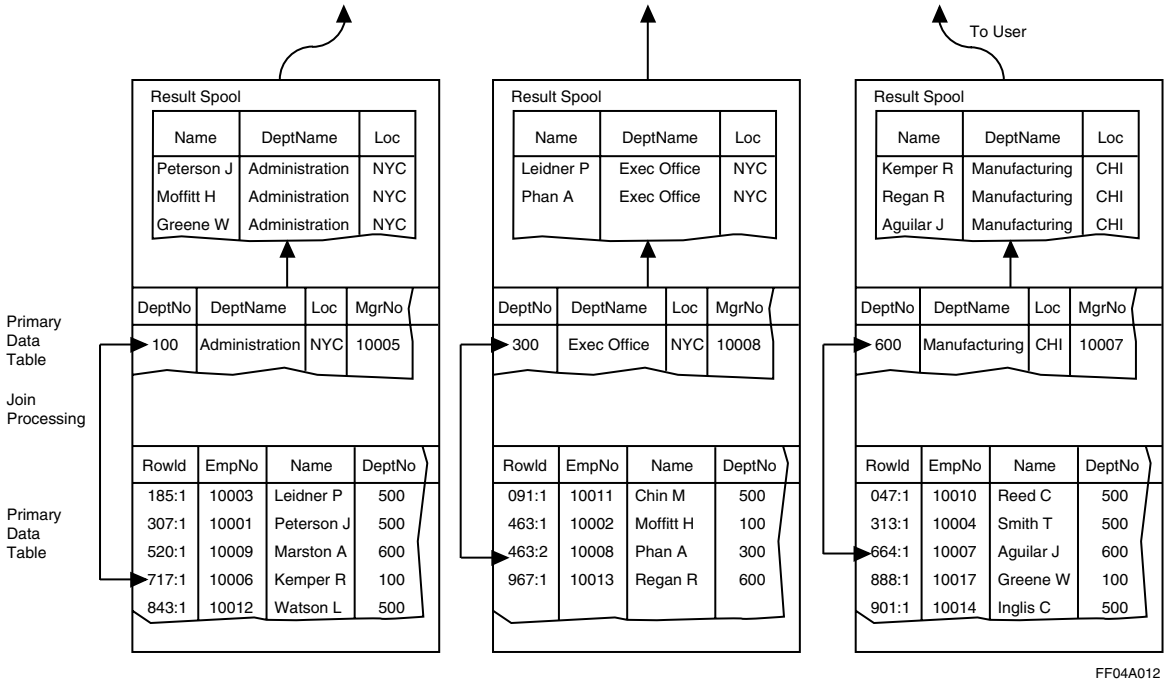
```
SELECT Name, DeptName, Loc
FROM Employee, Department
WHERE Employee.DeptNo = Department.DeptNo;
```

The following table presents the stages of processing this merge join:

Stage	Process
1	Because Department rows are distributed according to the hash code for DeptNo (the unique primary index of the Department table), Employee rows are themselves redistributed by the hash code of their own DeptNo values.
2	The redistributed Employee rows are stored in a spool file on each AMP and sorted by the hash value for DeptNo. This puts them in the same order as the Department rows on the AMP.
3	The hash value of the first row from the Department table will be used to read the first row with the same or bigger row hash from the Employee spool. That is, rows from either table of a MERGE join will be skipped where possible.
4	If there is a hash codes match, an additional test is performed to verify that the matched rows are equivalent in DeptNo value as well as hash value.
5	If there is no hash code match, then the larger of the two hash codes is used to position to the other table. The hash code and value comparisons continue until the end of one of the tables is reached.
6	On each AMP the Name, DeptName, and Loc values from each of the qualifying rows are placed in a result spool file.
7	When the last AMP has completed its merge join, the contents of all result spools are merged and returned to the user.

When many rows fail to meet a constraint, the hash-match-reposition process might skip several rows. Skipping disqualified rows can speed up the merge join execution, especially if the tables are very large.

The following graphic illustrates this merge join process:



Example 2

This example uses the following table definitions:

Employee

Enum	Name	Dept
PK		FK
UPI		
1	Brown	200
2	Smith	310
3	Jones	310
4	Clay	400
5	Peters	150
6	Foster	400
7	Gray	310
8	Baker	310

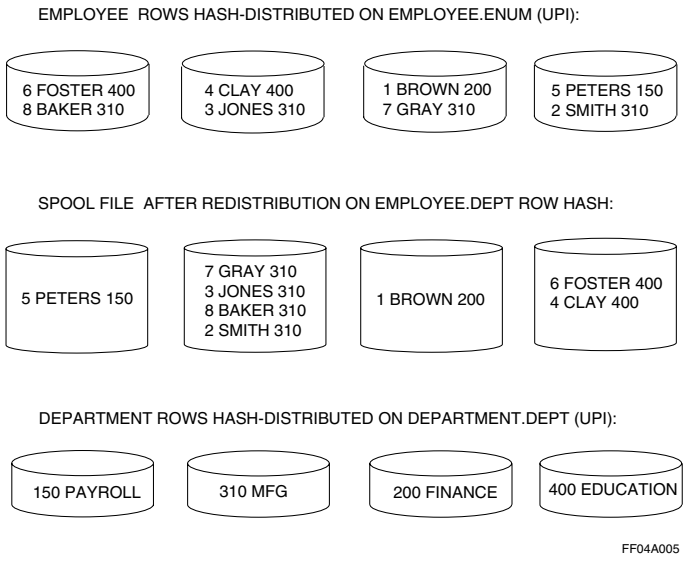
Department

Dept	Name
PK	
UPI	
400	Education
150	Payroll
200	Finance
310	Mfg

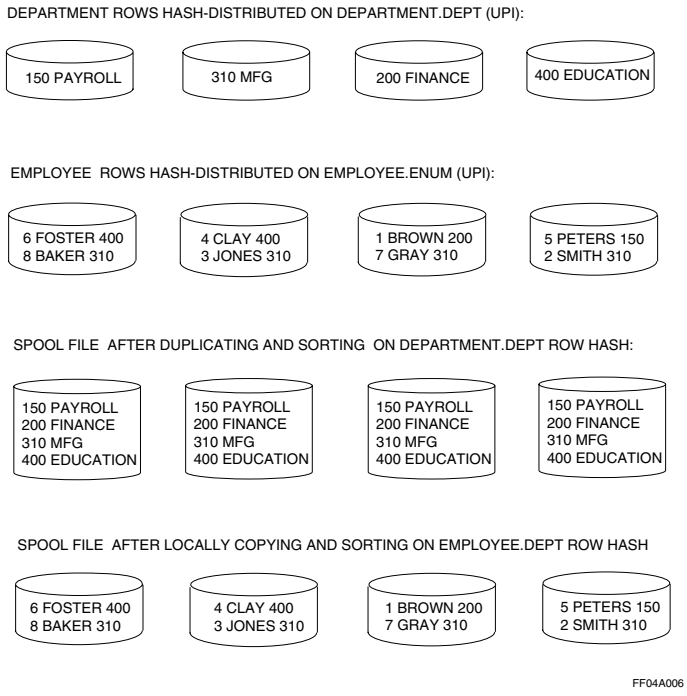
One of the merge join row redistribution methods (see "Merge Join Row Distribution (Hash Redistribution)" on page 2-91 or "Merge Join Row Distribution (Duplicate Table)" on page 2-91) is used if you perform the following SELECT statement against these tables.

```
SELECT *
FROM Employee, Department
WHERE Employee.Dept = Department.Dept;
```

Merge Join Row Distribution (Hash Redistribution)



Merge Join Row Distribution (Duplicate Table)



Example 3

This example uses the following Employee and Employee_phone table definitions.

Employee

Enum	Name
PK	
UPI	
1	Brown
2	Smith
3	Jones
4	Clay
5	Peters
6	Foster
7	Gray
8	Baker

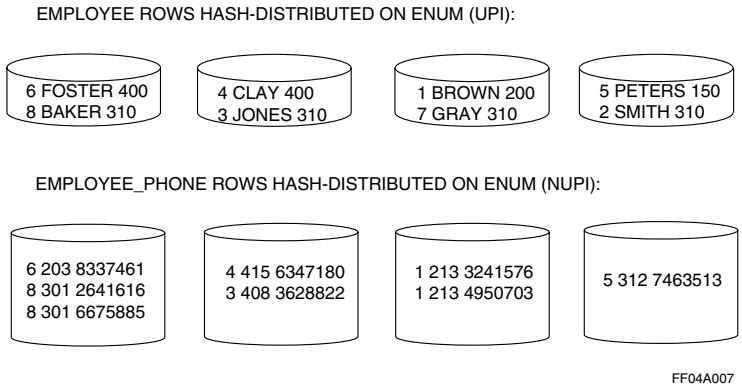
Employee_phone

Enum	Area Code	Phone
PK		
FK		
NUPI		
1	213	3241576
1	213	4950703
3	408	3628822
4	415	6347180
5	312	7463513
6	203	8337461
8	301	6675885
8	301	2641616

The matching indexes row distribution strategy shown in "Merge Join Row Distribution (Matching Indexes)" on page 2-92 is used if you perform the following SELECT statement against these tables.

```
SELECT *
FROM Employee, Employee_Phone
WHERE Employee.Enum = Employee_Phone.Enum;
```

Merge Join Row Distribution (Matching Indexes)



Hash Join

Introduction

Hash Join is a method that performs better than merge join under certain conditions. The performance gain comes mainly from eliminating the need for sorting the join tables before performing the join.

Two different hash join algorithms are available:

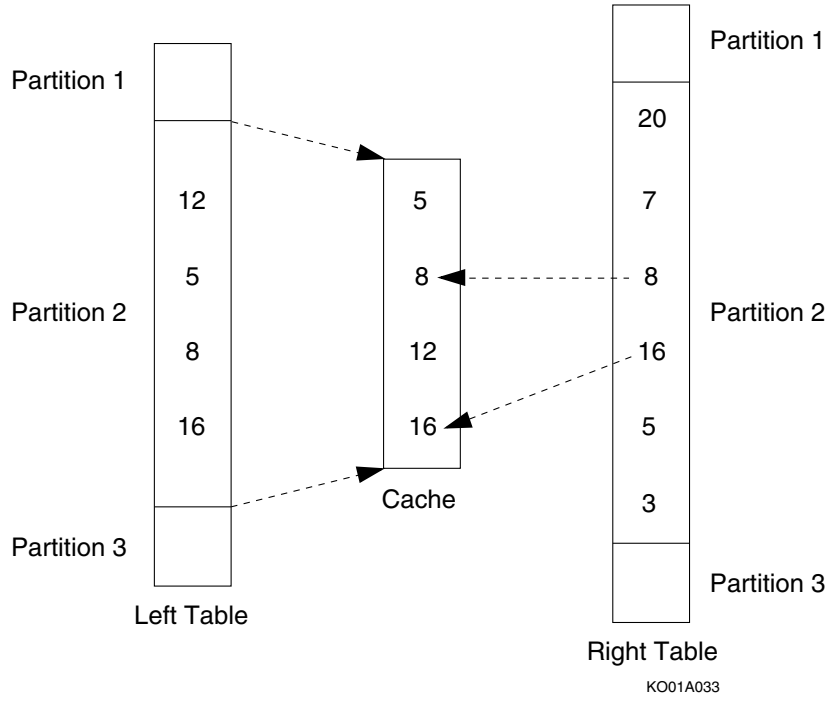
- Classic
- Direct table

Classic Hash Join

The process applied by the classic hash join algorithm is provided in the following table:

Stage	Process
1	Read a row from the right table, which is an unsorted spool file containing the row hash value for each row as well as its row data.
2	Match each right row with all the left table rows having the same row hash.
3	Join the rows.

The following graphic illustrates the classic hash join process:



When a table or spool file pair is too large to fit into memory for hash join processing, then each table or spool file is split into several smaller, range-bounded partitions whose pairs do fit into the available memory. Each left table partition is then hash-joined in turn with its corresponding right table partition. The graphic shows partition 2 of the left table being hash-joined with partition 2 of the right table.

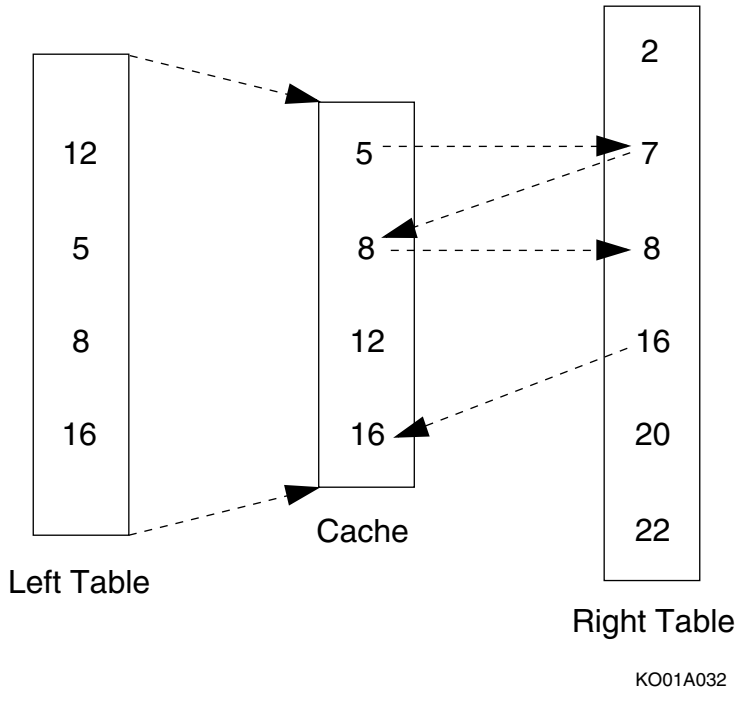
Partitions are created by hashing the left and right table rows on their join columns in such a way that rows from a given left table partition can only match with rows in the corresponding right table partition. The process of creating the partitions is called *fanning out* in EXPLAIN reports.

Direct Table Hash Join

The following table describes the process applied by the direct table join algorithm:

Stage	Process
1	Sort the right table by row hash and read it directly using row hash match scan logic.
2	Read the left table rows from the hash table.
3	Join the rows.

The following graphic illustrates the direct table hash join process:



Hash Join Control Variables

The following two user-definable fields, accessible through the DBS Control utility, are a part of the performance controls in the DBSControl Record. Use them to optimize the performance of your hash joins.

DBSControl Field	Function
HTMemAlloc	Varies the hash table size and is used by the SQL optimizer in the following formula: $\text{Hash table size} = \frac{\text{HTMemAlloc}}{100} \times \frac{\text{Amount of memory}}{\text{vproc}}$ Valid input values range from 0 to 10 (percent). The recommended maximum value is 1. Higher values can allocate so much memory to hash join processing that the system either hangs or crashes. Default is 0 (percent), which disables off hash join.
SkewAllowance	Provides for data skew by making the size of each partition smaller than the hash table. Valid input values range from 20 to 80 (percent). Default is 75 (percent), indicating that the partition size is set to 25 percent of the hash table to allow the actual partition to be four times more than the estimate made by the optimizer before it is too large to fit into the hash table.

See *Teradata RDBMS Utilities* for more information on the DBSControl Record.

Hash Join Example

```
EXPLAIN
SELECT Employee.EmpNum,Department.DeptName, Employee.Salary
FROM Employee, Department
WHERE Employee.Location = Department.Location ;

***Help information returned. 30 rows.
***Total elapsed time was 1 second. This request returns the
following explanation:

Explanation -----------------------------------------------------

1) First, we lock a distinct PERSONNEL."pseudo table" for read on
   a RowHash to prevent global deadlock for PERSONNEL.Department.
2) Next, we lock a distinct PERSONNEL."pseudo table" for read on
   a RowHash to prevent global deadlock for PERSONNEL.Employee.
3) We lock PERSONNEL.Department for read, and we lock
   PERSONNEL.Employee for read.
4) We do an all-AMPs RETRIEVE step from PERSONNEL.Employee by way
   of an all-rows scan with no residual conditions into Spool 2
   fanned out into 22 hash join partitions, which is redistributed
   by hash code to all AMPs. The size of Spool 2 is estimated to
   be 3,995,664 rows. The estimated time for this step is 3
   minutes and 54 seconds.
5) We do an all-AMPs RETRIEVE step from PERSONNEL.Department by
   way of an all-rows scan with no residual conditions into Spool
   3 fanned out into 22 hash join partitions, which is
   redistributed by hash code to all AMPs. The size of Spool 3 is
   estimated to be 4,000,256 rows. The estimated time for this
   step is 3 minutes and 54 seconds.
6) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of
   an all-rows scan, which is joined to Spool 3 (Last Use). Spool
   2 and Spool 3 are joined using a hash join of 22 partitions,
   with a join condition of ("Spool_2.Location =
   Spool_3.Location"). The result goes into Spool 1, which is
   built locally on the AMPs. The result spool field will not be
   cached in memory. The size of Spool 1 is estimated to be
   1,997,895,930 rows. The estimated time for this step is 4 hours
   and 42 minutes.
7) Finally, we send out an END TRANSACTION step to all AMPs
   involved in processing the request.
-> The contents of Spool 1 are sent back to the user as the result
   of statement 1. The total estimated time is 4 hours and 49
   minutes.

DBS Control Record - Performance Fields:

HTMemAlloc       =  5%
SkewAllowance    = 75%
```

Nested Join

Definition

A nested join is a join for which the WHERE conditions specify a constant value for a unique index in one table and those conditions also match some column of that single row to the primary or secondary index of the second table.

Types of Nested Join

There are two types of nested joins: local and remote.

Local nested joins are more commonly used than remote nested joins. Local nested joins are described in "Local Nested Join" on page 2-98.

Remote nested joins are described in "Remote Nested Join" on page 2-104.

Process

Teradata uses the following general process to perform nested joins:

Stage	Process
1	Retrieve the single row that satisfies the WHERE conditions from the first table.
2	Use that row to locate the AMP having the matching rows on which the join is to be made.

Cost Effectiveness

Nested joins are very cost-effective because they are the only join type that does not always use all AMPs. Because of this, nested joins are generally the best choice for OLTP applications.

The Optimizer can select a nested join only if both of the following conditions are true.

- There is an equality condition on a unique index of one table.
- There is a join on a column of the row specified by the first table to any primary index or USI of the second table. In rare cases, the index on the second table can be a NUSI.

Local Nested Join

Definition

Use of a local nested join implies several things.

- No messages are sent during the execution of the nested join.
- If necessary, the resulting rows of a nested join are redistributed by row hashing the rowID of the right table rows.
- The rowID is used to retrieve the data rows from the right table.

Only local nested joins can result in a rowID join (see "RowID Join" on page 2-117). A rowID join is needed if and only if a nested join is carried out and only rowIDs for rows in the right table are retrieved.

Two different local nested join algorithms are available:

- Slow path (see "Slow Path Local Nested Join" on page 2-100)
- Fast path (see "Fast Path Local Nested Join" on page 2-103)

Join Process as a Function of Secondary Index Type on Equality Conditions

A local nested join can be selected by the Optimizer if there is an equality condition on a NUSI or USI of one of the join tables.

Whether the equality condition is made on a USI or a NUSI, steps 3 and 4 in the following process tables (the rowID join) are not always required, depending on the situation. For more information on rowID joins, see "RowID Join" on page 2-117.

<table>
<tr><th>IF the equality condition is on this index type ...</th><th>THEN the left table is ...</th></tr>
<tr><td>USI</td><td>
<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>Hash-redistributed based on the joined field.</td></tr>
<tr><td>2</td><td>Nested joined with the right table.</td></tr>
<tr><td>3</td><td>The resulting rows are redistributed by row hashing the rowID of the right table rows.</td></tr>
<tr><td>4</td><td>The rowID is used to retrieve the data rows from the right table to complete the join.</td></tr>
</table>
</td></tr>
<tr><td>NUSI</td><td>
<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>Duplicated on all AMPs.</td></tr>
<tr><td>2</td><td>Nested joined with the right table.</td></tr>
<tr><td>3</td><td>The resulting rows are redistributed by row hashing the rowID of the right table rows.</td></tr>
<tr><td>4</td><td>The rowID is used to retrieve the data rows from the right table to complete the join.</td></tr>
</table>
</td></tr>
</table>

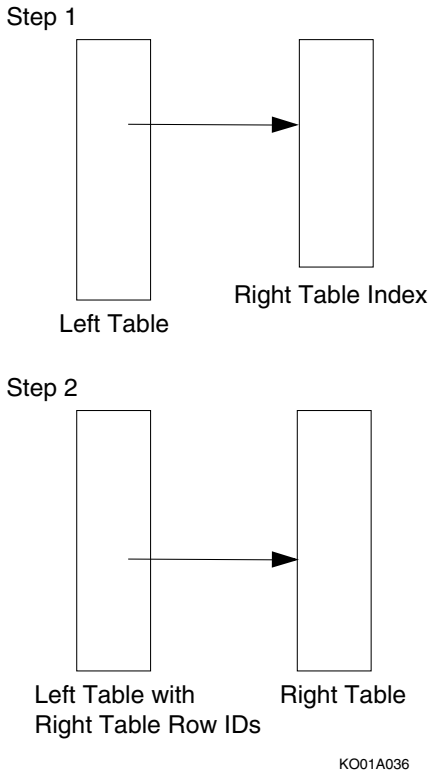
Slow Path Local Nested Join

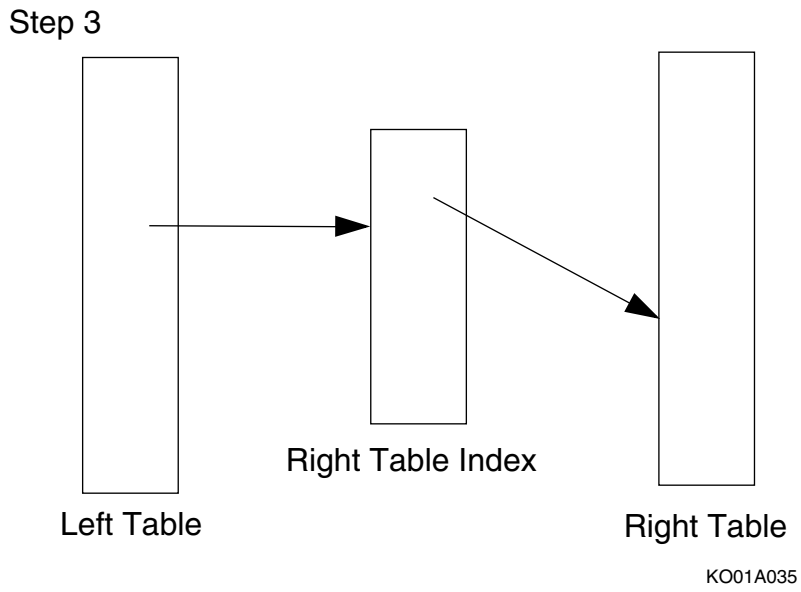
Slow Path Local Nested Join Process

The following table documents the process applied by the slow path nested join algorithm:

Stage	Process
1	Read each row from the left table.
2	Evaluate each left table row against the right table index value.
3	Retrieve the right table index rows that correspond to the matching right table index entries.
4	Retrieve the rowIDs for the right table rows to be joined with left table rows from the qualified right table index rows.
5	Read the right table data rows using the retrieved rowIDs.
6	Produce the join rows.
7	Produce the final join using the left table rows and the right table rowIDs.

The following graphics illustrate the generic slow path local nested join process:





Example 1

The following is an example of a query processed using a slow path local nested join.

To determine who manages Department 100, you could make the following query:

```
SELECT DeptName, Name, YrsExp
FROM Employee, Department
WHERE Employee.EmpNo = Department.MgrNo
AND Department.DeptNo = 100;
```

To process this query, the Optimizer uses the unique primary index value DeptNo = 100 to access the AMP responsible for the Department row with that value. The hash code for the MgrNo value in that row is calculated.

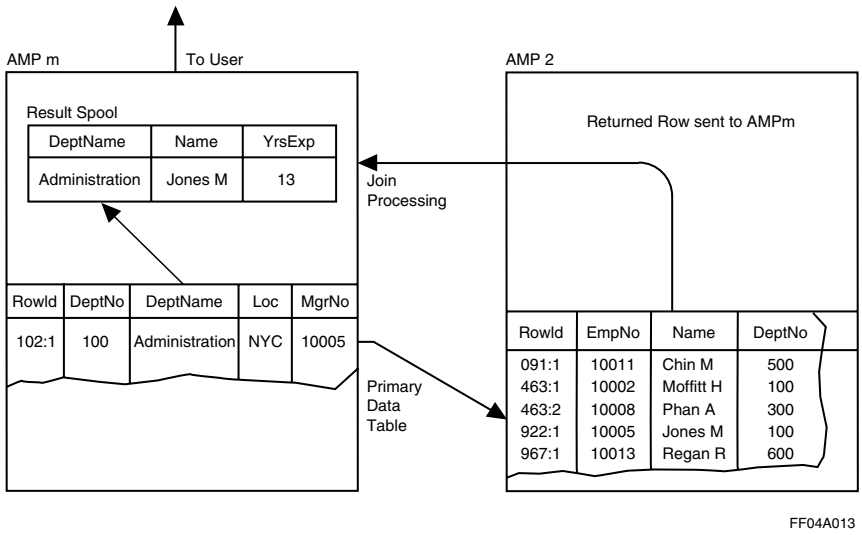
Note that this MgrNo value is the same as the value of EmpNo (the unique primary index of the Employee table) for the employee who manages Department 100. Thus, the hash code that is calculated for MgrNo is the same as the hash code for the equivalent EmpNo value.

The calculated hash code for MgrNo is used to access the AMP responsible for the Employee row that contains the equivalent EmpNo hash code.

The Name and YrsExp information in this row is sent back to the initiating AMP, which places the information, plus the DeptName for Department 100, in a result spool file. This information is returned to the user.

This two-AMP process is illustrated in "Example of a Local Nested Join Operation" on page 2-102.

Example of a Local Nested Join Operation



Example 2

This example shows a query that is processed using a slow path nested join on the Employee and Department tables:

```
SELECT Employee.Name, Department.Name
FROM Employee, Department
WHERE Employee.Enum = 5
AND Employee.Dept = Department.Dept;
```

Employee

Enum	Name	Dept
PK		FK
UPI		
1	Brown	200
2	Smith	310
3	Jones	310
4	Clay	400
5	Peters	150
6	Foster	400
7	Gray	310
8	Baker	310

Department

Dept	Name
PK	
UPI	
400	Education
150	Payroll
200	Finance
310	Mfg

Fast Path Local Nested Join

Fast Path Local Nested Join Process

The process applied by the fast path nested join algorithm is provided in the following table. Note that it is similar to the fast path merge join except that the right table is a NUSI subtable instead of a base table.

This logic returns multiple join rows because there can be multiple rowIDs from the right NUSI subtable for each pair of left and right table rows.

Stage	Process
1	Read a row from the left base table and record its hash value.
2	Read the next row from the right NUSI subtable that has a row hash >= to that of the left base table row.

IF the row hash values are ...	THEN ...
equal	join the two rows.
not equal	use the larger row hash value to read the row from the other table.

Example

The following SELECT statement is an example of a query that is processed using a very simple fast path local nested join:

```
SELECT *
FROM table_1, table_2
WHERE table_1.x_1 = 10
AND table_1.y_1 = table_2.NUSI;
```

Remote Nested Join

Definition

Remote nested joins are used for the case in which a WHERE condition specifies a constant value for a unique index of one table, and the conditions might also match some column of that single row to the primary or secondary index of a second table.

The expression *remote nested join* implies that a message is to be sent to another AMP to get the rows from the right table.

A remote nested join does not always use all AMPs. For this reason, it is the most efficient join in terms of system resources and is almost always the best choice for OLTP applications.

Remote nested joins normally avoid the duplication or redistribution of large amounts of data and minimize the number of AMPs involved in join processing.

The following SELECT statement is an example of a remote nested join in that no join condition exists between the two tables:

```
SELECT *
FROM table_1, table_2
WHERE table_1.USI_1
AND table_2.USI_2 = 1;
```

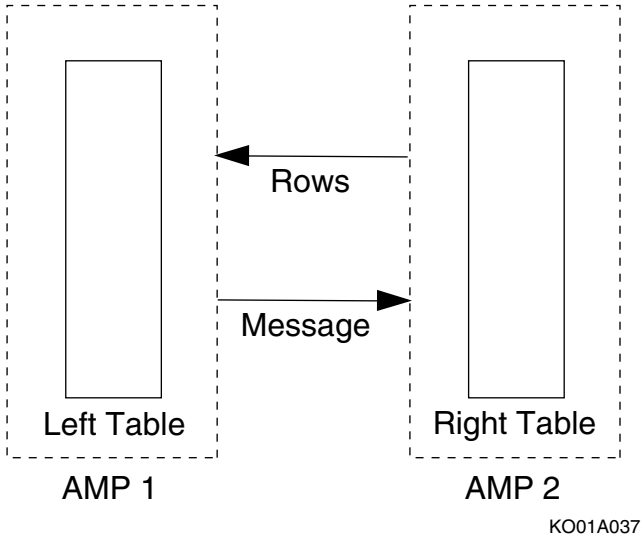
When there is no such join condition, then the index of the second (right) table must be defined with a constant as illustrated by Examples 1, 2, and 3 if a remote nested join is to be used.

How a Remote Nested Join Is Processed

The process applied by the remote nested join is provided in the following table.

Stage	Process
1	Read the single left row.
2	Evaluate the index value for the right table.
3	Read the right table rows using the index value.
4	Produce the join result.

The following graphic illustrates the remote nested join process:



Process

Remote nested joins are used for the condition where one table contains the key to the table with which it is to be joined.

The key can be of any of the following database objects.

- Unique primary index (UPI)
- Nonunique primary index (NUPI)
- Unique secondary index (USI)
- Nonunique secondary index (NUSI)
- Non-indexed column that is matched to an index

If there is such a join condition, and the conditions of the first table match a column of the primary or secondary index of the second table, then the process illustrated in the following table occurs.

Stage	Process
1	Retrieve the single qualifying row from the first table.
2	Use the row hash value to locate the AMP having the matching rows in the second table to make the join.

Examples

A remote nested join can be used when there is no equality condition between the primary indexes of the two tables and other conditions.

This is illustrated by the following example conditions:

Example 1

```
    (table_1.UPI = constant OR table_1.USI = constant)
AND (table_2.UPI = constant OR table_2.USI = constant)
```

In this case, there may or may not be a suitable join condition.

Example 2

```
     (table_1.UPI = constant OR table_1.USI = constant)
AND ((table_2.NUPI = table_1.field)
OR (table_2.USI = table_1.field))
```

Example 3

```
    (table_1.NUPI = constant)
AND (table_2.UPI = table_1.field)
AND (few rows returned for the table_1.NUPI = constant)
```

Nested Join Examples

Introduction

This section provides the EXPLAIN outputs for the same two-table join under two different circumstances.

- No nested join
- Nested join in where the USI for one table is joined on a column from the other table.

Table Definitions

For these examples, assume the following table definitions.

Table Name	Column 1 Name	Column 2 Name	Primary Index Name	Unique Secondary Index Name	Number of Rows
table_1	NUPI	y_1	NUPI	none defined	2
table_2	NUPI	USI_2	NUPI	USI_2	1 000 000

Test Query

The following query is tested against this database, both using and without using nested joins:

```
SELECT *
FROM table_1,table_2
WHERE y_1 = USI_2;
```

Join Plan Without Nested Join

Unoptimized Join Plan

Without using a nested join, the Optimizer generates the following join plan:

Operation	Joined Tables	Total Processing Time
Spool 1:Product Join	duped table_1, direct table_2	1 hour 15 minutes

Completion Time

The total estimated completion time is 1 hour 15 minutes.

EXPLAIN Output for Unoptimized Join Plan

The following EXPLAIN output is generated when nested joins are not used.

```
Explanation
-----------------------------------------------------------------------
1) First, we lock test.tab1 for read, and we lock test.tab2 for read.
2) Next, we do an all-AMPs RETRIEVE step from test.tab1 by way of an
   all-rows scan with no residual conditions into Spool 2, which is
   duplicated on all AMPs. The size of Spool 2 is estimated to be 4
   rows. The estimated time for this step is 0.08 seconds.
3) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
   all-rows scan, which is joined to test.tab2. Spool 2 and test.tab2
   are joined using a product join, with a join condition of
   ("Spool_2.y1 = test.tab2.y2"). The result goes into Spool 1, which is
   built locally on the AMPs. The size of Spool 1 is estimated to be 2
   rows. The estimated time for this step is 1 hour and 15 minutes.
4) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 1 hour and 15 minutes.
```

Join Plan With Nested Join

Optimized Join Plan

Using nested joins, the Optimizer generates the following join plan:

Operation	Joined Tables	Total Processing Time
Spool 3:Nested Join	(Hashed table_1, index table_2)	0.24 seconds
Spool 1:rowID Join	(Hashed spool_3, direct table_2)	0.22 seconds

Completion Time

The total estimated completion time is 0.46 seconds.

The estimated performance improvement factor is 9782.

EXPLAIN Output for Optimized Join Plan

The following EXPLAIN output is generated.

```
Explanation
---------------------------------------------------------------------
1) First, we lock test.tab1 for read, and we lock test.tab2 for read.
2) Next, we do an all-AMPs RETRIEVE step from test.tab1 by way of an
   all-rows scan with no residual conditions into Spool 2, which is
   redistributed by hash code to all AMPs. Then we do a SORT to order
   Spool 2 by row hash. The size of Spool 2 is estimated to be 2 rows.
   The estimated time for this step is 0.06 seconds.
3) We do a all-AMP JOIN step from Spool 2 (Last Use) by way of an
   all-rows scan, which is joined to test.tab2 by way of unique
   index #4 "test.tab2.y2 = test.tab1.y1" extracting row ids only.
   Spool 2 and test.tab2 are joined using a nested join. The result
   goes into Spool 3, which is redistributed by hash code to all AMPs.
   Then we do a SORT to order Spool 3 by row hash. The size of Spool 3
   is estimated to be 2 rows. The estimated time for this step is 0.18
   seconds.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to test.tab2. Spool 3 and test.tab2
   are joined using a row id join. The result goes into Spool 1, which
   is built locally on the AMPs. The size of Spool 1 is estimated to be
   2 rows. The estimated time for this step is 0.22 seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 0.46 seconds.
```

Exclusion Join

Definition

An exclusion join is a product or merge join where only the rows that do not satisfy (are NOT IN) any condition specified in the request are joined.

In other words, an exclusion join finds rows in the first table that do *not* have a matching row in the second table.

Exclusion joins are an implicit form of the outer join.

SQL That Often Causes an Exclusion Join Operation

The following SQL specifications frequently cause the Optimizer to select an exclusion join.

- Use of the NOT IN logical operator in a subquery.
- Use of the EXCEPT and MINUS set operators.

Exclusion Joins and NULLABLE Columns

To avoid returning join rows that are null on the join column, use one of the following methods.

- When you create a table, define any columns that might be used for NOT IN join conditions as NOT NULL.
- When you write a query, qualify a potentially problematic join with an IS NOT NULL specification. For example,

```
WHERE Customer.CustAddress IS NOT NULL
```

Types of Exclusion Join

There are two types of exclusion join.

- Exclusion merge (see "Exclusion Merge Join" on page 2-111)
- Exclusion product (see "Exclusion Product Join" on page 2-114)

Exclusion Merge Join

Exclusion Merge Join Process

The process applied by the exclusion join algorithm is provided in the following table.

Stage	Process
1	The left and right tables are distributed and sorted based on the row hash values of the join columns.
2	For each left table row, read all right table rows having the same row hash value until one is found that matches the join condition.
3	Produce the join result. If no matching right table rows are found, return the left row.

Example 1

The following SELECT statement is an example of an exclusion merge join:

```
SELECT Name
FROM Employee
WHERE DeptNo NOT IN
 (SELECT DeptNo
  FROM Department
  WHERE Loc <> 'CHI');
```

A concrete example of the exclusion join process is documented in the following table:

Stage	Process
1	All AMPs are searched for Department rows where Loc = 'CHI'.
2	The multiple rows found to satisfy this condition, that for Department 600, is placed in a spool file on the same AMP.
3	The spool file containing the single Department row is *redistributed.*
4	The rows in the two spools undergo an exclusion *merge* join on each AMP.
5	Name information for any Employee row whose DeptNo is not 600 is placed in a result spool file on each AMP.

When the last AMP has completed its portion of the join, the contents of all result spools are sent to the user via a BYNET merge.

The steps used in an exclusion merge join are like those used in the exclusion product join example (see "Exclusion Join" on page 2-110), with the following exceptions.

- Multiple rows are retrieved in stage 2.
- Stages 2 and 4 are combined in the exclusion merge join, and redistribution occurs instead of duplication.
- Stage 3 is removed.
- Stage 5 is changed to an exclusion product.

Example 2

Consider the following EMPLOYEE and CUSTOMER tables:

Employee

Enum	Name	Job_code
PK		FK
UPI		
1	Brown	512101
2	Smith	412101
3	Jones	512101
4	Clay	412101
5	Peters	512101
6	Foster	512101
7	Gray	413201
8	Baker	512101

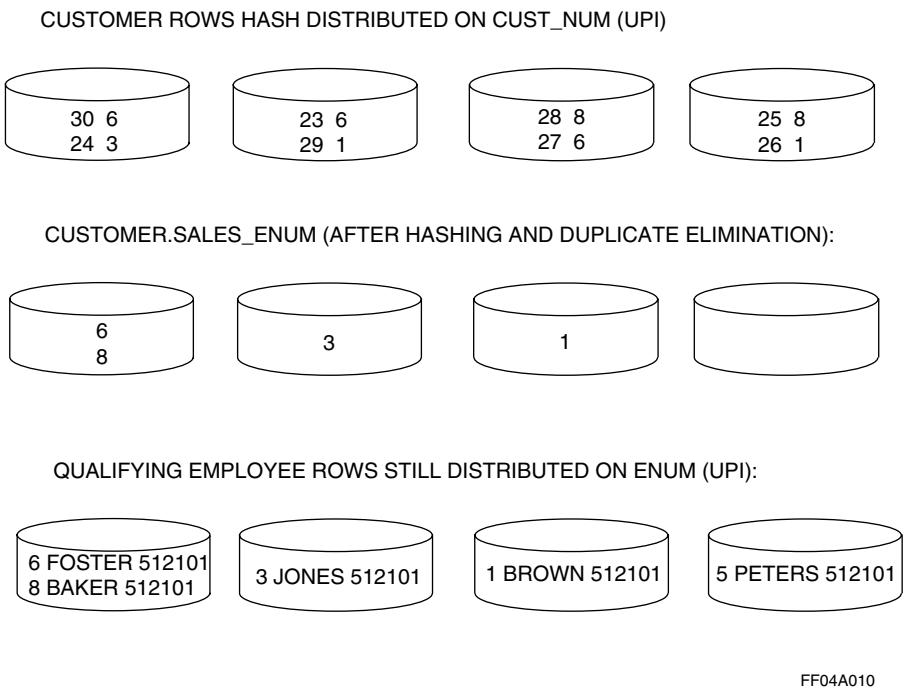
Customer

Cust	Sales_enum
PK	FK
UPI	
23	6
24	3
25	8
26	1
27	6
28	8
29	1
30	6

The graphic "Exclusion Merge Join Row Distribution" on page 2-113 shows the row redistribution caused by the following SELECT statement:

```
SELECT Name
FROM Employee
WHERE Job_Code = 51210
AND Enum NOT IN
 (SELECT Sales_Enum
  FROM Customer);
```

Exclusion Merge Join Row Distribution



Exclusion Product Join

Exclusion Product Join Process

The process applied by the exclusion product algorithm is provided in the following table:

Stage	Process
1	For each left table row, read all right table rows from the beginning until one is found that can be joined with it.
2	Produce the join result. If no matching right table rows are found, return the left row.

Example: Exclusion Product Join

The following requests names of employees who do not work in Chicago.

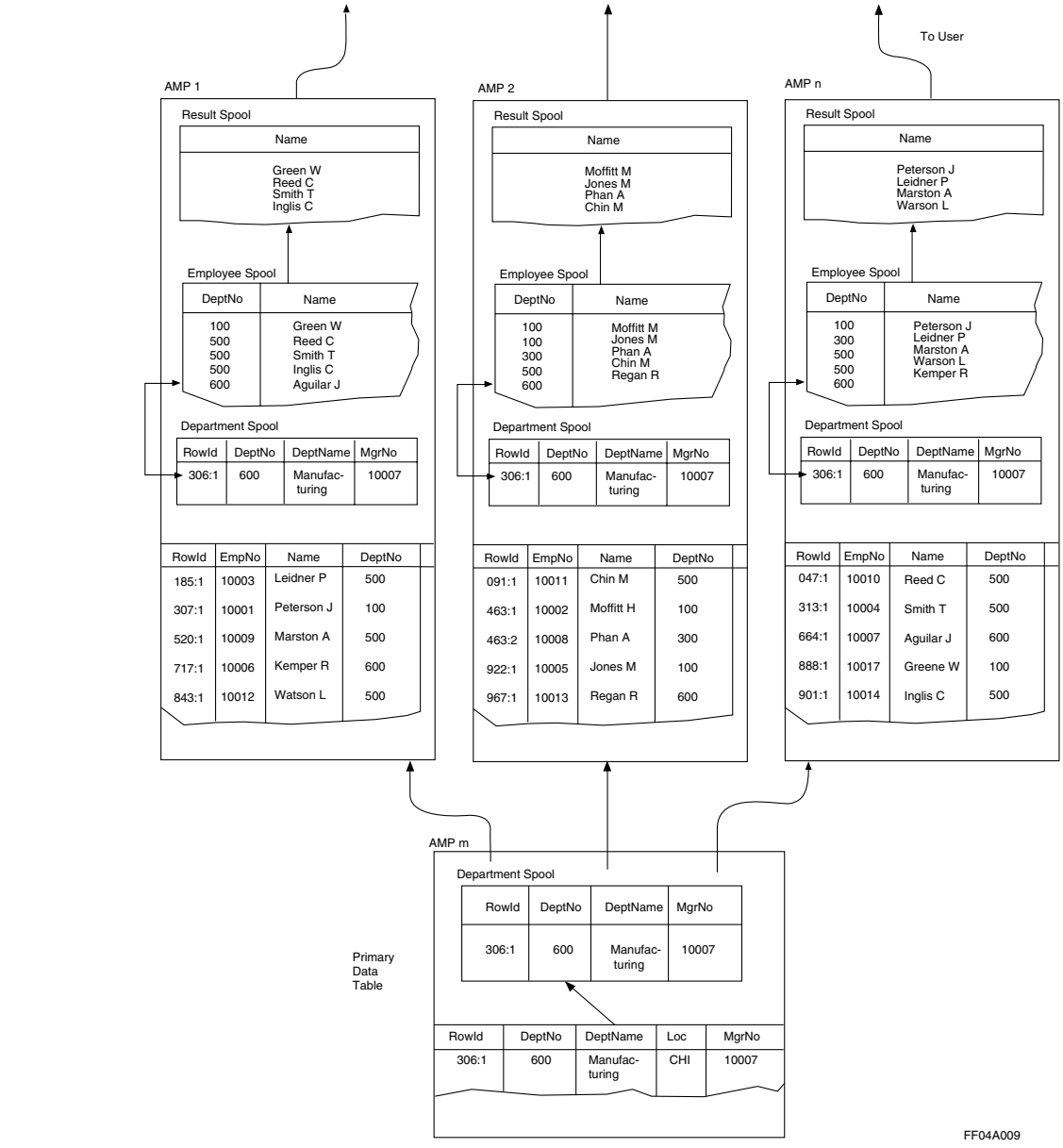
```
SELECT Name
FROM Employee
WHERE DeptNo NOT IN
 (SELECT DeptNo
  FROM Department
  WHERE Loc = 'CHI');
```

Because the subquery has only one row, an exclusion product join is used with the following process:

Stage	Process
1	All AMPs are searched for Department rows where Loc = 'CHI' • If only one AMP is selected, and if Loc is an index, then an all-AMPs retrieve is *not* performed. • The spool file containing the single Department row is *duplicated* on every AMP that contains the spooled Employee rows.
2	The single row found to satisfy this condition, that for Department 600, is duplicated right away, without being spooled in the local AMP.
3	The rows in the two spools undergo an exclusion *product* join on each AMP.
4	Name information for any Employee row whose DeptNo is not 600 is placed in a result spool file on each AMP.
5	When the last AMP has completed its portion of the join, the contents of all results spools are sent to the requesting application via a BYNET merge.

This process is illustrated by the graphic "Exclusion Product Join Operation" on page 2-115.

Exclusion Product Join Operation



Inclusion Join

Definition

An inclusion join is a product or merge join where the first right table row that matches the left row is joined.

There are two types of inclusion join.

- Inclusion merge
- Inclusion product

Inclusion Merge Join

The process applied by the inclusion merge join algorithm is provided in the following table:

Stage	Process
1	Read each row from the left table.
2	Join each left table row with the first right table row having the same hash value.

Inclusion Product Join

The process applied by the inclusion product join algorithm is provided in the following table:

Stage	Process
1	For each left table row read all right table rows from the beginning until one is found that can be joined with it.
2	Return the left row iff a matching right row is found for it.

RowID Join

Introduction

The RowID join is a special form of the nested join. The Optimizer selects a RowID join instead of a nested join when the first condition in the query specifies a literal for the first table. This value is then used to select a small number of rows which are then equijoined with a secondary index from the second table.

Rules

The Optimizer can select a RowID join only if both of the following conditions are true.

- The WHERE clause condition must match another column of the first table to a NUSI or USI of the second table.
- Only a subset of the NUSI or USI values from the second table are qualified via the join condition (this is referred to as a weakly selective index condition), and a nested join is done between the two tables to retrieve the rowIDs from the second table.

Process

Consider the following generic SQL query:

```
SELECT *
FROM table_1, table_2
WHERE table_1.NUPI = value
AND table_1.column = table_2.weakly_selective_NUSI;
```

The process involved in solving this query is illustrated in the following table.

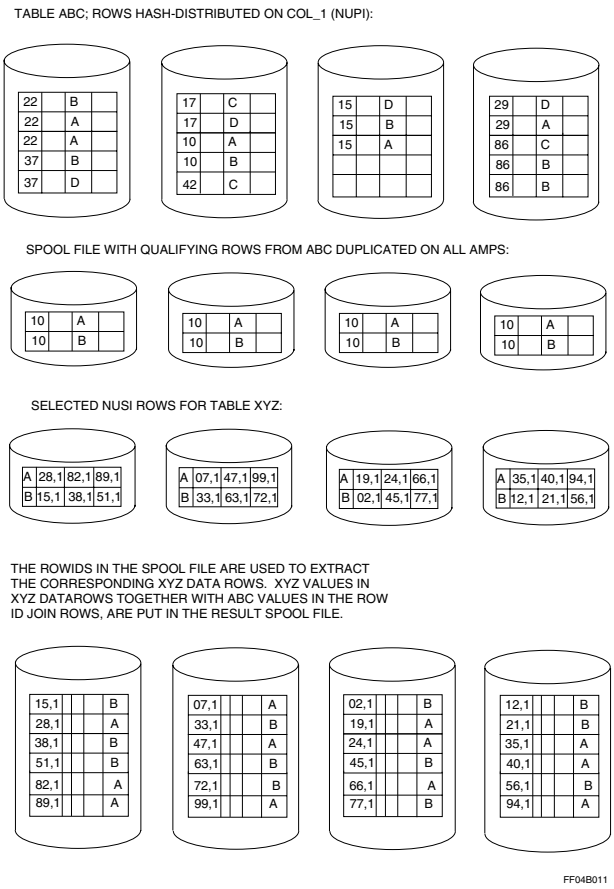
Stage	Process
1	The qualifying table_1 rows are duplicated on all AMPS.
2	The value in the join column of a table_1 row is used to hash into the table_2 NUSI (similar to a nested join).
3	The rowIDs are extracted from the index subtable and placed into a spool file together with the corresponding table_1 columns. This becomes the left table for the join.
4	When all table_1 rows have been processed, the spool file is sorted into rowID sequence.

Stage	Process
5	The rowIDs in the spool file are then used to extract the corresponding table_2 data rows.
6	table_2 values in table_2 data rows are put in the results spool file together with table_1 values in the rowID join rows.

Steps 2 and 3 are part of a nested join. Steps 4, 5, and 6 describe the rowID join.

This process is illustrated by the graphic "RowID Join" on page 2-118.

RowID Join



Example

Assume the following SELECT statement is performed. The first WHERE condition is on a NUPI and the second is on a NUSI. The Optimizer specifies a RowID join to process the join.

```
SELECT *
FROM table_1, table_2
WHERE table_1.column_1 = 10
AND table_1.column_3 = table_2.column_5;
```

Correlated Joins

Introduction

The correlated join constitute a class of join methods developed to process correlated subqueries.

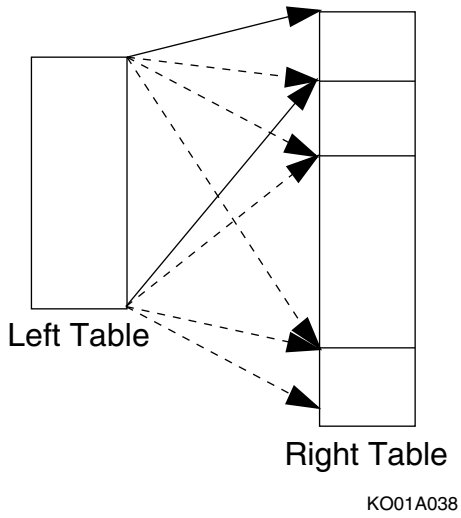
Some correlated joins are extensions of the following more general join types.

- Inclusion Merge
- Exclusion Merge
- Inclusion Product
- Exclusion Product

For each of these types the right table is a collection of groups and a left row can be returned once for each group.

Other members of the correlated join family are unique types.

The following graphic illustrates the generic correlated join process:



Correlated Join Types

There are six basic types of correlated join. Each type has an inner join version and an outer join version.

- Correlated Inclusion Merge Join

  Similar to the simple inclusion merge join (see "Inclusion Merge Join" on page 2-116) except for the handling of groups and the following additional considerations.

  - Right table rows are sorted by row hash within each group.
  - Each left table row must be merge joined with each group of the right table.
- Correlated Exclusion Merge Join

  Correlated version of standard exclusion merge join. See "Exclusion Merge Join" on page 2-111.
- Correlated Inclusion Product Join

  Correlated version of standard inclusion product join. See "Inclusion Product Join" on page 2-116.
- Correlated Exclusion Product Join

  Correlated version of standard exclusion product join. See "Exclusion Product Join" on page 2-114.
- EXISTS Join

  If a right table row exists, then return all left table rows that satisfy the condition.
- NOT EXISTS Join

  If the right table has *no* rows, then return all left table rows that do *not* satisfy the condition.

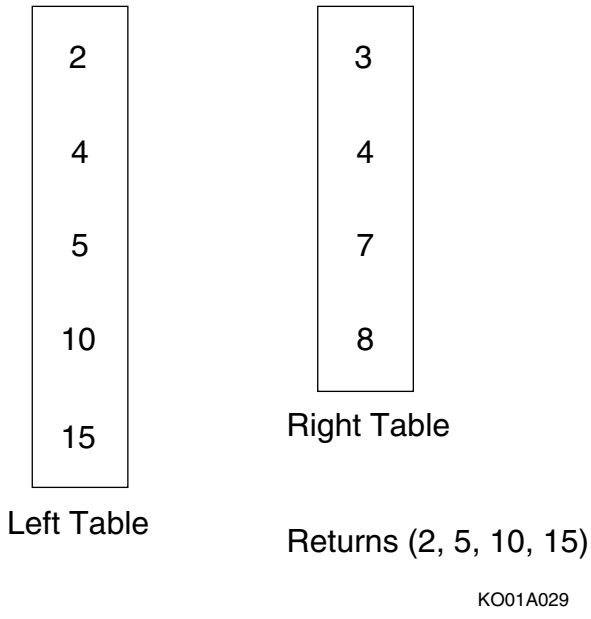
Minus All Join

Definition

The Minus All join method is used to implement MINUS, INTERSECT, and outer joins.

The process applied by the minus all join algorithm is provided in the following table:

Stage	Process
1	Distribute and sort the left and right tables based on their column_1 values.
2	For each left table row, start at the current right table row and read until a row having a value >= the left table column_1 value is found.
3	If the right table row column_1 > the left table row column_1 or if no more right table rows are found, then return the left table row.



Relational Query Optimization References

The following references are helpful for acquiring a basic understanding of how relational query optimizers work.

Topic	Reference
Origins of SQL Query Optimizers	P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of SIGMOD 1979*, 23-34, 1979. Surveys the optimizer designed for the IBM System R prototype relational system. This paper is the starting point for all commercially available cost-based SQL query optimizers. Implicitly divides query optimization into 3 components: cost estimation, plan selection, and cost comparison.
Basic Principles of Query Optimization	Surajit Chaudhuri, "An Overview of Query Optimization in Relational Systems," *Proceedings of PODS 1998*, 34-43, 1998. Presents a concise summary of the general principles of cost-based query optimization for relational database management systems. C.J. Date, *An Introduction to Database Systems* (7th ed.), Reading, MA: Addison-Wesley, 1999. This textbook presents an excellent overview of optimization in a rigorous, but less formal manner than Ullman. Goetz Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2):73-170, 1993. Extends Jarke and Koch by adding a great deal of material on optimization in a parallel relational environment. Waqar Hasan, *Optimization of SQL Queries for Parallel Machines*, Berlin: Springer-Verlag, 1996. Examines optimization problems peculiar to the parallel processing environment using the Compaq/Tandem NonStop SQL/MP query optimizer as a paradigm. The book is the Ph.D. dissertation submitted by the author to Stanford University. Matthias Jarke and Jürgen Koch, "Query Optimization in Database Systems," *ACM Computing Surveys*, 16(2):111-152, 1984. An excellent overview of the basic principles of query optimization in a relational environment.

Topic	Reference
Basic Principles of Query Optimization (continued)	David Maier, *The Theory of Relational Databases*, Rockville, MD: Computer Science Press, 1983. Like the Ullman text, this book is strongly rooted in mathematics and is not for beginners. As its title suggests, the presentation is entirely theoretical. Also like Ullman, many of the methods presented here have not been widely adapted for commercial use. Jeffrey D. Ullman, *Principles of Database Systems* (2nd ed.), Rockville, MD: Computer Science Press, 1982. A rigorously formal textbook that first provides a mathematical introduction to the principles of query optimization and then examines how these principles are adapted to the real world using the System R and INGRES query optimizers as examples. Clement T. Yu and Weiyi Meng, *Principles of Database Query Processing for Advanced Applications*, San Francisco, CA: Morgan Kaufmann Publishers, 1998. An advanced textbook covering numerous special topics in relational and object/relational query processing. The first chapter is an excellent introductory overview of relational query optimization.
Statistical Profiles of Data Demographics	Michael V. Mannino, Paicheng Chu, and Thomas Sager, "Statistical Profile Estimation in Database Systems," *ACM Computing Surveys*, 20(3):191-221, 1988. A basic overview of relational database statistics: how they are gathered and how they are used.
Interval Histograms	Yannis E. Ioannidis and Stavros Christodoulakis, "Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results," *ACM Transactions on Database Systems*, 18(4):709-448, 1993. Demonstrates that high-biased interval histograms are always optimal for estimating join cardinalities for a commonly used form of equijoin. Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *Proceedings of SIGMOD 1996*, 294-305, 1996. Provides a capsule review of older types of interval histograms used in relational systems and introduces a few new types, including compressed histograms.
High-Biased Interval Histograms	Philip B. Gibbons and Yossi Matias, "Synopsis Data Structures for Massive Data Sets," in: *External Memory Systems: DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, 1998. A survey of synopsis data structures for extremely large databases. This laboratory developed the concept of high-biased histograms as a method of compensating for extreme skew when equal-height histograms are the method of choice for a query optimizer. The histogram methods used by Teradata were developed by this group.

Chapter 3:

Join Optimizations

This chapter describes several join optimizations, including the following topics.

- Star and snowflake join optimization

  For additional information, see:

  - “Star and Snowflake Join Optimization” on page 3-3
  - “Miscellaneous Considerations for Star Join Optimization” on page 3-10
  - “Selecting Indexes for Star Joins” on page 3-12
- Classes of large table/small table joins

  For additional information, see:

  - “LT/ST-J1 Indexed Joins” on page 3-8
  - “LT-ST-J2 Unindexed Joins” on page 3-9
- Reasonable indexed joins (LT/ST-J1a)

  For additional information, see:

  - “Reasonable Indexed Join Plan Without Star Join Optimization” on page 3-18
  - “Reasonable Indexed Join Plan With Star Join Optimization: Large Table Primary Index Joined to Small Tables” on page 3-20
  - “Reasonable Indexed Join Plan With Star Join Optimization: Large Table USI Joined to Small Tables” on page 3-22
  - “Reasonable Indexed Join Plan With Star Join Optimization: Large Table NUSI Joined to Small Tables” on page 3-24
- Reasonable unindexed joins (LT/ST-J2a)

  For additional information, see:

  - “Reasonable Unindexed Join Without Join Optimization” on page 3-29
  - “Reasonable Unindexed Join With Join Optimization” on page 3-31

- Joins Using Join Indexes

  For additional information, see:

  - “Join Indexes” on page 3-33
  - “Maintaining Join Index During INSERT” on page 3-44
  - “General Method of Maintaining Join Index During DELETE” on page 3-41
  - “Optimized Method of Maintaining Join Index During DELETE” on page 3-43
  - “Maintaining Join Index During INSERT” on page 3-44
  - “General Method of Maintaining Join Index During UPDATE” on page 3-46
  - “Optimized Method of Maintaining Join Index During UPDATE” on page 3-48

The performance-oriented aspects of these topics refers only to how the Teradata SQL parser works, not how to design your system to make the work of the parser more efficient.

More information about how to optimize the performance of your database from a database design perspective can be found in the following books:

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

Star and Snowflake Join Optimization

Introduction

Star and snowflake joins are terms used to describe various large table/small table joins.

The following concepts apply when optimizing star and snowflake joins:

- Large and small are relative terms.

  Generally the ratio of the cardinalities of the large table to each of the small tables ranges from 100:1 to 1 000:1.

  Note that these cardinality ratios apply to the results tables being joined, not to the base tables before they are reduced by predicate qualifications.
- The join plan in which all small tables are joined first is called the product/merge join plan because the small tables are usually joined via a product join and the joined result of the small tables is usually joined to the large table via a merge join.

Simple binary or ternary joins of a large results table with small results tables are not treated as star joins by the Optimizer, so you should not expect to see any mention of star joins in the EXPLAIN text for queries involving them. Note that the EXPLAIN text typically refers to star and snowflake joins as LT/ST, or large table/small table joins.

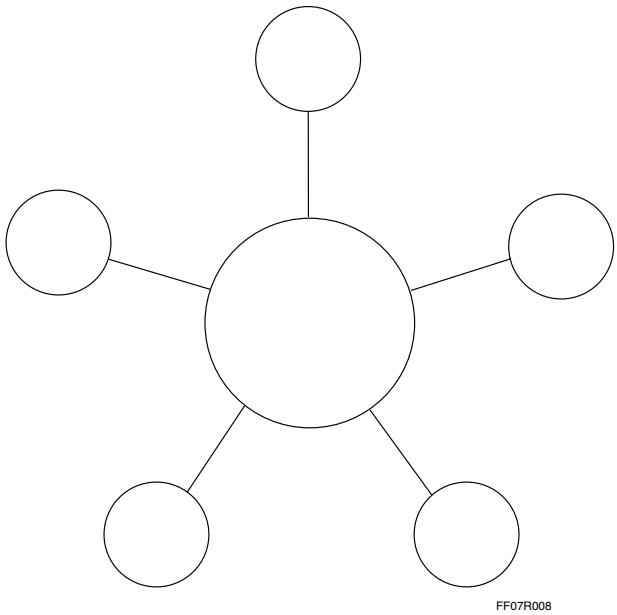
Definition: Star Join

A star join is a join where three or more relations with relatively small cardinality are joined to another relation having a much larger relative cardinality. In dimensional modeling terminology, the large table is called a fact table and the smaller tables are referred to as dimension tables.

The difference between a large relation and a small relation is defined in terms of cardinality ratios on the order of 100:1 at minimum. The smaller relations in a star join do not necessarily derive from base tables having a smaller cardinality; they can be an intermediate result of having applied a condition to a medium- or large-sized base table.

The term star derives from the pictorial representation of such a join, which resembles a childlike drawing of a star, as seen in the following graphic.

As you can see, the graphic representation of such a join resembles a crude drawing of a star.



The concept of star schemas in physical database design is described in *Teradata RDBMS Database Design*.

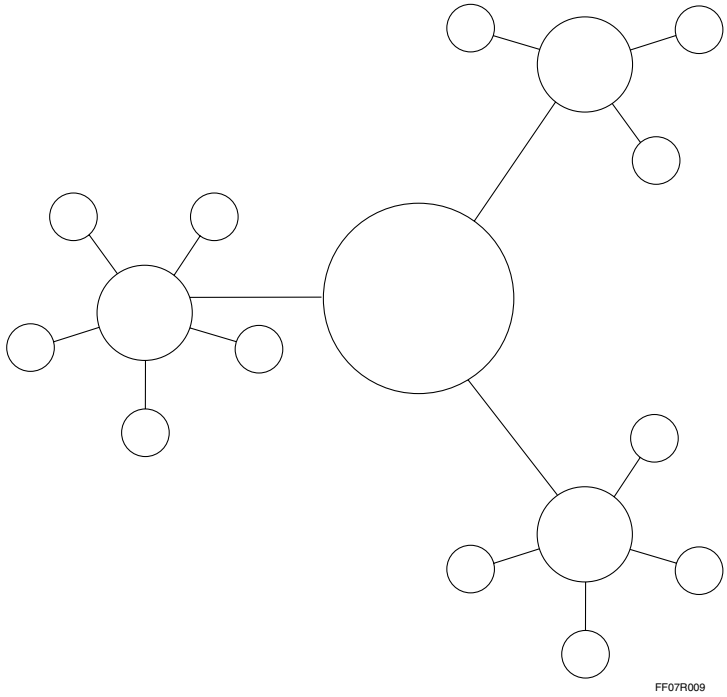
Definition: Snowflake Join

A snowflake join is a join where a large table is joined with three or more smaller base tables or join relations, some or all of which themselves are joined to three or more smaller base tables or join relations. Another way of looking at a snowflake is to think of it as a star with normalized dimension tables.

The concept of snowflake schemas in physical database design is described in *Teradata RDBMS Database Design*.

As with a star join, the cardinality ratios determine whether a table or join relation is large or small. The minimum cardinality ratio defining a large:small table relationship is 100:1.

As you can see, the graphic representation of such a join resembles a crude drawing of a snowflake.



Primary Target of a Star Join

Star joins are a fundamental component of a dimensional database model (see "Dimensional Modeling, Star, and Snowflake Schemas" in the chapter "The Normalization Process" in *Teradata RDBMS Database Design* for more information about dimensional modeling). Therefore, the primary target of star join processing is the product join of numerous small relations to build a composite column set that can be used to access a large relation directly as either its primary index or a secondary index.

Secondary Targets of a Star Join

The star join method is sometimes useful for handling the following situations:

- Complex cases involving multiple large relations.
- A product join of several small relations in order to permit a merge join with a locally spooled large relation when there is no available index.

Star Join Optimization

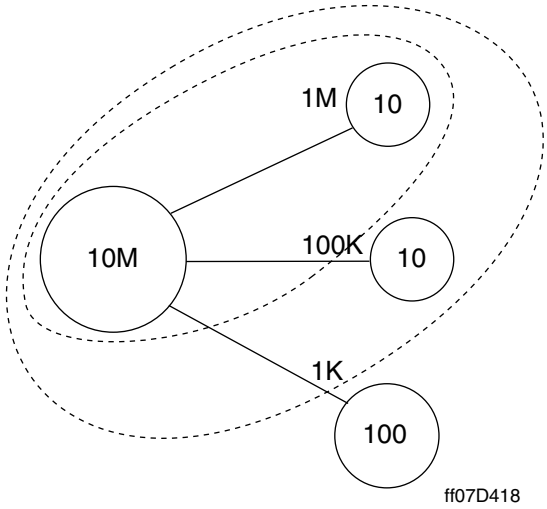
With star join optimization, the Optimizer searches for a better join plan in which all the small tables are joined first, after which the resulting relation is joined with the large table. The Optimizer then uses the join plan that has the lowest estimated cost.

Without star join optimization, the Optimizer does an adequate job joining one or two small tables to a large table. However, when joining three or more small tables to one large table, the Optimizer usually generates a join plan in which a small table (or the join result of small tables) is joined directly with the large table.

Example Star Join Optimization

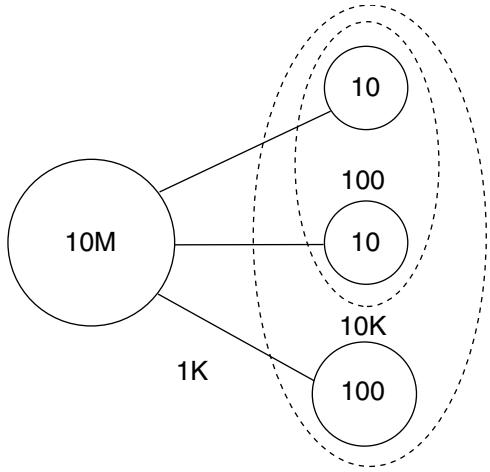
The following graphic illustrates a nonoptimal star join plan of four tables. The relative cardinalities of the tables are given as integers within each table (represented in each case by a circle). The relative cost of each join is given as an integer number on the line connecting the joined relations.

In this example, the first join is between the large table and one of the small tables. The relative cost of this join is 1×10^6. The next join is between this joined relation and another of the small tables. Its relative cost is 1×10^5. Finally, this relation is joined with the last small table at a relative cost of 1×10^3.



The next graphic presents an optimized join plan for the same set of tables. This plan uses a compromise join of all the unconnected (small) tables prior to making the join to the large table.

The first join has a relative cost of 1 x 10^2, the second a cost of 1 x 10^4, and the final join between the large table and its joined small tables relation has a cost of 1 x 10^3.



Cost of nonoptimized star join $= 1\times10^6 + 1\times10^5 + 1\times10^3 = 1\ 101\ 000$

Cost of optimized star join $= 1\times10^2 + 1\times10^4 + 1\times10^3 = 11\ 100$

The results indicate that for this example, the optimized star join plan is two orders of magnitude cheaper than the nonoptimized star join plan for the same four tables.

Star Join Categories

For purposes of the following descriptions, star joins are referred to as LT/ST joins. LT/ST joins always fall into one of the following categories:

- Indexed Joins, LT/ST-J1

  For more detailed information on LT/ST-J1 joins, see "LT/ST-J1 Indexed Joins" on page 3-8.
- Unindexed Joins, LT/ST-J2

  For more detailed information on LT/ST-J2 joins, see "LT-ST-J2 Unindexed Joins" on page 3-9.

LT/ST-J1 Indexed Joins

Definition

In the LT/ST-J1 class index join, some combination of the join columns of the small tables comprises an index of the large table.

Reasonable Indexed Joins, LT/ST-J1a

This subclass consists of those LT/ST-J1 joins in which the cardinality of the Cartesian product of the small tables is small relative to the cardinality of the large table.

The magnitude of this cardinality difference cannot be defined rigorously; it depends on factors such as the following.

- Types of indexes defined
- Conditions used by the old join plan
- Conditions used by the new join plan
- Size of the columns retrieved from the small tables

Unreasonable Indexed Joins, LT/ST-J1b

This subclass consists of all LT/ST-J1 joins that are not of subclass LT/ST-J1a (Reasonable Indexed Joins).

LT-ST-J2 Unindexed Joins

Definition

In the LT/ST-J2 class unindexed join, no combination of the join columns of the small tables comprises any index of the large table.

Reasonable Unindexed Joins, LT/ST-J2a

This subclass consists of those LT/ST-J2 joins in which the cardinality of the Cartesian product of the small tables is much smaller than the cardinality of the large table.

The difference between the cardinalities of the small tables Cartesian product and the cardinality of the large table is much larger for LT/ST-J2a joins than for LT/ST-J1a joins, though it cannot be defined rigorously.

Unreasonable Unindexed Joins, LT/ST-J2b

This subclass consists of all LT/ST-J2 joins that are not of subclass LT/ST-J2a.

Miscellaneous Considerations for Star Join Optimization

Introduction

This section describes several miscellaneous considerations for star join optimization.

Statistics

Join planning is based on the estimated cardinalities of the results tables. The cardinalities usually cannot be precisely estimated without accurate statistics.

The cardinality of a star join is estimated based on the cardinality of the small table join result and the selectivity of the collection of large table join columns. Therefore, to guarantee a good join plan for queries involving star joins, the following usage considerations apply:

- Statistics must be collected for all the tables on their primary indexes, as well as for each index used in the query.
- If constraints are specified on nonindexed columns, statistics must be collected on these columns as well.

Avoiding Hash Synonyms

Depending on the columns making up the primary index, hash synonyms may occur. Hash synonyms, which usually occur when the primary index is composed of only small integer columns, always degrade query performance.

Changing Data Types to Enhance Performance

If possible, design your tables and queries so that joined fields are from the same domain (of the same data type), and if numeric, of the same size. If the joined fields are of different data types (and different sizes, if numeric), changing the type definition of one of the tables should improve join performance.

If there is no join condition specified on any index, neither table need be changed. In such cases, if the same data types are specified on the joined fields, the primary index might be used for an intermediate join result, thus eliminating the need for rehashing.

If, however, an index can be used in the join, and if some fields of the index are of smaller size, then one of the tables may have to be changed. To improve performance, it is frequently better to change the *smaller* table to define its join columns using the data type of the *larger* table.

For example, consider the following join condition, assuming that table_1.NUPI is typed SMALLINT and table_2.NUPI is typed INTEGER.

```
table_1.NUPI = table_2.NUPI
```

If table_1 is the larger table, you should consider changing table_2.NUPI to type SMALLINT. However, if table_2 is the larger table, consider changing table_1.NUPI to type INTEGER.

Changing Conditional Expressions to Use one Index Operand

If one side of a join condition combines expressions and indexing, performance is generally not as good as if just the index is one operand. Consider modifying the equality to isolate the index on one side, exclusive of any expressions.

For example, consider the following conditional expressions. Note that the first way the condition is stated uses the primary index of table_2, table_2.NUPI, in an expression, while the second condition separates the primary index of table_2 from the expression, moving it to the other side of the equality condition.

```
table_1.x = table_2.NUPI - 1

table_1.x + 1 = table_2.NUPI
```

Selecting Indexes for Star Joins

Introduction

This topic provides guidelines for selecting indexes for use in star joins. These are rules of thumb only—a more complete performance model is required in order to select indexes that optimize the entire mix of join queries on the table.

Create Indexes on Join Columns for Each Star Join

The performance of a star join can be improved only if an index is created on the collection of some join columns of the large table so that redistributing and sorting of the large table are avoided. If a large table is involved in more than one star join, you should create an index on the collection of some join columns associated with each star join.

For example, if the large table Widgets is joined with the small tables Color, Shape, and Size using the collection of columns (color, shape, size) in one star join, and with the small tables Period, State, and Country using the collection of columns (period, state, country) in another star join, then you can create the following indexes on Widgets to be used in those star joins.

- Primary index on (color, shape, size).
- Nonunique secondary index on (period, state, country).

Criteria for Selecting an Index Type

You must decide the type of index that is to be created on each collection of join columns of a large table. When making that decision, consider the following guidelines.

- A primary index is the best index for star joins.
- Each table can have only one primary index.
- The Optimizer does not use USIs and NUSIs for star joins when the estimated number of rows to be retrieved is large.

  Applications of NUSIs and USIs for star joins are limited, so always verify that when an index is created on a large table, it will be used by the Optimizer.

  If a NUSI or USI *is* used, the rowIDs are retrieved via a nested join, after which a rowID join is used to retrieve the data rows. Note that rowID joins are sometimes very ineffective.

Criteria When Any Join Column Can Be the Primary Index

If any of the collections of join columns meets the criteria for a good candidate primary index (that is, has enough unique values to guarantee that the large table is distributed evenly across the AMPs), then you should consider the following guidelines.

IF LT/ST joins are . . .	THEN the primary index should be created using the . . .
used with the same frequency	star join that results in the most number of rows selected. This leaves the star joins that select fewer rows for the NUSIs and USIs.
not used with the same frequency	collection of join columns associated with the most often used star join.

For example, if the star join between Widgets and (period, state, country) is used more often than the star join between Widgets and (color, shape, size), the primary index should be created on (period, state, country).

However, if the former join selects far fewer number of rows than the latter join, it may be better to associate the primary index with the latter join (on columns color, shape, and size).

Performance Modeling: Optimizing All Join Queries

To optimize the entire mix of join queries, you should design a more complete performance model for your database.

For example, a user might have a relatively short star join query that is used more frequently than an extremely long query.

In such cases, it might be better to select the *primary* index favoring the long query, even though the guidelines indicate otherwise. This is because the benefit of the long query may be very great compared to the cost of the short query, and the combination of joins results in a net gain in performance.

Not all join columns of the small tables must join with the large table index in an LT/ST-J1a (Reasonable Indexed) star join.

Using a Common Set of Join Columns in the Primary Index

If more than one combination of the large table columns is used in different star joins, and if the combinations are overlapping, then the primary index should consist of the common set of these combinations (if the set is qualified for the primary index).

This has two advantages.

- Fewer indexes are required
- More than one star join can share the same primary index

For example, assume that the following conditions are true.

- The collection of columns (color, shape, size) of the large table Widgets is joined with the small tables (color, shape, and size) of a star join
- The collection of columns (shape, size, and period) is joined with the small tables (shape, size, and period) of another star join

In this case, the primary index of Widgets should consist of the columns (shape, size) if the set is qualified for the primary index.

Star Join Examples

Introduction

The sections that follow provide examples of these join expression types.

- LT/ST-J1a (Reasonable Indexed)
- LT/ST-J2a (Reasonable Unindexed)
- Nested join

How the Examples Are Structured

For each type of query, two summaries of the join plans and estimated execution times are provided—one with and the other without star join optimization.

To be consistent with the EXPLAIN output, if the input table of a join is the result of a previous join, the cost of preparing the input table for the join is included in the cost of performing the previous join. Otherwise, the preparation cost is included into the cost of performing a join.

The total estimated cost for each query is taken directly from the EXPLAIN outputs, which take into account the parallelism of steps.

Costs are relative, and vary depending on the number of AMPs in the configuration. The example costs given are for a system with two AMPs.

The estimated percentage of performance improvement is provided for each example. Remember that these percentages are achieved only when the same join examples are performed under the identical conditions.

Other queries may achieve more or less performance improvement, depending on the join conditions and table statistics, but the general trends are consistently in the same direction.

The following table definitions are used in the examples.

Small Table Definitions

Table name	Columns	Primary index
Color	code, description	description
Size	code, description	description
Options	code, description	description

Large Table Definition

Table name	Columns	Cardinality
Widgets	color, size, options, units, period	1 000 000 rows

Star Join Example Types

Examples are provided for the following Large Table/Small Table join expression types.

- "Reasonable Indexed Join Plan Without Star Join Optimization"
- "Reasonable Indexed Join Plan With Star Join Optimization: Large Table Primary Index Joined to Small Tables"
- "Reasonable Indexed Join Plan With Star Join Optimization: Large Table USI Joined to Small Tables"
- "Reasonable Indexed Join Plan With Star Join Optimization: Large Table NUSI Joined to Small Tables"
- "Join Plan With Star Join Optimization: Large Table Subquery Join"
- "Reasonable Unindexed Join Without Join Optimization"
- "Reasonable Unindexed Join With Join Optimization"

Cardinality and Uniqueness Statistics for the Reasonable Indexed Join Examples

Small Table Cardinality Statistics

For all LT/ST-J1a (Reasonable Indexed Joins) examples, the following row information is given for the small tables:

Attribute	Cardinality
color	2
size	10
options	10

Large Table Uniqueness Statistics

The following statistics information is given for the large table:

Column Name	Number of Unique Values
color	10
size	100
options	1000

Test Query

These examples explain the join plans and estimated time for execution of the following query, when different types of indexes (primary index, unique secondary index, nonunique secondary index) are created on the various join columns (color, size, and options) of the large table.

```
SELECT …
WHERE widgets.color = color.code
AND widgets.size = size.code
AND widgets.options = options.code;
```

Reasonable Indexed Join Plan Without Star Join Optimization

Unoptimized Join Plan

Without join optimization, the following join plan is generated independently of the type of index created on the collection of join columns (color, size, and options) of the large table:

Operation	Joined Tables	Total Processing Time (seconds)
Spool 4: Product Join	duped options, direct size	2.67
Spool 5: Product Join	duped color, direct widgets	6 660.00[a]
Spool 1: Merge Join	duped 4, local 5	7.43

a. 1 hour, 51 minutes.

Completion Time

Note that the total estimated completion time, including time for two product joins and a merge join, is 1 hour 52 minutes.

EXPLAIN Output for Unoptimized Join Plan

The following EXPLAIN output is generated without star join optimization, independently of the type of index created on the collection of join columns (color, size, and options) of the large table.

```
Explanation
------------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we execute the following steps in parallel.
   a) We do an all-AMPs RETRIEVE step from TEST.Color by way of
      an all-rows scan with no residual conditions into Spool 2, which
      is duplicated on all AMPs. The size of Spool 2 is estimated to be
      4 rows. The estimated time for this step is 0.08 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Options by way of an
      all-rows scan with no residual conditions into Spool 3, which is
      duplicated on all AMPs. The size of Spool 3 is estimated to be 20
      rows. The estimated time for this step is 0.24 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from TEST.Size by way of an all-rows
      scan with no residual conditions, which is joined to Spool 3
      (Last Use). TEST.Size and Spool 3 are joined using a product
      join. The result goes into Spool 4, which is duplicated on all
      AMPs. Then we do a SORT to order Spool 4 by row hash. The size
      of Spool 4 is estimated to be 200 rows. The estimated time for
      this step is 2.43 seconds.
   b) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of
      an all-rows scan, which is joined to TEST.Widgets. Spool 2 and
      TEST.Widgets are joined using a product join, with a join
      condition of ("TEST.Widgets.color = Spool_2.code"). The result
      goes into Spool 5, which is built locally on the AMPs. Then we do
      a SORT to order Spool 5 by row hash. The size of Spool 5 is
      estimated to be 200,000 rows. The estimated time for this step is
      1 hour and 51 minutes.
4) We do an all-AMPs JOIN step from Spool 4 (Last Use) by way of an
   all-rows scan, which is joined to Spool 5 (Last Use). Spool 4 and
   Spool 5 are joined using a merge join, with a join condition of
   ("(Spool_5.size=Spool_4.code)AND(Spool_5.options=Spool_4.code)").
   The result goes into Spool 1, which is built locally on the AMPs. The
   size of Spool 1 is estimated to be 200 rows. The estimated time for
   this step is 7.43 seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 1 hour and 52 seconds.
```

Reasonable Indexed Join Plan With Star Join Optimization: Large Table Primary Index Joined to Small Tables

Optimized Join Plan

With join optimization, the following join plan is generated when the collection of join columns (color, size, and options) makes up the primary index of the large table.

Operation	Joined Tables	Total Processing Time (seconds)
Spool 3: Product Join	duped color, direct options	0.31
Spool 5: Product Join	duped size, direct 3	1.62
Spool 1: Merge Join	hashed 5, direct widgets	4.09

Completion Time

Total estimated execution time is 5.80 seconds.

The estimated performance improvement factor is 1158.

EXPLAIN Output for Optimized Join Plan

The EXPLAIN output for this optimized join plan is as follows.

```
Explanation
-----------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we    lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we do an all-AMPs RETRIEVE step from TEST.Color by way of an
   all-rows scan with no residual conditions into Spool 2, which is
   duplicated on all AMPs. The size of Spool 2 is estimated to be 4
   rows. The estimated time for this step is 0.08 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
      all-rows scan, which is joined to TEST.Options. Spool 2 and
      TEST.Options are joined using a product join. The result goes
      into Spool 3, which is built locally on the AMPs. The size of
      Spool 3 is estimated to be 20 rows. The estimated time for this
      step is 0.23 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Size by way of an
      all-rows scan with no residual conditions into Spool 4, which is
      duplicated on all AMPs. The size of Spool 4 is estimated to be 20
      rows. The estimated time for this step is 0.24 seconds.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a product join. The result goes into
   Spool 5, which is redistributed by hash code to all AMPs. Then we do
   a SORT to order Spool 5 by row hash. The size of Spool 5 is
   estimated to be 200 rows. The estimated time for this step is
   1.38 seconds.
5) We do an all-AMPs JOIN step from TEST.Widgets by way of an all-rows
   scan with no residual conditions, which is joined to Spool 5 (Last
   Use). TEST.Widgets and Spool 5 are joined using a merge join, with a
   join condition of ("(TEST.Widgets.size = Spool_5.code) AND
   ((TEST.Widgets.color = Spool_5.code) AND (TEST.Widgets.options =
   Spool_5.code ))"). The result goes into Spool 1, which is built
   locally on the AMPs. The size of Spool 1 is estimated to be 200 rows.
   The estimated time for this step is 4.09 seconds.
6) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 5.80 seconds.
```

Reasonable Indexed Join Plan With Star Join Optimization: Large Table USI Joined to Small Tables

Optimized Join Plan

With join optimization, the following join plan is generated when the collection of join columns (color, size, and options) makes up a unique secondary index of the large table:

Operation	Joined Tables	Total Processing Time (seconds)
Spool 3: Product Join	duped color, direct options	0.31
Spool 5: Product Join	duped size, direct 3	1.62
Spool 6: Nested Join	hashed 5, index widgets	2.71
Spool 1: rowID Join	hashed 6, index widgets	5.65

Completion Time

The total estimated time is 10.07 seconds.

The estimated performance improvement factor is 667.

EXPLAIN Output for Optimized Join Plan

The EXPLAIN output for this optimized join plan is as follows.

```
Explanation
-----------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we do an all-AMPs RETRIEVE step from TEST.Color by way of an
   all-rows scan with no residual conditions into Spool 2, which is
   duplicated on all AMPs. The size of Spool 2 is estimated to be 4
   rows. The estimated time for this step is 0.08 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
      all-rows scan, which is joined to TEST.Options. Spool 2 and
      TEST.Options are joined using a product join. The result goes
      into Spool 3, which is built locally on the AMPs. The size of
      Spool 3 is estimated to be 20 rows. The estimated time for this
      step is 0.23 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Size by way of an
      all-rows scan with no residual conditions into Spool 4, which is
      duplicated on all AMPs. The size of Spool 4 is estimated to be 20
      rows. The estimated time for this step is 0.24 seconds.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a product join. The result goes into Spool
   5, which is redistributed by hash code to all AMPs. Then we do a
   SORT to order Spool 5 by row hash. The size of Spool 5 is estimated
   to be 200 rows. The estimated time for this step is 1.38 seconds.
5) We do a all-AMP JOIN step from Spool 5 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets by way of unique
   index # 4 extracting row ids only. Spool 5 and TEST.Widgets are
   joined using a nested join. The result goes into Spool 6, which is
   redistributed by hash code to all AMPs. Then we do a SORT to order
   Spool 6 by row hash. The size of Spool 6 is estimated to be 200
   rows. The estimated time for this step is 2.71 seconds.
6) We do an all-AMPs JOIN step from Spool 6 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets. Spool 6 and
   TEST.Widgets are joined using a row id join. The result goes into
   Spool 1, which is built locally on the AMPs. The size of Spool 1 is
   estimated to be 200 rows. The estimated time for this step is 5.65
   seconds.
7) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 10.07 seconds.
```

Reasonable Indexed Join Plan With Star Join Optimization: Large Table NUSI Joined to Small Tables

Optimized Join Plan

With join optimization, the following join plan is generated when the collection of join columns (color, size, and options) makes up a nonunique secondary index of the large table:

Operation	Joined Tables	Total Processing Time (seconds)
Spool 3: Product Join	duped color, direct options	0.31
Spool 5: Product Join	duped size, direct 3	4.43
Spool 1: Nested Join	duped 5, index widgets	22.73

Completion Time

The total estimated execution time is 27.26 seconds.

The estimated performance improvement factor is 246.

EXPLAIN Output for Optimized Join Plan

The EXPLAIN output for this optimized join plan is as follows.

```
Explanation
------------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we do an all-AMPs RETRIEVE step from TEST.Color by way of an
   all-rows scan with no residual conditions into Spool 2, which is
   duplicated on all AMPs. The size of Spool 2 is estimated to be 4
   rows. The estimated time for this step is 0.08 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
      all-rows scan, which is joined to TEST.Options. Spool 2 and
      TEST.Options are joined using a product join. The result goes
      into Spool 3, which is built locally on the AMPs. The size of
      Spool 3 is estimated to be 20 rows. The estimated time for this
      step is 0.23 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Size by way of an
      all-rows scan with no residual conditions into Spool 4, which is
      duplicated on all AMPs. The size of Spool 4 is estimated to be 20
      rows. The estimated time for this step is 0.24 seconds.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a product join. The result goes into Spool
   5, which is duplicated on all AMPs. The size of Spool 5 is estimated
   to be 400 rows. The estimated time for this step is 4.19 seconds.
5) We do an all-AMPs JOIN step from Spool 5 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets by way of index # 4.
   Spool 5 and TEST.Widgets are joined using a nested join. The result
   goes into Spool 1, which is built locally on the AMPs. The size of
   Spool 1 is estimated to be 200 rows. The estimated time for this
   step is 22.73 seconds.
6) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 27.26 seconds.
```

Join Plan With Star Join Optimization: Large Table Subquery Join

Example Query

The following query is used for this example.

```
SELECT …
WHERE Widgets.color=COLOR.code
AND Widgets.size=SIZE.code
AND Widgets.options IN (SELECT OPTIONS.code);
```

Optimized Join Plan

With join optimization, the following join plan is generated when the collection of join columns (color, size, and options) makes up a nonunique secondary index of the large table:

Operation	Joined Tables	Total Processing Time (seconds)
Spool 4: Product Join	duped color, direct size	0.31
Spool 6: Product Join	local 4, duped options	4.46
Spool 1: Nested Join	duped 6, index widgets	22.73

Completion Time

The total estimated completion time is 27.40 seconds.

The estimated performance improvement factor is 245.

EXPLAIN Output for Optimized Join Plan

The EXPLAIN output for this optimized join plan is as follows.

```
Explanation
-----------------------------------------------------------------------
1) First, we lock TEST.Options for read, we lock TEST.Color for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we execute the following steps in parallel.
   a) We do an all-AMPs RETRIEVE step from TEST.Options by way of
      an all-rows scan with no residual conditions into Spool 2, which
      is redistributed by hash code to all AMPs. Then we do a SORT to
      order Spool 2 by the sort key in spool field1 eliminating
      duplicate rows. The size of Spool 2 is estimated to be 10 rows.
      The estimated time for this step is 0.19 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Color by way of an
      all-rows scan with no residual conditions into Spool 3, which is
      duplicated on all AMPs. The size of Spool 3 is estimated to be 4
      rows. The estimated time for this step is 0.08 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
      all-rows scan, which is joined to TEST.Size. Spool 3 and
      TEST.Size are joined using a product join. The result goes into
      Spool 4, which is built locally on the AMPs. The size of Spool 4
      is estimated to be 20 rows. The estimated time for this step is
      0.23 seconds.
   b) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of
      an all-rows scan into Spool 5, which is duplicated on all AMPs.
      The size of Spool 5 is estimated to be 20 rows. The estimated time
      for this step is 0.27 seconds.
4) We do an all-AMPs JOIN step from Spool 4 (Last Use) by way of an
   all-rows scan, which is joined to Spool 5 (Last Use). Spool 4 and
   Spool 5 are joined using a product join. The result goes into Spool
   6, which is duplicated on all AMPs. The size of Spool 6 is estimated
   to be 400 rows. The estimated time for this step is 4.19 seconds.
5) We do an all-AMPs JOIN step from Spool 6 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets by way of index # 4.
   Spool 6 and TEST.Widgets are joined using a nested join. The result
   goes into Spool 1, which is built locally on the AMPs. The size of
   Spool 1 is estimated to be 200 rows. The estimated time for this
   step is 22.73 seconds.
6) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 27.40 seconds.
```

Cardinality and Uniqueness Statistics for the Reasonable Unindexed Join Examples

Small Table Cardinality Statistics

For LT/ST-J2a (Reasonable Unindexed) join example, the following row information is given for the small tables.

Attribute	Cardinality (rows)
color	3
size	10
options	10

Large Table Uniqueness Statistics

The following statistics information is given for the large table.

Column Name	Number of Unique Values
color	6
size	30
options	300

No index is created on the collection of join columns (color, size, and options) of the large table.

Test Query

The following query is used for the LT/ST-J2a (Reasonable Unindexed) join in this example:

```
SELECT *
FROM widget, color, size, options
WHERE widgets.color = color.code
AND widgets.size = size.code
AND widgets.options = options.code
AND size.description = options.description;
```

Reasonable Unindexed Join Without Join Optimization

Unoptimized Join Plan

Without star join optimization, the following join plan is generated.

Operation	Joined Tables	Total Processing Time (seconds)
Spool 3: Merge Join	direct options, direct size	0.46
Spool 4: Product Join	duped color, direct widgets	12 491.00[a]
Spool 1: Merge Join	duped 3, local 4	21.05

a. 3 hours, 28 minutes, 11 seconds.

Completion Time

The total estimated completion time is 3 hours 28 minutes.

EXPLAIN Output for Unoptimized Join Plan

The following EXPLAIN output is generated without LT/ST optimization.

```
Explanation
-----------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we execute the following steps in parallel.
   a) We do an all-AMPs RETRIEVE step from TEST.Color by way of an
      all-rows scan with no residual conditions into Spool 2, which is
      duplicated on all AMPs. The size of Spool 2 is estimated to be 6
      rows. The estimated time for this step is 0.11 seconds.
   b) We do an all-AMPs JOIN step from TEST.Options by way of an
      all-rows scan with no residual conditions, which is joined to
      TEST.Size. TEST.Options and TEST.Size are joined using a merge
      join, with a join condition of ("TEST.Widgets.description =
      TEST.Options.description"). The result goes into Spool 3, which
      is duplicated on all AMPs. Then we do a SORT to order Spool 3 by
      row hash. The size of Spool 3 is estimated to be 20 rows. The
      estimated time for this step is 0.46 seconds.
3) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets. Spool 2 and
   TEST.Widgets are joined using a product join, with a join condition
   of ("TEST.Widgets.color = Spool_2.code"). The result goes into
   Spool 4, which is built locally on the AMPs. Then we do a SORT to
   order Spool 4 by row hash. The size of Spool 4 is estimated to be
   500,000 rows. The estimated time for this step is 3 hours and 28
   minutes.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a merge join, with a join condition of
   ("(Spool_4.options = Spool_3.code) AND (Spool_4.size =
   Spool_3.code)"). The result goes into Spool 1, which is built
   locally on the AMPs. The size of Spool 1 is estimated to be 555
   rows. The estimated time for this step is 21.05 seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 3 hours and 28 minutes.
```

Reasonable Unindexed Join With Join Optimization

Optimized Join Plan

With star join optimization, the following join plan is generated.

Operation	Joined Tables	Total Processing Time (seconds)
Spool 2: Merge Join	direct options, direct size	0.44
Spool 3: Product Join	direct color, duped 2	1.24
Spool 1: Merge Join	local widgets, duped 3	7 761.00[a]

a. 2 hours, 9 minutes, 21 seconds.

Completion Time

The total estimated completion time is 2 hours 9 minutes.

The estimated performance improvement factor is 1.6.

EXPLAIN Output for Optimized Join Plan

The following EXPLAIN output is generated.

```
Explanation
-----------------------------------------------------------------------
1) First, we lock TEST.Options for read, we lock TEST.Size for read, we
   lock TEST.Color for read, and we lock TEST.Widgets for read.
2) Next, we do an all-AMPs JOIN step from TEST.Options by way of an
   all-rows scan with no residual conditions, which is joined to
   TEST.Size. TEST.Options and TEST.Size are joined using a merge join,
   with a join condition of ("TEST.Size.description =
   TEST.Options.description"). The result goes into Spool 2, which is
   duplicated on all AMPs. The size of Spool 2 is estimated to be 20
   rows. The estimated time for this step is 0.44 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from TEST.Color by way of an
      all-rows scan with no residual conditions, which is joined to
      Spool 2 (Last Use). TEST.Color and Spool 2 are joined using a
      product join. The result goes into Spool 3, which is duplicated
      on all AMPs. Then we do a SORT to order Spool 3 by row hash. The
      size of Spool 3 is estimated to be 60 rows. The estimated time for
      this step is 1.24 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Widgets by way of an
      all-rows scan with no residual conditions into Spool 4, which is
      built locally on the AMPs. Then we do a SORT to order Spool 4 by
      row hash. The size of Spool 4 is estimated to be 1,000,000 rows.
      The estimated time for this step is 2 hours and 9 minutes.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a merge join, with a join condition of (
   "(Spool_4.color = Spool_3.code) AND ((Spool_4.size = Spool_3.code
   AND (Spool_4.options = Spool_3.code ))"). The result goes into Spool
   1, which is built locally on the AMPs. The size of Spool 1 is
   estimated to be 556 rows. The estimated time for this step is 21.94
   seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 2 hours and 9 minutes.
```

Join Indexes

Introduction

The following sections demonstrate the performance optimization achieved on table selects, deletes, inserts, and updates resulting from the use of join indexes.

For additional information about join indexes, see “Join Indexes” in Chapter 2: “SQL Data Definition, Control, and Manipulation” of *Teradata RDBMS SQL Reference, Volume 1* and “CREATE JOIN INDEX” in Chapter 1: “SQL Data Definition Language Statement Syntax” of *Teradata RDBMS SQL Reference, Volume 4*.

Example 1: Simple Join Query

The following is an example of a simple join query:

```
EXPLAIN
SELECT o_orderdate, o_custkey, 1_partkey, 1_quantity,
 1_extendedprice
FROM Lineitem, Ordertbl
WHERE 1_orderkey = o_orderkey;

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

Example 2: Search on Join Indexed Rows

The following is an example of a search condition on the join indexed rows:

```
EXPLAIN
SELECT o_orderdate, o_custkey, 1_partkey,
 1_quantity,1_extendedprice
FROM Lineitem, Ordertbl
WHERE 1_orderkey = o_orderkey
AND o_orderdate > '1997-11-01';

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

Example 3: Aggregation Against Join-Indexed Rows

The following is an example of an aggregation against the Join-Indexed rows:

```
EXPLAIN
SELECT 1_partkey, AVG(1_quantity), AVG(1_extendedprice)
FROM Lineitem, Ordertbl
WHERE 1_ordkey=o_orderkey
AND o_orderdate > '1997-11-01'
GROUP BY 1_partkey;
```

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

Example 4: Join Indexed Rows Used to Join With Another Base Table

The following is an example of a join indexed rows used to join with another base table:

```
EXPLAIN
SELECT o_orderdate, c_name, c_phone, 1_partkey,1_quantity,
 1_extendedprice
FROM Lineitem, Ordertbl, Customer
WHERE 1_orderkey = o_orderkey
AND o_custkey = c_custkey;
```

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

Example 5: Join Indexed Row Used to Resolve Single Table Query

The following is an example of a join index rows used to resolve single table query:

```
EXPLAIN
SELECT 1_orderkey, 1_partkey, 1_quantity, 1_extendedprice
FROM Lineitem
WHERE 1_partkey = 1001;

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

Example 6: Creating and Using Secondary Index on Top of Join Index

The following is an example creating and using secondary index on top of the join index:

```
CREATE INDEX shipidx(1_shipdate) ON OrderJoinLine;

***Index has been created.
***Total elapsed time was 5 seconds.
EXPLAIN
SELECT o_orderdate, o_custkey, 1_partkey, 1_quantity,
 1_extendedprice
FROM Lineitem, Ordertbl
WHERE 1_ordkey=o_orderkey
AND l_shipdate='1997-09-18';

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

Example 7: Using Join Index Defined With Multiway Join Result

The following is an example defining and using a join index defined with a multiway join result:

```
CREATE JOIN INDEX CustOrderJoinLine
AS SELECT (1_orderkey, o_orderdate, c_nationkey,
 o_totalprice), (1_partkey, 1_quantity, 1_extendedprice,
 1_shipdate)
FROM (Lineitem
LEFT OUTER JOIN Ordertbl ON 1_orderkey=o_orderkey)
INNER JOIN Customer ON o_custkey=c_custkey
PRIMARY INDEX (1_orderkey);

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

```
EXPLAIN
SELECT (1_orderkey, o_orderdate, c_nationkey, o_totalprice),
 (1_partkey, 1_quantity, 1_extendedprice, 1_shipdate)
FROM Lineitem, Ordertbl, Customer
WHERE 1_orderkey = o_custkey
AND o_custkey = c_custkey
AND c_nationkey = 10;

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


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

Maintenance of Join Index for DELETE, INSERT, and UPDATE

Introduction

As with other indexes (for example, secondary indexes), join indexes are automatically maintained by the system when DELETE, INSERT, or UPDATE statements are issued against the underlying base tables of a join index.

Overhead Costs

When considering the use of join indexes, carefully analyze the overhead cost associated with maintaining them during updates and weigh these costs against the benefits to query performance.

Join indexes are maintained by generating additional AMP steps in the execution plan.

The general case method involves first reproducing the affected portion of the join index rows. This is accomplished by re-executing the join query, as defined in the join index, using only those base table rows that are relevant to the update at hand.

The entire join index result is not reproduced for each update statement.

FOR this category of database operation ...	Join indexes are maintained in this way ...
DELETE	The corresponding rows in the join index are located and then deleted with a Merge Delete step.
INSERT	Newly formed join result rows are added to the join index with a Merge step.
UPDATE	The necessary modifications are performed and the corresponding rows in the join index are then replaced by first deleting the old rows (with a Merge Delete) and then inserting the new rows (with a Merge). Join indexes defined with outer joins usually require additional steps to maintain unmatched rows.

As with secondary indexes, updates can cause a physical join index row to split into multiple rows. Each newly formed row has the same fixed field value but contains a different list of repeated field values.

Teradata does not automatically recombine such rows, so the join index must be dropped and recreated to recombine them.

Join Index Definition for Examples

The examples in the following subsections assume the presence of the following join index:

```
CREATE JOIN INDEX OrderJoinLine AS
SELECT (l_orderkey, o_orderdate, o_custkey, o_totalprice),
 (l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM Lineitem
LEFT OUTER JOIN Ordertbl ON l_orderkey=o_orderkey
ORDER BY o_orderdate
PRIMARY INDEX (l_orderkey);
```

General Method of Maintaining Join Index During DELETE

The following is an example of a general case method for maintaining join index during a DELETE.

Note the following items in the EXPLAIN output:

This step ...	Does this ...
5	Reproduces the affected portion of the join index rows.
6.1	Deletes the corresponding rows from the join index.
8	Inserts new non-matching outer join rows into the join index.

```
EXPLAIN
DELETE FROM Ordertbl
WHERE o_custkey = 1001;

 *** Help information returned. 37 rows.
 *** Total elapsed time was 2 seconds.
```

```
Explanation
------------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for write on a
     RowHash to prevent global deadlock for LOUISB.OrderJoinLine.
  2) Next, we lock a distinct LOUISB."pseudo table" for write on a
     RowHash to prevent global deadlock for LOUISB.Ordertbl.
  3) We lock a distinct LOUISB."pseudo table" for read on a RowHash to
     prevent global deadlock for LOUISB.Lineitem.
  4) We lock LOUISB.OrderJoinLine for write, we lock LOUISB.Ordertbl
     for write, and we lock LOUISB.Lineitem for read.
  5) We do an all-AMPs JOIN step from LOUISB.Ordertbl by way of a
     RowHash match scan with a condition of (
     "LOUISB.Ordertbl.o_custkey = 1001"), which is joined to
     LOUISB.Lineitem.  LOUISB.Ordertbl and LOUISB.Lineitem are joined
     using a merge join, with a join condition of (
     "LOUISB.Lineitem.l_orderkey = LOUISB.Ordertbl.o_orderkey").  The
     input tables LOUISB.Ordertbl and LOUISB.Lineitem will not be
     cached in memory, but they are eligible for synchronized scanning.
     The result goes into Spool 1, which is built locally on the AMPs.
     Then we do a SORT to order Spool 1 by row hash.  The result spool
     file will not be cached in memory.  The size of Spool 1 is
     estimated to be 100,000 rows.  The estimated time for this step is
     6 minutes and 5 seconds.
  6) We execute the following steps in parallel.
       1) We do a MERGE DELETE to LOUISB.OrderJoinLine from Spool 1.
       2) We do an all-AMPs DELETE from LOUISB.Ordertbl by way of an
          all-rows scan with a condition of
          ("LOUISB.Ordertbl.o_custkey = 1001").
  7) We do an all-AMPs RETRIEVE step from Spool 1 (Last Use) by way of
     an all-rows scan into Spool 2, which is redistributed by hash code
     to all AMPs.  Then we do a SORT to order Spool 2 by join index.
     The result spool file will not be cached in memory.  The size of
     Spool 2 is estimated to be 100,000 rows.  The estimated time for
     this step is 35 minutes and 59 seconds.
  8) We do a MERGE into LOUISB.OrderJoinLine from Spool 2 (Last Use).
  9) We spoil the parser's dictionary cache for the table.
 10) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

Optimized Method of Maintaining Join Index During DELETE

Direct Access Index Updates

Optimizations have been made for update statements that allow the affected join index rows to be located via direct access. For example, if a DELETE specifies a search condition on the primary or secondary of a join index, the affected join index rows are not reproduced. Instead, the join index may be directly searched for the qualifying rows and modified accordingly.

Preconditions for Optimization

To use this optimized method (that is, the direct update approach), the following conditions must be present:

- A primary or secondary access path to the join index.
- If *join_index_field_2* is defined, no modifications to *join_index_field_1* columns.
- No modifications to the join condition columns appearing in the join index definition.
- No modifications to the primary index columns of the join index.

Example

The following is an example of an optimized method for maintaining join index during a DELETE:

```
EXPLAIN
DELETE FROM LineItem
WHERE l_orderkey = 10;

 *** Help information returned. 11 rows.
 *** Total elapsed time was 2 seconds.

Explanation
-----------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP DELETE from join index table
          LOUISB.OrderJoinLine by way of the primary index
          "LOUISB.OrderJoinLine.l_orderkey = 10" with a residual
          condition of ("LOUISB.OrderJoinLine.l_orderkey = 10").
       2) We do a single-AMP DELETE from LOUISB.LineItem by way of the
          primary index "LOUISB.LineItem.l_orderkey = 10" with no
          residual conditions.
  2) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

Maintaining Join Index During INSERT

The following is an example of maintaining join index during an INSERT.

Note the following items in the EXPLAIN output.:

This step ...	Does this ...
9	Produces the new join result rows for the join index.
11	Deletes any formerly unmatched outer join rows in the join index.
12	Inserts the new join result rows into the join index.

```
   EXPLAIN
   INSERT Ordertbl
   SELECT *
   FROM NewOrders;

    *** Help information returned. 46 rows.
    *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     RowHash to prevent global deadlock for LOUISB.NewOrders.
  2) Next, we lock a distinct LOUISB."pseudo table" for write on a
     RowHash to prevent global deadlock for LOUISB.OrderJoinLine.
  3) We lock a distinct LOUISB."pseudo table" for write on a RowHash to
     prevent global deadlock for LOUISB.Ordertbl.
  4) We lock a distinct LOUISB."pseudo table" for read on a RowHash to
     prevent global deadlock for LOUISB.Lineitem.
  5) We lock LOUISB.NewOrders for read, we lock LOUISB.OrderJoinLine
     for write, we lock LOUISB.Ordertbl for write, and we lock
     LOUISB.Lineitem for read.
  6) We do an all-AMPs RETRIEVE step from LOUISB.NewOrders by way of an
     all-rows scan with no residual conditions into Spool 1, which is
     built locally on the AMPs.  Then we do a SORT to order Spool 1 by
     row hash.  The input table will not be cached in memory, but it is
     eligible for synchronized scanning.  The result spool file will
     not be cached in memory.  The size of Spool 1 is estimated to be 2
     rows.  The estimated time for this step is 0.04 seconds.
  7) We do a MERGE into LOUISB.Ordertbl from Spool 1.
  8) We do an all-AMPs RETRIEVE step from Spool 1 (Last Use) by way of
     an all-rows scan into Spool 3, which is redistributed by hash code
     to all AMPs.  Then we do a SORT to order Spool 3 by row hash.  The
     result spool file will not be cached in memory.  The size of Spool
     3 is estimated to be 2 rows.  The estimated time for this step is
     0.07 seconds.
  9) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of a
     RowHash match scan, which is joined to LOUISB.Lineitem.  Spool 3
     and LOUISB.Lineitem are joined using a merge join, with a join
     condition of ("LOUISB.Lineitem.l_orderkey = Spool_3.o_orderkey").
```

```
    The input table LOUISB.Lineitem will not be cached in memory, but
    it is eligible for synchronized scanning.  The result goes into
    Spool 2, which is built locally on the AMPs.  Then we do a SORT to
    order Spool 2 by row hash.  The result spool file will not be
    cached in memory.  The size of Spool 2 is estimated to be 20 rows.
    The estimated time for this step is 0.37 seconds.
10) We do an all-AMPs RETRIEVE step from Spool 2 by way of an all-rows
    scan into Spool 4, which is built locally on the AMPs.  Then we do
    a SORT to order Spool 4 by join index.
11) We do a MERGE DELETE to LOUISB.OrderJoinLine from Spool 2 (Last
    Use).
12) We do a MERGE into LOUISB.OrderJoinLine from Spool 4 (Last Use).
13) We spoil the parser's dictionary cache for the table.
14) We spoil the parser's dictionary cache for the table.
15) Finally, we send out an END TRANSACTION step to all AMPs involved
    in processing the request.
 -> No rows are returned to the user as the result of statement 1.
```

General Method of Maintaining Join Index During UPDATE

The following is an example of a general method of maintaining join index during an UPDATE.

Note the following items in the EXPLAIN output.:

This step ...	Does this ...
5	reproduces the affected portion of the join index rows.
6.1	deletes the corresponding rows from the join index.
6.2	reproduces the affected portion of the join index rows and makes the necessary modifications to form the new rows.
7.1	inserts the newly modified rows into the join index.

```
EXPLAIN
UPDATE Lineitem
SET l_extendedprice = l_extendedprice * .80
WHERE l_partkey = 50;

 *** Help information returned. 47 rows.
 *** Total elapsed time was 3 seconds.
```

```
Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for write on a
     RowHash to prevent global deadlock for LOUISB.OrderJoinLine.
  2) Next, we lock a distinct LOUISB."pseudo table" for read on a
     RowHash to prevent global deadlock for LOUISB.Ordertbl.
  3) We lock a distinct LOUISB."pseudo table" for write on a RowHash to
     prevent global deadlock for LOUISB.Lineitem.
  4) We lock LOUISB.OrderJoinLine for write, we lock LOUISB.Ordertbl
     for read, and we lock LOUISB.Lineitem for write.
  5) We do an all-AMPs JOIN step from LOUISB.Lineitem by way of a
     RowHash match scan with a condition of (
     "LOUISB.Lineitem.l_partkey = 50"), which is joined to
     LOUISB.Ordertbl.  LOUISB.Lineitem and LOUISB.Ordertbl are
     left outer joined using a merge join, with a join condition of (
     "LOUISB.Lineitem.l_orderkey = LOUISB.Ordertbl.o_orderkey").  The
     input tables LOUISB.Lineitem and LOUISB.Ordertbl will not be
     cached in memory, but they are eligible for synchronized scanning.
     The result goes into Spool 1, which is built locally on the AMPs.
     Then we do a SORT to order Spool 1 by row hash.  The result spool
     file will not be cached in memory.  The size of Spool 1 is
     estimated to be 100,000 rows.  The estimated time for this step is
     6 minutes and 14 seconds.
  6) We execute the following steps in parallel.
       1) We do a MERGE DELETE to LOUISB.OrderJoinLine from Spool 1
          (Last Use).
       2) We do an all-AMPs JOIN step from LOUISB.Lineitem by way of a
          RowHash match scan with a condition of (
          "LOUISB.Lineitem.l_partkey = 50"), which is joined to
          LOUISB.Ordertbl.  LOUISB.Lineitem and LOUISB.Ordertbl are
          left outer joined using a merge join, with a join condition
        of ("LOUISB.Lineitem.l_orderkey = LOUISB.Ordertbl.o_orderkey").
          The input tables LOUISB.Lineitem and LOUISB.Ordertbl will not
          be cached in memory, but they are eligible for synchronized
          scanning.  The result goes into Spool 2, which is built
          locally on the AMPs.  Then we do a SORT to order Spool 2 by
          join index.  The result spool file will not be cached in
          memory.  The size of Spool 2 is estimated to be 100,000 rows.
          The estimated time for this step is 5 minutes and 7 seconds.
  7) We execute the following steps in parallel.
       1) We do a MERGE into LOUISB.OrderJoinLine from Spool 2 (Last
          Use).
       2) We do an all-AMPs UPDATE from LOUISB.Lineitem by way of an
          all-rows scan with a condition of (
          "LOUISB.Lineitem.l_partkey = 50").
  8) We spoil the parser's dictionary cache for the table.
  9) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

Optimized Method of Maintaining Join Index During UPDATE

Direct Access Index Updates

Optimizations have been made for update statements that allow the affected join index rows to be located via direct access. For example, if an UPDATE specifies a search condition on the primary or secondary of a join index, the affected join index rows are not reproduced. Instead, the join index may be directly searched for the qualifying rows and modified accordingly.

Preconditions for Optimization

To use this optimized method (that is, the direct update approach), the following conditions must be present:

- A primary or secondary access path to the join index.
- If *join_index_field_2* is defined, no modifications to *join_index_field_1* columns.
- No modifications to the join condition columns appearing in the join index definition.
- No modifications to the primary index columns of the join index.

Example

The following is an example of an optimized method for maintaining join index during an UPDATE:

```
   EXPLAIN
   UPDATE Lineitem
   SET l_quantity = l_quantity - 5
   WHERE l_orderkey = 10;

    *** Help information returned. 11 rows.
    *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP UPDATE from join index table
          LOUISB.OrderJoinLine by way of the primary index
          "LOUISB.OrderJoinLine.l_orderkey = 10" with a residual
          condition of ("LOUISB.OrderJoinLine.l_orderkey = 10").
       2) We do a single-AMP UPDATE from LOUISB.Lineitem by way of the
          primary index "LOUISB.Lineitem.l_orderkey = 10" with no
          residual conditions.
  2) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

Chapter 4:

Query Capture Facility

This chapter describes the Query Capture Database (QCD) that supports the various Database Query Analysis tools such as the Teradata Query Capture Facility, the Visual EXPLAIN utility, and the Teradata Index and Statistics wizards.

The purpose of QCD is described and the DDL for creating the QCD tables is provided as are various usage suggestions.

The physical implementation of QCD is presented here so you can create and perform your own queries and applications against the information it stores.

Among the topics documented are the following:

- "Compatibility Issues With Prior Teradata Releases" on page 4-2
- "Functional Overview of the Query Capture Facility" on page 4-3
- "Creating the QCD Tables" on page 4-7
- "Querying QCD" on page 4-13
- "QCD Table Definitions" on page 4-17

Compatibility Issues With Prior Teradata Releases

Note the following backward compatibility issues with respect to query capture databases:

- Because the QCD is almost completely redesigned, query capture databases generated prior to release 5.0 are not usable with the Teradata Index and Statistics wizards.

  The QCD table SeqNumber stores the QCD version number.

  If the version of any existing QCD database is lower than QCF03.00.00, then you must migrate the data to a new QCD.

  You can use the Control Center feature of the client Visual EXPLAIN utility to load data from existing query capture databases to new QCDs.
- Applications written to manipulate QCDs created in earlier releases must be modified to work with the new QCD schema.
- Teradata no longer provides a default QCD. Because of this, DUMP EXPLAIN and INSERT EXPLAIN statements for release 5.0 require you to specify a QCD name.

  QCD applications that worked prior to release 5.0 must be examined carefully to ensure that they continue to work and must be converted if they are no longer compatible with the QCD schema.

Functional Overview of the Query Capture Facility

Introduction

The Query Capture Facility, or QCF, provides a method to capture and store the steps from any query plan in a set of predefined relational tables called the Query Capture Database, or QCD.

You can also create your own QCD databases using the procedures described in "Creating the QCD Tables" on page 4-7.

QCD Information Source

The principal source of the captured information in QCD is the white tree produced by the Teradata query optimizer, the same data structure used to produce EXPLAIN reports (note that the current implementation of QCD does not represent *all* the information reported by EXPLAIN). The white tree was chosen because it represents the output of the final stage of optimization by the Teradata query optimizer.

Statistical and other demographic information in the QCD is captured using the following set of SQL statements:

- COLLECT DEMOGRAPHICS

  See the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4* for further information.
- COLLECT STATISTICS (QCD Form)

  See the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4* for further information.
- INSERT EXPLAIN … WITH STATISTICS

  See the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4* for further information.

Applications of QCF and QCD

Teradata supports the following applications of QCF and QCD:

- QCD provides the foundation for the Teradata Index and Statistics Wizard utilities.

  These utilities analyze various SQL query workloads and recommend the following categories of actions be taken:

  - In the case of the Index Wizard, candidate indexes to enhance the performance of those queries or candidate columns in the context of the defined workloads are suggested.
  - In the case of the Statistics Wizard, columns and indexes on which statistics should be collected or recollected, respectively, in the context of the defined workloads.

  The workload definitions, supporting statistical and demographic data, and index recommendations are stored in various QCD tables.
- QCD can store all query plans for customer queries. You can then compare and contrast queries as a function of software release, hardware platform, and hardware configuration.
- QCD provides the foundation for the Visual EXPLAIN tool, which displays EXPLAIN output graphically.

  Visual EXPLAIN also has an option that compares different EXPLAIN reports. This feature can be used to compare visually the white trees of identical queries performed on different hardware configurations or software releases as well as comparing semantically identical but syntactically different DML statements to analyze their relative performance.
- You can generate your own detailed analyses of captured query steps using standard SQL DML statements and third party query management tools by asking such questions as “how many spool files are used by this query,” “did this query plan involve a product join,” or “how many of the steps performed by this query were done in parallel.”

Capacity Planning for QCD

See "Sizing a Query Capture Database" in *Teradata RDBMS Database Design* for information about physical capacity planning for your QCDs.

Query Capture Database

Introduction

The Query Capture Database, or QCD, is the minimum set of tables required to store the Teradata query optimizer white tree information, query analysis workload data, and the statistical and demographic data needed to support query analysis.

QCD is a user database, not a set of system tables. You can define multiple QCD databases for your systems. The DDL for the QCD tables is documented in this chapter on page 4-18 through page 4-69.

Use the Control Center feature of the Visual EXPLAIN tool (see "Procedure Using the Visual EXPLAIN Utility" on page 4-7) or BTEQ using the CreateQCF table (see "Procedure Using BTEQ" on page 4-9) to create your Query Capture databases.

QCD tables are populated using various methods, including the following:

- COLLECT DEMOGRAPHICS
- COLLECT STATISTICS (QCD Form)
- INSERT EXPLAIN … WITH STATISTICS
- Invoking workload macros from the client-based Teradata Index Wizard utility.

Caution Because the QCD is a set of user tables, you can change both their individual structure and the structure of the QCD; however, you should *never* change *any* of the QCD structures in any way.

The reason for this constraint is that the SQL statements DUMP EXPLAIN, INSERT EXPLAIN, INITIATE INDEX ANALYSIS, RESTART INDEX ANALYSIS, COLLECT DEMOGRAPHICS, and COLLECT STATISTICS (QCD Form) all assume the default physical model when they capture the specified query, statistical, and demographic information for insertion into the QCD tables. The results of changing the structure of the QCD tables are unpredictable.

Deleting Query Plans from QCD

Use the Control Center option of Visual EXPLAIN to delete query plans from the QCD.

Step	Action
1	Start Visual EXPLAIN.
2	Pull down the Tools menu and select Control Center.
3	Select the Manage QCD tab.
4	Select the Clean up QCD button.
5	Select the database from which query plans are to be removed from the Database scroll bar.
6	Select the Delete Database Objects radio button.
7	Select the OK button.

QCD Physical Model

Note the following facts about the physical implementation of QCD.

- All text columns that might contain names of any type are explicitly defined as UNICODE to ensure proper handling of any Teradata-supported character set.
- The table SeqNumber was created to feed the values of the artificial sequential number columns defined for several attributes in QCD.

Creating the QCD Tables

Introduction

Before you can capture query plan information from the Optimizer white tree and statistical and data demographic information from their respective synopsis data structures, you must either create and secure the QCD tables or create your own user-defined query capture database. This topic explains two ways to set up your QCDs.

For instructions on dropping the tables from a QCD, see "Dropping the QCD Tables" on page 4-11.

Procedure Using the Visual EXPLAIN Utility

Perform the following procedure to create the tables and other database objects for the Query Capture Database using Visual EXPLAIN. If you do not have the Visual EXPLAIN tool, then see "Procedure Using BTEQ" on page 4-9.

Step	Action
1	Start Visual EXPLAIN.
2	Pull down the Tools menu and select Control Center.
3	Select the Manage QCD tab.
4	Select the Setup QCD button.
5	Select the Create all QCF database objects (tables and macros) radio button.
6	Type the name of the database owner in the Owner selection box.
7	Specify the Perm Space parameter in the Perm Space selection box.
8	Specify the Spool Space parameter in the Spool Space selection box.
9	Check the Fallback check box depending on your requirements.
10	Select the Create button.
11	Secure the QCD tables by granting the appropriate access rights to the users who will be analyzing its data. You can implement additional secured access by creating views on QCD.

Step	Action
12	Populate the QCD tables using the appropriate tools: • INSERT EXPLAIN statements to capture query plans. See the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4* for the syntax of this statement. • Any of the following statements, as appropriate, to capture statistical and demographic data: – COLLECT DEMOGRAPHICS – COLLECT STATISTICS (QCD Form) – INSERT EXPLAIN … WITH STATISTICS • The appropriate workload macros to create workloads for index analysis.

Procedure Using BTEQ

Perform the following procedure to create the tables for the Query Capture Database using BTEQ. This procedure does not create the macros that the Visual EXPLAIN utility needs to work with the QCD. You must create these macros from the Visual EXPLAIN Control Center before the utility can visualize or load QCD query plans.

The procedure assumes that you have already created *QCF_database_name*.

Step	Action
1	Start BTEQ.
2	Change your current database to the database in which the QCF tables are to be created as follows. `DATABASE QCF_database_name;` where *QCF_database_name* is the name of the database you created for the QCD tables.
3	Perform the following steps in order. 1 `.SET WIDTH 254` 2 `.EXPORT FILE = file_name` 3 `SELECT TabDefinition FROM systemfe.CreateQCF ORDER BY SeqNumber;` 4 `EXPORT RESET;` 5 `.RUN FILE = file_name` where *file_name* is the name of the file you create to contain the output of the SELECT statement in Step 2.
4	Secure QCD by granting the appropriate access rights to the users who will be analyzing its data. You can implement additional secured access by creating views on QCD.

Step	Action
5	Populate the QCD tables using the appropriate tools: • INSERT EXPLAIN statements to capture query plans. See the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4* for the syntax of this statement. • Any of the following statements, as appropriate, to capture statistical and demographic data: – COLLECT DEMOGRAPHICS – COLLECT STATISTICS (QCD Form) – INSERT EXPLAIN … WITH STATISTICS • The appropriate workload macros to create workloads for index analysis. Note that client tools like Visual EXPLAIN and the Teradata Index Wizard access QCD tables using the views and macros created using the Control Center feature of Visual EXPLAIN.

Dropping the QCD Tables

Introduction

This topic describes how to drop all the tables and other database objects from a QCD database.

Procedure Using Visual EXPLAIN

Perform the following procedure to drop the tables and other database objects for the Query Capture Database using Visual EXPLAIN. If you do not have the Visual EXPLAIN tool, then see "Procedure Using BTEQ" on page 4-12.

Step	Action
1	Start Visual Explain.
2	Pull down the Tools menu and select Control Center.
3	Select the Manage QCD tab.
4	Select the Clean up QCD button.
5	Select the database to be cleaned up from the Database scroll bar menu.
6	Select the Delete Database Objects radio button.
7	Select the OK button.

Procedure Using BTEQ

Perform the following procedure to drop the tables for the Query Capture Database using BTEQ. The procedure assumes that you have already created *QCF_database_name*. This procedure does *not* drop the views and macros on the specified QCD.

Step	Action
1	Start BTEQ.
2	Change your current database to the database from which the QCF tables are to be dropped as follows. `DATABASE QCF_database_name;` where *QCF_database_name* is the name of the database you created for the QCD tables.

3	Perform the following steps in order.
	1 `.EXPORT FILE = file_name`
	2 `SELECT DelTable`
	`FROM systemfe.CleanupQCF`
	`ORDER BY SeqNumber;`
	3 `EXPORT RESET;`
	4 `.RUN FILE = file_name`
	where *file_name* is the name of the file you create to contain the output of the SELECT statement in Step 2.

Querying QCD

You can analyze the information in QCD using several different approaches.

- To create and compare graphical EXPLAIN reports, use the Visual EXPLAIN tool.
- To perform ad hoc queries of QCD, use interactive SQL DML statements.
- To perform standard queries for index analysis, use the index recommendations macros (see “Index Recommendation Query Macros” on page 1-30.)
- To perform standardized analyses of QCD, create macros, stored procedures, or embedded SQL applications to perform your standard queries and report the results.

  See “QCD Query Macros and Views” on page 4-14 for a set of macros you can use for this purpose.
- To run index validation diagnostics for the Teradata Index Wizard, use the DIAGNOSTIC "Validate Index" statement.

  See "DIAGNOSTIC ’Validate Index’" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.

QCD Query Macros and Views

Introduction

Teradata provides a set of views on the QCD tables to restrict their access. Various levels of access to QCD tables are granted to different user categories.

Teradata also provides a set of macros for maintaining various aspects of the Query Capture Database. The Control Center feature of the Visual EXPLAIN client utility provides the interface to create the views and macros and to grant access rights to QCD users.

QCD Macro and View Versions

Two versions of all the views and macros are created in a QCD:

Version	Definition
X	Access restricted to information inserted by the current user in the QCD.
NonX	Access permitted to information inserted by any user of the QCD.

These macros and views are usually performed via the Visual EXPLAIN and Teradata Index Wizard client utilities.

User Categories

The following categories of users are defined on a QCD to enhance its security:

User Category	Description
Normal	Loads, views, and deletes own plans or workloads only.
Power	Loads and views plans or workloads inserted by any user. Deletes own plans or workloads only.
Administrator	Loads, view, and deletes any plan created by any user. Drops and deletes QCD tables. QCD creator has Administrator privileges granted by default.

Specific User Category Privileges

The following table indicates the specific privileges granted to each user category:

User Category	Database Object Type	Privileges Granted
Normal	QCD tables	• INSERT • UPDATE • SELECT on the following tables: – IndexRecommendations – SeqNumber – TableStatistics • DELETE on the following table: – AnalysisLog
	X views	SELECT
	X macros	EXEC
	Non-X views	none
	Non-X macros	none
Power	QCD tables	• INSERT • UPDATE • SELECT on the following tables: – IndexRecommendations – SeqNumber – TableStatistics • DELETE on the following table: – AnalysisLog
	X views	SELECT
	X macros	EXEC
	Non-X views	SELECT
	Non-X macros	EXEC (excluding DELETE plan and workload macros)

User Category	Database Object Type	Privileges Granted
Administrator	QCD tables	• DELETE • DROP • INSERT • SELECT • UPDATE
	X views	• DELETE • DROP • INSERT • SELECT • UPDATE
	X macros	• DROP • EXECUTE
	Non-X views	• DELETE • DROP • INSERT • SELECT • UPDATE
	Non-X macros	• DROP • EXECUTE

QCD Table Definitions

The definitions for the QCD tables are provided in the next several pages.

Tables defined and the pages on which their definitions begin are listed below.

- "AnalysisLog" on page 4-18
- "DataDemographics" on page 4-20
- "Field" on page 4-22
- "Index_Field" on page 4-25
- "IndexColumns" on page 4-26
- "IndexRecommendations" on page 4-27
- "IndexTable" on page 4-31
- "Predicate" on page 4-35
- "Predicate_Field" on page 4-37
- "QryRelX" on page 4-38
- "Query" on page 4-40
- "QuerySteps" on page 4-44
- "Relation" on page 4-51
- "SeqNumber" on page 4-57
- "TableStatistics" on page 4-59
- "User_Database" on page 4-62
- "UserRemarks" on page 4-63
- "ViewTable" on page 4-65
- "Workload" on page 4-67
- "WorkloadQueries" on page 4-68
- "WorkloadStatus" on page 4-69

AnalysisLog

Function

Records the restart log information for index analysis by the Teradata Index Wizard utility.

The table maintains the checkpoint information that is recorded when you enable the CHECKPOINT option for the INITIATE INDEX ANALYSIS statement.

Table Definition

```
CREATE TABLE AnalysisLog (
 WorkLoadID       INTEGER NOT NULL,
 RecommendationID INTEGER NOT NULL,
 IndexNameTag     VARCHAR(30) CHARACTER SET UNICODE
                  NOT CASESPECIFIC NOT NULL,
 SeqNumber        SMALLINT NOT NULL,
 Cflag            CHARACTER(1) CHARACTER SET LATIN
                  NOT CASESPECIFIC NOT NULL,
 UserName         VARCHAR(30) CHARACTER SET UNICODE
                  NOT CASESPECIFIC NOT NULL,
 AnalysisStatus   VARBYTE(32000) NOT NULL,
 StartTime        TIMESTAMP(6) NOT NULL,
 UpdateTime       TIMESTAMP(6) NOT NULL
)
PRIMARY INDEX (WorkloadID, IndexNameTag);
```

Attribute Definitions

The following table defines the AnalysisLog table attributes.

Attribute	Definition
WorkloadID	• Uniquely identifies the workload. • Partial NUPI for the table.
RecommendationID	• Uniquely identifies the index recommendation generated for the specific index analysis.
IndexNameTag	• User-specified name for the index analysis. • Partial NUPI for the table.
SeqNumber	A sequence number to identify individual rows that belong to a multirow AnalysisStatus. The sequence begins at 1.

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>Cflag</td><td>Used with multirow AnalysisStatus.
<table><tr><th>IF the row is …</th><th>THEN Cflag is set to …</th></tr><tr><td>the last in the sequence</td><td>F</td></tr><tr><td>not the last in the sequence</td><td>T</td></tr></table></td></tr>
<tr><td>UserName</td><td>Name of the user performing the index analysis.</td></tr>
<tr><td>AnalysisStatus</td><td>Identifies the status of the index analysis.
This column contains encoded information that the index wizard infrastructure on the Teradata RDBMS uses. During a checkpoint operation, this information is updated</td></tr>
<tr><td>StartTime</td><td>Timestamp for the beginning of the index analysis.</td></tr>
<tr><td>UpdateTime</td><td>Timestamp for the moment the AnalysisStatus column is most recently updated.</td></tr>
</table>

DataDemographics

Function

Contains table demographic information for use by the Teradata Index Wizard and Visual EXPLAIN client utilities.

Table Definition

The following CREATE TABLE statement defines the DataDemographics table.

```
CREATE TABLE DataDemographics (
 MachineName    VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 TableName      VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 DatabaseName   VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 DBSize         INTEGER NOT NULL,
 CollectedTime  TIMESTAMP(6) NOT NULL,
 AMPNumber      INTEGER NOT NULL,
 ClusterNumber  INTEGER NOT NULL,
 SubTableID     SMALLINT NOT NULL,
 SubTableType   VARCHAR(120),
 RowCount       DECIMAL(18,0) NOT NULL,
 AvgRowSize     INTEGER NOT NULL,
 QueryID        INTEGER,
 IndexName      VARCHAR(2048) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 DemographicsID INTEGER)
PRIMARY INDEX (MachineName, DatabaseName, TableName);
```

Attribute Definitions

The following table defines the DataDemographics table attributes.

Attribute	Definition
MachineName	• The name of the system to which TableName belongs. • Partial NUPI for the table.
TableName	• The name of the table on which demographic data has been collected. • Partial NUPI for the table.
DatabaseName	• The name of the containing database for TableName. • Partial NUPI for the table.
DBSize	The size in KB of data blocks in TableName.

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>CollectedTime</td><td>The timestamp value when the data demographics were collected.</td></tr>
<tr><td>AMPNumber</td><td>The AMP Vproc number to which the row information pertains.</td></tr>
<tr><td>ClusterNumber</td><td>The number of the cluster to which AMPNumber belongs.</td></tr>
<tr><td>SubTableID</td><td>The unique identifier for the subtable in which the data the row describes is stored.</td></tr>
<tr><td>SubTableType</td><td>The subtable data in text format.</td></tr>
<tr><td>RowCount</td><td>The cardinality of the subtable.</td></tr>
<tr><td>AvgRowSize</td><td>The average size of a row in the subtable.</td></tr>
<tr><td>QueryID</td><td>The value for QueryID depends on how the demographics are captured.
<table>
<tr><th>IF demographics are captured by this statement …</th><th>THEN QueryID is …</th></tr>
<tr><td>COLLECT DEMOGRAPHICS</td><td>null</td></tr>
<tr><td>INSERT EXPLAIN</td><td>the unique ID of the query.</td></tr>
</table></td></tr>
<tr><td>IndexName</td><td>A comma-separated list of the names of the primary and secondary indexes for the table.
If an index has no name, then it is represented in this field by a comma-separated list of the names of the columns that compose it.</td></tr>
<tr><td>DemographicsID</td><td>Set to 1 if the demographics are captured by a COLLECT DEMOGRAPHICS or INSERT EXPLAIN AND DEMOGRAPHICS statement.
1 indicates that the capture is on the system on which the row is inserted.
DemographicsID has different values if the demographics are imported rather than captured directly.</td></tr>
</table>

Spatial Distribution of a Table Across AMPs

DataDemographics does not contain information about the spatial distribution of tables across the AMPs.

You can view those details by querying the system view DBC.TableSizeX.

Field

Function

Captures all the columns (fields) used or referenced in a captured query plan.

Table Definition

The following CREATE TABLE statement defines the Field table.

```
CREATE TABLE Field (
   FieldID              INTEGER NOT NULL,
   RelationKey          INTEGER NOT NULL,
   Name                 VARCHAR(30) CHARACTER SET UNICODE
                        NOT CASESPECIFIC,
   FldAlias             VARCHAR(30) CHARACTER SET UNICODE
                        NOT CASESPECIFIC,
   QueryID              INTEGER NOT NULL,
   ValueAccessFrequency INTEGER,
   JoinAccessFrequency  INTEGER,
   RangeAccessFrequency INTEGER,
   ChangeRate           INTEGER,
   DataLength           INTEGER,
   StatsKind            CHAR(1),
   NumNulls             FLOAT,
   NumIntervals         INTEGER,
   MinValue             VARCHAR(512),
   ModeValue            VARCHAR(512),
   ModeFreq             DOUBLE PRECISION,
   TotalValues          DOUBLE PRECISION,
   TotalRows            INTEGER)
PRIMARY INDEX(RelationKey)
UNIQUE INDEX USK_FieldID_RelationKey ( FieldID, RelationKey );
```

Attribute Definitions

The following table defines the Field table attributes.

Attribute	Definition
FieldID	• Unique field identifier within a specific table. • Partial USI for the table.
RelationKey	• Unique identifier for the relation in which the field is defined within a certain database. • NUPI for the table. • Partial USI for the table.

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>Name</td><td>The name of the captured column.</td></tr>
<tr><td>FldAlias</td><td>• Alias name for the field.
• Null if none exists.</td></tr>
<tr><td>QueryID</td><td>The unique ID for the query.</td></tr>
<tr><td>ValueAccessFrequency</td><td>The number of times the column is used with an equality condition.</td></tr>
<tr><td>JoinAccessFrequency</td><td>The number of times the column is used in a join condition.</td></tr>
<tr><td>RangeAccessFrequency</td><td>The number of times the column is used in a range condition.</td></tr>
<tr><td>ChangeRate</td><td>The change rating value for the column.
<table><tr><th>IF the column is …</th><th>THEN ChangeRate is set to this value …</th></tr><tr><td>modified</td><td>1</td></tr><tr><td>not modified</td><td>0</td></tr></table></td></tr>
<tr><td>DataLength</td><td>The maximum length of the column.</td></tr>
<tr><td>StatsKind</td><td>Defines whether the statistics are collected from the dictionary or from a QCD.
The data for this column is retrieved from interval 0 of the available statistics.
The value is set to null if no statistics are available for the column or index.
The value is set to S during index validation by the Teradata Index Wizard. Statistics for the column being analyzed are retrieved from the QCD.
<table><tr><th>Code</th><th>Description</th></tr><tr><td>P</td><td>Dictionary statistics.</td></tr><tr><td>S</td><td>QCD statistics.</td></tr><tr><td>null</td><td>No statistics.</td></tr></table></td></tr>
<tr><td>NumNulls</td><td>Number of nulls for the column or index.
The data for this column is retrieved from interval 0 of the available statistics. The value is set to null if no statistics are available.</td></tr>
</table>

Attribute	Definition
NumIntervals	Number of intervals for the column or index. The data for this column is retrieved from interval 0 of the available statistics. The value is set to null if no statistics are available.
MinValue	Minimum value for the interval. The data for this column is retrieved from interval 0 of the available statistics. The value is set to null if no statistics are available.
ModeValue	Modal value for the interval. The data for this column is retrieved from interval 0 of the available statistics. The value is set to null if no statistics are available.
ModeFreq	Number of occurrences of the modal value in the interval. The data for this column is retrieved from interval 0 of the available statistics. The value is set to null if no statistics are available.
TotalValues	Number of unique values for the column or index in the table. The data for this column is retrieved from interval 0 of the available statistics. The value is set to null if no statistics are available.
TotalRows	Cardinality of the table. The data for this column is retrieved from interval 0 of the available statistics. The value is set to null if no statistics are available.

Index_Field

Function

Captures all the index fields used in the query plan.

Table Definition

The following CREATE TABLE statement defines the Index_Field table.

```
CREATE TABLE Index_Field(
 RelationKey  INTEGER NOT NULL,
 IndexNum     INTEGER NOT NULL,
 FieldID      INTEGER NOT NULL)
PRIMARY INDEX PK_Relationkey_IdxNum(RelationKey, IndexNum)
UNIQUE INDEX USK_RelationKey_IdxNum_FieldID (RelationKey,
IndexNum, FieldID);
```

Attribute Definitions

The following table defines the Index_Field table attributes.

Attribute	Definition
RelationKey	• Unique identifier for the relation in which the captured index fields are defined. • Partial NUPI for the table. • Partial USI for the table.
IndexNum	• Unique identifier for the captured index. • Partial NUPI for the table. • Partial USI for the table.
FieldID	• Unique field identifier within a specific table. • Partial USI for the table.

IndexColumns

Function

Captures the columns that form the index identified by the IndexID during index analysis by the Teradata Index Wizard.

Table Definition

The following CREATE TABLE statement defines the IndexColumns table.

```
CREATE TABLE IndexColumns(
 WorkLoadID       INTEGER NOT NULL,
 RecommendationID INTEGER NOT NULL,
 TableID          BYTE(6) NOT NULL,
 IndexID          INTEGER NOT NULL,
 ColumnName       VARCHAR(30) CHARACTER SET UNICODE
                  NOT CASESPECIFIC NOT NULL)
PRIMARY INDEX ( RecommendationID, TableID, IndexID );
```

Attribute Definitions

The following table defines the IndexColumns table attributes.

Attribute	Definition
WorkloadID	Uniquely identifies the workload.
RecommendationID	• Uniquely identifies the recommendation ID in the IndexRecommendations table. • Partial NUPI for the table.
TableID	• Uniquely identifies the table ID in the IndexRecommendations table. • Partial NUPI for the table.
IndexID	• Uniquely identifies the indexID in the IndexRecommendations table. • Partial NUPI for the table.
ColumnName	The name of the column in the index. The number of rows in IndexColumns for a given index is equal to the number of columns in the index, so any composite index has multiple rows associated with it.

IndexRecommendations

Function

Contains information about the index recommendations made by the Teradata Index Wizard utility.

You can query these tables to retrieve the indexes the Index Wizard recommends. IndexRecommendations also records the options specified during index analysis for later retrieval.

Table Definition

The following CREATE TABLE statement defines the IndexRecommendations table.

```
CREATE TABLE IndexRecommendations(
 WorkLoadID           INTEGER NOT NULL,
 UserName             VARCHAR(30) CHARACTER SET UNICODE
                      NOT NULL,
 TimeOfAnalysis       TIMESTAMP(6) NOT NULL,
 RecommendationID     INTEGER NOT NULL,
 QueryID              INTEGER NOT NULL,
 IndexID              INTEGER,
 IndexNameTag         VARCHAR(30) CHARACTER SET UNICODE
                      NOT CASESPECIFIC NOT NULL,
 TableName            VARCHAR(30) CHARACTER SET UNICODE
                      NOT CASESPECIFIC NOT NULL,
 DatabaseName         VARCHAR(30) CHARACTER SET UNICODE
                      NOT CASESPECIFIC NOT NULL,
 TableID              BYTE(6) NOT NULL,
 IndexType            INTEGER,
 IndexTypeText        VARCHAR(30),
 StatisticsInfo       VARBYTE(16383),
 OriginalCost         FLOAT,
 NewCost              FLOAT,
 SpaceEstimate        FLOAT,
 TimeEstimate         FLOAT,
 DropFlag             CHAR(1),
 IndexDDL             VARCHAR(10000) CHARACTER SET UNICODE
                      NOT CASESPECIFIC,
 StatsDDL             VARCHAR(10000) CHARACTER SET UNICODE
                      NOT CASESPECIFIC,
 Remarks              VARCHAR(1024) CHARACTER SET UNICODE
                      NOT CASESPECIFIC,
 AnalysisData         VARCHAR(2048),
 IndexesPerTable      BYTEINT,
 SearchSpaceSize      SMALLINT,
 ChangeRateThreshold BYTEINT,
 ColumnPerIndex       SMALLINT)
PRIMARY INDEX (WorkloadID);
```

Attribute Definitions

The following table defines the IndexRecommendations table attributes.

Attribute	Definition
WorkloadID	• Uniquely identifies the workload. • NUPI for the table.
UserName	Name of the user performing the index analysis.
TimeOfAnalysis	The timestamp when the index recommendations were analyzed. You can compare this field with the modified timestamp for TableName to verify the correctness of the recommendation before applying it to the system.
RecommendationID	Uniquely identifies a set of index recommendations.
QueryID	Uniquely identifies the QueryID of the workload for which the current entry is an index recommendation.
IndexID	Uniquely identifies an index recommended by the Teradata Index Wizard for a table.
IndexNameTag	Name of the index recommendation as specified in the INITIATE INDEX ANALYSIS statement (see "INITIATE INDEX ANALYSIS" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*).
TableName	Name of the table for which the row defines an index recommendation.
DatabaseName	Name of the database containing TableName.
TableID	The unique internal identifier for TableName.
IndexType	A number that identifies the type of index recommended. Each unique index type is associated with its own IndexTypeText.

IndexType Code	IndexTypeText
1	USI
2	VOSI
3	NUSI
4	HOSI

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>IndexTypeText</td><td>The textual representation of IndexType.

The valid IndexTypeText strings and their meanings are as follows:

<table>
<tr><th>IndexTypeText</th><th>Description</th></tr>
<tr><td>USI</td><td>Unique Secondary Index</td></tr>
<tr><td>VOSI</td><td>Value-Ordered Secondary Index</td></tr>
<tr><td>NUSI</td><td>Nonunique Secondary Index</td></tr>
<tr><td>HOSI</td><td>Hash-Ordered Secondary Index</td></tr>
</table>
</td></tr>
<tr><td>StatisticsInfo</td><td>The statistics, if any, used to make the index recommendations.</td></tr>
<tr><td>OriginalCost</td><td>The estimated cost of the query without implementing the recommended indexes.</td></tr>
<tr><td>NewCost</td><td>The estimated cost of the query after implementing the recommended indexes.</td></tr>
<tr><td>SpaceEstimate</td><td>An estimate of the space in bytes the recommended index occupies when created.</td></tr>
<tr><td>TimeEstimate</td><td>TimeEstimate is the estimated time it takes to act on the recommendation</td></tr>
<tr><td>DropFlag</td><td>Identifies whether the specified index is to be added or dropped.

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>N</td><td>The index is not recommended for dropping.</td></tr>
<tr><td>Y</td><td>The index is recommended for dropping.</td></tr>
</table>
</td></tr>
<tr><td>IndexDDL</td><td>The DDL text of the CREATE INDEX or DROP INDEX statement for the index recommended by the Teradata Index Wizard.</td></tr>
<tr><td>StatsDDL</td><td>The DDL text of the COLLECT STATISTICS statements used for the analysis of the index.</td></tr>
<tr><td>Remarks</td><td>Provides details on the analysis involved in making the index recommendation.</td></tr>
<tr><td>AnalysisData</td><td>Reserved for future use.</td></tr>
<tr><td>IndexesPerTable</td><td>The limit on the number of indexes on a given table as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis.</td></tr>
</table>

Attribute	Definition
SearchSpaceSize	The maximum number of candidate indexes that are searched on a given table as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis.
ChangeRateThreshold	The threshold value of the column volatility as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis. Any column with a change rating less than ChangeRateThreshold is available for selection as a candidate index during index analysis.
ColumnsPerIndex	The maximum number of columns permissible in the index as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis.

IndexTable

Function

Describes all indexes on the tables specified by the query.

Table Definition

The following CREATE TABLE statement defines IndexTable.

```
CREATE TABLE IndexTable (
 IndexNum        INTEGER NOT NULL,
 RelationKey     INTEGER NOT NULL,
 OrderBy         CHAR(1) NOT NULL,
 AccessInfo      CHAR(1) NOT NULL,
 Field1Only      CHAR(1) NOT NULL,
 RangeConstraint CHAR(1) NOT NULL,
 IndexFlag       CHAR(1),
 IndexName       VARCHAR(30) CHARACTER SET UNICODE
                 NOT CASESPECIFIC,
 IndexType       CHAR(1),
 UniqueFlag      CHAR(1),
 IndexKind       CHAR(1),
 NumNulls        FLOAT,
 NumIntervals    INTEGER,
 MinValue        VARCHAR(512),
 ModeValue       VARCHAR(512),
 ModeFreq        DOUBLE PRECISION,
 TotalValues     DOUBLE PRECISION,
 TotalRows       DOUBLE PRECISION)
PRIMARY INDEX (RelationKey)
UNIQUE INDEX USK_IdxNum_RelationKey ( IndexNum, RelationKey );
```

Attribute Definitions

The following table defines the IndexTable table attributes.

Attribute	Description
IndexNum	• Unique identifier for the captured index. • Partial USI for the table.
RelationKey	• Unique identifier for the relation in which the captured index is defined. • NUPI for the table. • Partial USI for the table.

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>OrderBy</td><td>Defines whether the index has an associated ORDER BY clause.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Index has no ORDER BY clause.</td></tr><tr><td>T</td><td>Index has an ORDER BY clause.</td></tr></table></td></tr>
<tr><td>AccessInfo</td><td>Specifies if the index is a covering index, bit map, or neither.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>B</td><td>Bit map</td></tr><tr><td>C</td><td>Covering index</td></tr><tr><td>N</td><td>Neither bit map nor covering index</td></tr><tr><td>P</td><td>Primary index access</td></tr></table></td></tr>
<tr><td>Field1Only</td><td>Defines whether the index is a join index and Field1 is the only part needed.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Not a join index requiring Field1 only.</td></tr><tr><td>T</td><td>Join index requiring Field1 only.</td></tr></table></td></tr>
<tr><td>RangeConstraint</td><td>Flag for value-ordered indexes that have a range constraint used by the query plan.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>N</td><td>Nonresidual condition</td></tr><tr><td>R</td><td>Residual condition</td></tr></table></td></tr>
<tr><td>IndexFlag</td><td>Flag indicating whether the index was used in the query plan.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>False. The index was not used in the query plan.</td></tr><tr><td>T</td><td>True. The index was used in the query plan.

IndexFlag is also set to T if a subset of the partitions of a partitioned primary index is accessed because of partition elimination.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>IndexName</td><td>• The name of the index if it has one.
• Null if this is not a named index.</td></tr>
<tr><td>IndexType</td><td>Flag indicating the type of the index.
<table><tr><th>Code</th><th>Description</th></tr><tr><td>H</td><td>Hash index.</td></tr><tr><td>J</td><td>Join index.</td></tr><tr><td>O</td><td>Covering index.</td></tr><tr><td>P</td><td>Nonpartitioned primary index.</td></tr><tr><td>Q</td><td>Partitioned primary index.</td></tr><tr><td>S</td><td>Secondary index.</td></tr></table></td></tr>
<tr><td>UniqueFlag</td><td>Flag indicating whether the index is unique or nonunique.
<table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Nonunique index.</td></tr><tr><td>T</td><td>Unique index.</td></tr></table></td></tr>
<tr><td>IndexKind</td><td>Flag indicating whether the index is permanent or simulated.
Indexes are simulated using the index validation function of the Teradata Index Wizard (see “Index Validation” on page 5-16).
<table><tr><th>Code</th><th>Description</th></tr><tr><td>P</td><td>Permanent index.</td></tr><tr><td>S</td><td>Simulated index.</td></tr></table></td></tr>
<tr><td>NumNulls</td><td>The number of nulls in the index.</td></tr>
<tr><td>NumIntervals</td><td>The number of intervals in the index statistics.</td></tr>
<tr><td>MinValue</td><td>The minimum value of the index.
This is obtained from statistical histogram interval 0 for the index.</td></tr>
<tr><td>ModeValue</td><td>The value of the index that occurs the most in the table.
This is obtained from statistical histogram interval 0 for the index.</td></tr>
<tr><td>ModeFreq</td><td>The number of times the modal value occurs in the index.</td></tr>
</table>

Attribute	Description
TotalValues	The total number of values in the index other than the modal value.
TotalRows	The cardinality of the table.

Predicate

Function

Describes any residual conditions applied for specific AMP steps in a query.

Table Definition

The following CREATE TABLE statement defines the Predicate table.

```
CREATE TABLE Predicate (
 PredicateID   INTEGER NOT NULL,
 StepID        INTEGER NOT NULL,
 PredicateKind CHAR(1) NOT NULL,
 PredicateText VARCHAR(2000) CHARACTER SET UNICODE
               NOT CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX PK_PredID ( PredicateID );
```

Attribute Definitions

The following table defines the Predicate table attributes.

Attribute	Description
PredicateID	• Unique identifier for the predicate. • UPI for the table.
StepID	Unique identifier for the AMP step.
PredicateKind	Describes the kind of predicate condition associated with this step.

Code	Description
A	Additional join condition.
G	Range constraint. Used for value-ordered relations.
J	Join condition.
L	Condition on left relation in a join.
Q	Partition elimination occurs for a source condition. This is a residual condition on the left or right table in a join or on a single-table retrieval. Partition elimination occurs prior to accessing the rows, so the condition applies only to rows retrieved from partitions that were not eliminated.

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>PredicateKind
(continued)</td><td>
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>R</td><td>Condition on right relation in a join.</td></tr>
<tr><td>S</td><td>Source condition.

This is a residual condition on the left or right table in a join or on a single-table retrieval. No partition elimination occurs prior to accessing the rows.</td></tr>
</table>
</td></tr>
<tr><td>PredicateText</td><td>Full text of the predicate as it appears in the EXPLAIN report.</td></tr>
</table>

Predicate_Field

Function

Associates the list of fields specified in a captured predicate with the parent relation and predicate.

Table Definition

The following CREATE TABLE statement defines the Predicate_Field table.

```
CREATE TABLE predicate_field (
 PredicateID INTEGER NOT NULL,
 RelationKey INTEGER NOT NULL,
 FieldID     INTEGER NOT NULL)
PRIMARY INDEX ( RelationKey );
```

Attribute Definitions

The following table defines the Predicate_Field table attributes.

Attribute	Definition
PredicateID	Uniquely identifies the predicate.
RelationKey	• Unique identifier for the referenced table. • NUPI for the Predicate_Field table.
FieldID	Unique identifier for the field.

QryRelX

Function

Stores overflow text for the QueryText attribute of the Query table or the TableDDL attribute of the Relation table. There is at least one row in QryRelX for each row in Query or Relation with an Overflow flag set to T.

Table Definition

The following CREATE TABLE statement defines the QryRelX table.

```
CREATE TABLE QryRelX(
 RowType   CHARACTER(1),
 KeyValue  INTEGER NOT NULL,
 SeqNumber SMALLINT NOT NULL,
 Text      VARCHAR(30000) CHARACTER SET UNICODE
           NOT CASESPECIFIC NOT NULL)
PRIMARY INDEX(RowType, KeyValue);
```

Attribute Definitions

The following table defines the QryRelX table attributes.

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>RowType</td><td>Defines whether the row handles text overflow from the Query table or from the Relation table.
<table><tr><th>Code</th><th>Definition</th></tr><tr><td>Q</td><td>Text is overflow from the Query table.</td></tr><tr><td>R</td><td>Text is overflow from the Relation table.</td></tr></table></td></tr>
<tr><td>KeyValue</td><td>Defines the implicit primary key for QryRelX rows.
<table><tr><th>IF RowType is this value ...</th><th>THEN the value is the ...</th></tr><tr><td>Q</td><td>QueryID</td></tr><tr><td>R</td><td>RelationKey</td></tr></table></td></tr>
</table>

Attribute	Definition
SeqNumber	Specifies whether the current row is the last row in the Overflow. If the overflow QueryText or TableDDL text does not fit into a single row, it is divided into multiple rows, each with its unique SeqNumber value. SeqNumber begins at 1 and is incremented by 1 for each new text row required.
Text	The overflow QueryText or TableDDL text. The upper bound for this text is 30 000 characters per row.

Query

Function

Describes information about captured queries.

Table Definition

The following CREATE TABLE statement defines the Query table.

```
CREATE TABLE Query(
   QueryID         INTEGER NOT NULL,
   UDB_Key         INTEGER NOT NULL,
   MachName        VARCHAR(30) CHARACTER SET UNICODE
                   NOT CASESPECIFIC NOT NULL,
   NumAMPs         INTEGER NOT NULL,
   NumPEs          INTEGER NOT NULL,
   NumNodes        INTEGER NOT NULL,
   ReleaseInfo     VARCHAR(20) NOT NULL,
   VersionInfo     VARCHAR(20) NOT NULL,
   PENum           INTEGER NOT NULL,
   QueryText       VARCHAR(20000) CHARACTER SET UNICODE
                   NOT CASESPECIFIC NOT NULL,
   DateTimeStamp   DATE FORMAT 'YY/MM/DD' NOT NULL,
   QueryName       VARCHAR(30) CHARACTER SET UNICODE
                   NOT CASESPECIFIC,
   Frequency       INTEGER NOT NULL,
   StatementTypes VARCHAR(120) NOT NULL,
   DefaultDBName   VARCHAR(30) CHARACTER SET UNICODE
                   NOT CASESPECIFIC NOT NULL,
   Overflow        CHARACTER(1) CHARACTER SET LATIN,
   Complete        CHARACTER(1) CHARACTER SET LATIN,
   ValidatedPlan   CHARACTER(1),
   ImportedPlan    CHARACTER(1))
UNIQUE PRIMARY INDEX PK_QueryID ( QueryID );
```

Attribute Definitions

The following table defines the Query table attributes.

Attribute	Definition
QueryID	• Unique identifier for the query generated by the system when the query plan is captured. • UPI for the table.
UDB_Key	• Identifier for the user or database containing the relation described by this row.

Attribute	Definition
MachName	Name of the test machine on which the query plan is captured.
NumAMPs	Number of AMPs in the configuration.
NumPEs	Number of PEs in the configuration.
NumNodes	Number of nodes in the configuration.
ReleaseInfo	Teradata release number under which the query was captured. The value is defined in DBC.DBCInfo.
VersionInfo	Teradata version number under which the query was captured. The value is defined in DBC.DBCInfo.
PENum	Number of the parsing engine on which the query was processed.
QueryText	The SQL DML text of the captured query. If the text exceeds the upper limit of 20 000 characters, then it overflows to the QryRelX table. See "Overflow" on page 4-42 and "QryRelX" on page 4-38.
DateTimeStamp	Timestamp that identifies when the captured query was performed. Useful for distinguishing among multiple performances of the same query on the same machine under the same software version and release.
QueryName	The name of the query, if provided, as specified in the AS clause.
Frequency	The number of times the query is performed in the workload to which it is assigned. The value is specified by the INSERT EXPLAIN statement used to create the row.

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StatementTypes</td><td>A comma-separated list of 3-character statement type codes.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>ABT</td><td>ABORT statement.</td></tr><tr><td>BTS</td><td>BEGIN TRANSACTION statement.</td></tr><tr><td>DEL</td><td>DELETE statement.</td></tr><tr><td>ETS</td><td>END TRANSACTION statement.</td></tr><tr><td>INS</td><td>INSERT statement.</td></tr><tr><td>NUL</td><td>Null statement.</td></tr><tr><td>OTR</td><td>Other statement.

This code describes any statement type that is not described by the other 9 statement type codes.</td></tr><tr><td>RET</td><td>Retrieve statement.</td></tr><tr><td>UPD</td><td>UPDATE statement.</td></tr><tr><td>URT</td><td>Update-Retrieve statement.</td></tr></table></td></tr>
<tr><td>DefaultDBName</td><td>The name of the default database at the time the query plan is captured.</td></tr>
<tr><td>Overflow</td><td>Specifies whether QueryText exceeds 20 000 characters.

If QueryText exceeds 20 000 characters, then all text beyond that boundary is truncated.

<table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>QueryText <= 20 000 characters.</td></tr><tr><td>T</td><td>QueryText > 20 000 characters and has been truncated.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>Complete</td><td>Identifies whether Query stores the complete query text or a truncated version.

<table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>Query text is truncated.

The upper boundary on the number of characters stored is controlled by the LIMIT clause of the DUMP EXPLAIN and INSERT EXPLAIN statements (see "DUMP EXPLAIN" and "INSERT EXPLAIN" in the chapter "SQL Data Definition Language Statement Syntax" in Teradata RDBMS SQL Reference, Volume 4).

If no limit is specified by DUMP EXPLAIN or INSERT EXPLAIN, then full query text is captured and stored.</td></tr><tr><td>T</td><td>Full query text is captured and stored.

If there is overflow (indicated when the value for the Overflow attribute is T), then it is stored in the QryRelX table (see “QryRelX” on page 4-38).</td></tr></table></td></tr>
<tr><td>ValidatedPlan</td><td>Specifies whether the plan was captured in validation or non-validation mode.

The value is always set to F during the capture.

<table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>Query plan was captured in non-validation mode.</td></tr><tr><td>T</td><td>Query plan was captured in a validation mode.</td></tr></table></td></tr>
<tr><td>ImportedPlan</td><td>Specifies whether the query plan was captured on the current system or imported from another system.

Teradata always sets the flag to F during the capture.

<table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>Query plan was captured on the current system.</td></tr><tr><td>T</td><td>Query plan was imported to the current system from another system.</td></tr></table></td></tr>
</table>

QuerySteps

Function

Each row in the table lists the attributes of any step executed by the system corresponding to the query. If the step has more than one attribute, then multiple rows are stored for the step and the row set is linked by means of its common StepID and QueryID values and the individual rows are distinguished by their row type codes.

Table Definition

The following CREATE TABLE statement defines the QuerySteps table.

```
CREATE TABLE QuerySteps (
 StepID              INTEGER NOT NULL,
 QueryID             INTEGER NOT NULL,
 StepNum             INTEGER,
 ParallelStepNum     INTEGER DEFAULT 0,
 StepText            VARCHAR(32000) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 RowType             CHAR(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC NOT NULL,
 StepKind            CHAR(2) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 ParallelKind        CHAR(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 AMPUsage            CHAR(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 TriggerType         CHAR(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 Cost                FLOAT,
 MaxCost             FLOAT,
 SourceRelation1     INTEGER,
 SourceRelation2     INTEGER,
 TargetRelation1     INTEGER,
 TargetRelation2     INTEGER,
 StepAttributeType   CHAR(10) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 StepAttributeValue VARCHAR(100) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 LockType            CHAR(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 RowHashFlag         CHAR(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 NoWaitFlag          CHAR(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC)
PRIMARY INDEX (QueryID)
INDEX SK_StepID ( StepID );
```

Attribute Definitions

The following table defines the QuerySteps table attributes.

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StepID</td><td>• Unique identifier for the step.
• NUSI for the table.</td></tr>
<tr><td>QueryID</td><td>• Unique identifier for the query.
• NUPI for the table.</td></tr>
<tr><td>StepNum</td><td>The number of the step whose text is reported by this QuerySteps row.

<table><tr><th>IF the step is ...</th><th>THEN the value is ...</th></tr><tr><td>not performed in parallel</td><td>the step number.</td></tr><tr><td>performed in parallel</td><td>the number of the main step.</td></tr></table></td></tr>
<tr><td>ParallelStepNum</td><td>The number of the parallel step whose text is reported by this QuerySteps row.

<table><tr><th>IF the step is ...</th><th>THEN the value is ...</th></tr><tr><td>not performed in parallel</td><td>0.</td></tr><tr><td>performed in parallel</td><td>the number of the parallel step.</td></tr></table></td></tr>
<tr><td>StepText</td><td>Stores text describing the step.</td></tr>
<tr><td>RowType</td><td>Describes the type of detail this row characterizes.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>A</td><td>The row describes attributes for a query plan step beyond the first row for that step, which is coded with a RowType of G.

The additional attributes are described by the StepAttributeType and StepAttributeValue fields. The remaining fields in an A row type are set null.</td></tr><tr><td>G</td><td>The row describes the first row of step information for a particular step of the query plan.

All steps have one row of this type.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StepKind</td><td>Describes the kind of step characterized by this row.

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>AB</td><td>Abort step.</td></tr>
<tr><td>BM</td><td>Bitmap step.</td></tr>
<tr><td>CE</td><td>Correlated inclusion merge join step.</td></tr>
<tr><td>CI</td><td>Correlated exclusion product join step.</td></tr>
<tr><td>CJ</td><td>Correlated inclusion production join step.</td></tr>
<tr><td>CP</td><td>Correlated exclusion merge join step.</td></tr>
<tr><td>DE</td><td>Delete step.</td></tr>
<tr><td>EJ</td><td>Exists join step.</td></tr>
<tr><td>EM</td><td>Exclusion merge join step.</td></tr>
<tr><td>EP</td><td>Exclusion production join step.</td></tr>
<tr><td>FD</td><td>Flush database step.</td></tr>
<tr><td>HJ</td><td>Hash join step.</td></tr>
<tr><td>HS</td><td>Hash star join step.</td></tr>
<tr><td>IJ</td><td>Intersect all join step.</td></tr>
<tr><td>IM</td><td>Inclusion merge join step.</td></tr>
<tr><td>IN</td><td>Insert step.</td></tr>
<tr><td>IP</td><td>Inclusion product join step.</td></tr>
<tr><td>LK</td><td>Lock step.</td></tr>
<tr><td>MD</td><td>Merge delete step.</td></tr>
<tr><td>MG</td><td>Merge step.</td></tr>
<tr><td>MI</td><td>Minus all join step.</td></tr>
<tr><td>MJ</td><td>Merge join step.</td></tr>
<tr><td>MS</td><td>Other step.

This code describes any type of step not described by the other StepKind codes.</td></tr>
<tr><td>MT</td><td>Materialize temporary table step.</td></tr>
<tr><td>MU</td><td>Merge update step.</td></tr>
<tr><td>NJ</td><td>Nested join step.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StepKind
(continued)</td><td>
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>OJ</td><td>Not exists join step.</td></tr>
<tr><td>PJ</td><td>Product join step.</td></tr>
<tr><td>RJ</td><td>Rowid join step.</td></tr>
<tr><td>SA</td><td>Sample step.</td></tr>
<tr><td>SO</td><td>Sort step.</td></tr>
<tr><td>SP</td><td>Spoil step.</td></tr>
<tr><td>SR</td><td>Single retrieve step.</td></tr>
<tr><td>ST</td><td>Statistics step.</td></tr>
<tr><td>SU</td><td>Sum retrieve step.</td></tr>
<tr><td>UP</td><td>Update step.</td></tr>
</table>
</td></tr>
<tr><td>ParallelKind</td><td>Describes whether the step can be done in parallel with its preceding step or not.
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>P</td><td>Step can be performed in parallel with the preceding step.</td></tr>
<tr><td>S</td><td>Step is performed sequentially and cannot be performed in parallel with the preceding step.</td></tr>
</table>
</td></tr>
<tr><td>AMPUsage</td><td>Describes how AMPs are used to process the step.
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>A</td><td>Step is an all-AMPs operation.</td></tr>
<tr><td>G</td><td>Step operates on a group of AMPs.</td></tr>
<tr><td>O</td><td>Step is a one-AMP operation.</td></tr>
<tr><td>T</td><td>Step is a two-AMP operation.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>TriggerType</td><td>Describes triggering associated with this step.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>C</td><td>Cascaded triggering statement.</td></tr><tr><td>N</td><td>No triggering involved with this statement.</td></tr><tr><td>R</td><td>Triggered statement.</td></tr><tr><td>T</td><td>Triggering statement.</td></tr></table></td></tr>
<tr><td>Cost</td><td>Estimated cost of performing this step.</td></tr>
<tr><td>MaxCost</td><td>Estimated worst case cost of performing this step.</td></tr>
<tr><td>SourceRelation1</td><td>Number of the principal relation source in this step.</td></tr>
<tr><td>SourceRelation2</td><td>Number of an additional relation source in this step.</td></tr>
<tr><td>TargetRelation1</td><td>Number of the spool file or table for which the step operation results or acts upon.</td></tr>
<tr><td>TargetRelation2</td><td>Number of an additional spool file, if one is required, to hold additional results of this step.</td></tr>
<tr><td>StepAttributeType</td><td>Indicates the attribute that is described by the row.

<table><tr><th>Attribute Type</th><th colspan="2">Attribute Value</th><th>Step Where Attribute Occurs</th></tr><tr><td rowspan="2">Globalflag</td><td>T</td><td>Flag set.</td><td>Sum retrieve</td></tr><tr><td>F</td><td>Flag not set.</td><td></td></tr><tr><td>IndexNum</td><td colspan="2">Index number.</td><td>• Abort
• BitMap
• Delete
• Single retrieve
Indicates whether the index was used.</td></tr><tr><td rowspan="4">JoinType</td><td>F</td><td>Full outer</td><td rowspan="4">Join type in the join step.</td></tr><tr><td>I</td><td>Inner</td></tr><tr><td>L</td><td>Left outer</td></tr><tr><td>R</td><td>Right outer</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StepAttributeType
(continued)</td><td>
<table>
<tr><th>Attribute Type</th><th colspan="2">Attribute Value</th><th>Step Where Attribute Occurs</th></tr>
<tr><td>Kind</td><td>F</td><td>Fraction</td><td rowspan="2">Indicates whether the samples are specified as a fraction or rows or as an absolute number of rows in the sample step.</td></tr>
<tr><td></td><td>I</td><td>Fixed</td></tr>
<tr><td>LeftIndex</td><td colspan="2">Index number.</td><td>Index used in the left relation in the join step.</td></tr>
<tr><td>MergeMode</td><td>H</td><td>Match row hash.</td><td rowspan="3">Merge delete and merge update steps indicating the merge type used in the step.</td></tr>
<tr><td></td><td>R</td><td>Match row ID.</td></tr>
<tr><td></td><td>W</td><td>Match whole row.</td></tr>
<tr><td>RightIndex</td><td colspan="2">Index number.</td><td>Index used in the right relation in the join step.</td></tr>
<tr><td>SMSKind</td><td colspan="2">Intersect</td><td rowspan="3">BitMap step indicating the set manipulation operation performed.</td></tr>
<tr><td></td><td colspan="2">Minus</td></tr>
<tr><td></td><td colspan="2">Union</td></tr>
<tr><td>SourceIndex</td><td colspan="2">Index number.</td><td>Index used in the source relation in the Stat step.</td></tr>
<tr><td>Svalue</td><td colspan="2">Sample size.</td><td>Sample specified in the Sample step.</td></tr>
<tr><td>TableIndex</td><td colspan="2">Index number.</td><td>Update step indicating the index the step used.</td></tr>
<tr><td>True or False</td><td colspan="2">Text containing the error returned with the abort.</td><td>Abort step indicating whether to abort when the condition is true or when the condition is false.</td></tr>
</table>
</td></tr>
<tr><td>StepAttributeValue</td><td>Indicates the value of the attribute described by the row.</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>LockType</td><td>Defines the type of lock specified by the query plan for this step.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>A</td><td>Access lock.</td></tr><tr><td>R</td><td>Read lock.</td></tr><tr><td>W</td><td>Write lock.</td></tr><tr><td>X</td><td>Exclusive lock.</td></tr></table></td></tr>
<tr><td>RowHashFlag</td><td>Indicates if the table is locked on a row hash.

Used to specify if the table is locked on a row hash or not. It can be Y or N

<table><tr><th>Code</th><th>Description</th></tr><tr><td>N</td><td>The table is not locked on a row hash.</td></tr><tr><td>Y</td><td>The table is locked on a row hash.</td></tr></table></td></tr>
<tr><td>NoWaitFlag</td><td>Indicates if the no wait option is set for the lock step.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>N</td><td>The no wait option is disabled.</td></tr><tr><td>Y</td><td>The no wait option is enabled.</td></tr></table></td></tr>
</table>

Relation

Function

Describes all table and spool files in the access plan for the captured query.

Table Definition

The following CREATE TABLE statement defines the Relation table.

```
CREATE TABLE Relation(
   RelationKey    INTEGER NOT NULL,
   QueryID        INTEGER NOT NULL,
   UDB_Key        INTEGER NOT NULL,
   Name           VARCHAR(30) CHARACTER SET UNICODE
                  NOT CASESPECIFIC NOT NULL,
   RelationID     INTEGER NOT NULL,
   RelationKind   CHARACTER(1) NOT NULL,
   SortInfo       CHARACTER(1),
   SortKind       CHARACTER(3),
   SortKey        VARCHAR(1024),
   GeogInfo       CHARACTER(1),
   Cached         CHARACTER(1) NOT NULL,
   SyncScan       CHARACTER(1) NOT NULL,
   Cardinality    REAL,
   Confidence     CHARACTER(1),
   MaxCardinality REAL,
   ViewName       VARCHAR(30) CHARACTER SET UNICODE
                  UPPERCASE NOT CASESPECIFIC,
   TableDDL       VARCHAR(20000) CHARACTER SET UNICODE
                  NOT CASESPECIFIC,
   TableName      VARCHAR(30) CHARACTER SET UNICODE
                  NOT CASESPECIFIC,
   PartitionInfo  CHARACTER(1) NOT NULL,
   Overflow       CHARACTER(1) CHARACTER SET LATIN,
   Complete       CHARACTER(1) CHARACTER SET LATIN,
   Version        SMALLINT)
PRIMARY INDEX (QueryID),
UNIQUE INDEX (RelationKey);
```

Attribute Definitions

The following table defines the Relation table attributes.

Attribute	Definition
RelationKey	• Unique identifier for the relation within its database. • USI for the Relation table.
QueryID	• Unique identifier for the query generated by the system when the query plan is captured. • NUPI for the Relation table.
UDB_Key	Identifier for the user or database containing the relation described by this row.
Name	Either of two things: • Alias name of the table. • Spool ID of a spool.
RelationID	Unique identifier for the relation within a certain database.
RelationKind	Distinguishes among base tables, derived tables, and spool files. **Code** / **Description** D – Relation is a derived table. The Name attribute contains the name of the derived table. S – Relation is a spool file. T – Relation is a table.
SortInfo	Describes whether the relation is sorted or not. **Code** / **Description** F – The relation is not sorted. T – The relation is sorted.

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>SortKind</td><td>The way the relation is sorted.
Only used when the value for SortInfo is T.

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>F1S</td><td>Field1 sort.</td></tr>
<tr><td>F1U</td><td>Field1 unique sort.</td></tr>
<tr><td>FHS</td><td>Field1 Hash sort.</td></tr>
<tr><td>FID</td><td>FieldID sort.</td></tr>
<tr><td>FHU</td><td>Field1 Hash unique sort.</td></tr>
<tr><td>HN1</td><td>Field1 Hash min1 sort.</td></tr>
<tr><td>HN2</td><td>Field1 Hash min2 sort.</td></tr>
<tr><td>HX1</td><td>Field1 Hash max1 sort.</td></tr>
<tr><td>HX2</td><td>Field1 Hash max2 sort.</td></tr>
<tr><td>JIS</td><td>JoinIndex sort.</td></tr>
<tr><td>MN1</td><td>Field1 min1 sort.</td></tr>
<tr><td>MN2</td><td>Field1 min2 sort.</td></tr>
<tr><td>MX1</td><td>Field1 max1 sort.</td></tr>
<tr><td>MX2</td><td>Field1 max2 sort.</td></tr>
<tr><td>RF1</td><td>Rowhash field1 sort.</td></tr>
<tr><td>RHR</td><td>RowHashRow sort.</td></tr>
<tr><td>RHS</td><td>Rowhash sort.</td></tr>
<tr><td>UF1</td><td>Unique field1 sort.</td></tr>
<tr><td>UNK</td><td>Unknown sort kind.</td></tr>
<tr><td>URS</td><td>Unique rowID sort.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>SortKey</td><td>A list of sort information strings composed of the concatenated database, table, and column names that make up the sort key. Only the first 1 024 characters are captured: any remaining characters are truncated.

The format for the SortKey list is the following:

<code>SortKey1, SortKey2, …, SortKeyn</code>

There must be a SPACE character following each COMMA character in the list.

The format for the individual SortKey strings is one of the following:

<code>"database_name"."table_name"."column_name"</code>

<code>"SPOOL number"."column_name"</code>

Spool numbers are used in place of database.table names whenever the table information is not available.</td></tr>
<tr><td>GeogInfo</td><td>Describes the configuration geography of the relation.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>D</td><td>Relation is duplicated on all AMPs.</td></tr><tr><td>L</td><td>Relation is built locally.</td></tr><tr><td>R</td><td>Relation is hash-distributed.</td></tr></table></td></tr>
<tr><td>Cached</td><td>Describes whether the relation is cached or not.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Relation is not cached.</td></tr><tr><td>T</td><td>Relation is cached.</td></tr></table></td></tr>
<tr><td>SyncScan</td><td>Describes whether a relation is eligible for synchronized scanning or not.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Relation is not eligible for synchronized scanning.</td></tr><tr><td>T</td><td>Relation is eligible for synchronized scanning.</td></tr></table></td></tr>
<tr><td>Cardinality</td><td>Estimated cardinality of the relation.</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>Confidence</td><td>Describes the confidence level for the estimated cardinality.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>H</td><td>High confidence.</td></tr><tr><td>I</td><td>Index join confidence.</td></tr><tr><td>L</td><td>Low confidence.</td></tr><tr><td>N</td><td>No confidence.</td></tr></table></td></tr>
<tr><td>MaxCardinality</td><td>Estimated maximum cardinality of the relation.</td></tr>
<tr><td>ViewName</td><td>Name of a view, if any, used by the query to access the relation.</td></tr>
<tr><td>TableDDL</td><td>The SQL DDL text for the captured relation.

If the text exceeds the upper limit of 20 000 characters, then it overflows to the QryRelX table. See “Overflow” on page 4-55 and “QryRelX” on page 4-38.</td></tr>
<tr><td>TableName</td><td>The non-aliased name of the relation.

Compare with “Name” on page 4-52.</td></tr>
<tr><td>PartitionInfo</td><td>Identifies whether a table or spool has a partitioned primary index.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Relation does not have a partitioned primary index.</td></tr><tr><td>T</td><td>Relation has a partitioned primary index.</td></tr></table></td></tr>
<tr><td>Overflow</td><td>Specifies whether there is overflow query text stored in QryRelX or not.

The default value is F.

<table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>There is no overflow.</td></tr><tr><td>T</td><td>Overflow text is stored in a QryRelX table.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>Complete</td><td>Identifies whether Relation stores the complete relation DDL or a truncated version.

<table><tr><th>Flag</th><th>Definition</th></tr><tr><td>F</td><td>Table DDL is truncated.

The upper boundary on the number of characters stored is controlled by the LIMIT clause of the DUMP EXPLAIN and INSERT EXPLAIN statements (see "DUMP EXPLAIN" and "INSERT EXPLAIN" in the chapter "SQL Data Definition Language Statement Syntax" in Teradata RDBMS SQL Reference, Volume 4).

If no limit is specified by DUMP EXPLAIN or INSERT EXPLAIN, then full DDL is captured and stored.</td></tr><tr><td>T</td><td>Full Table DDL is stored.

If there is overflow (indicated when the value for the Overflow attribute is T), then it is stored in the QryRelX table (see “QryRelX” on page 4-38).</td></tr></table></td></tr>
<tr><td>Version</td><td>Stores the version number of the table at the time the plan was captured.

Used to ensure that any changes to the schema information captured for analysis are handled correctly.</td></tr>
</table>

SeqNumber

Function

Generates sequential alternate keys to be used by other QCD tables.

Each column in the table (except QCDVersion, DemographicsID, and StatisticsID) is initialized to 1 and then incremented after each use.

Table Definition

The following CREATE TABLE statement defines the SeqNumber table.

```
CREATE TABLE SeqNumber (
 PIndex           INTEGER NOT NULL,
 WorkLoadID       INTEGER NOT NULL,
 MachConfigID     INTEGER NOT NULL,
 QueryID          INTEGER NOT NULL,
 UDB_KEY          INTEGER NOT NULL,
 StepID           INTEGER NOT NULL,
 RelationKey      INTEGER NOT NULL,
 PredicateID      INTEGER NOT NULL,
 RecommendationID INTEGER NOT NULL,
 QCDVersion       VARCHAR(30) CHARACTER SET LATIN
                  NOT CASESPECIFIC,
 DemographicsID   INTEGER NOT NULL,
 StatisticsID     INTEGER NOT NULL
)
PRIMARY INDEX (PIndex)
UNIQUE INDEX (RelationKey);
```

Table Initialization Statement

SeqNumber is initialized by the following INSERT statement:

```
INSERT INTO SeqNumber VALUES(1,1,1,1,1,1,1,1,1,'QCF03.00.00',2,2);
```

Attribute Definitions

The following table defines the SeqNumber table attributes.

Attribute	Definition
PIndex	An artificial column used as the NUPI for this table
WorkLoadID	Unique identifier for the workload.
MachConfigID	Unique identifier for the configuration of a particular hardware configuration stored in QCD.
QueryID	Unique identifier for the query.

Attribute	Definition
UDB_KEY	Unique identifier on MachConfigID for the user or database that performed the query.
StepID	Unique identifier for a particular AMP step in the query.
RelationKey	Unique identifier for a table, derived table, or spool file used in the query.
PredicateID	Unique identifier for the predicate used in the query.
RecommendationID	Unique identifier for a particular set of index recommendations.
QCDVersion	Describes the version of QCD currently running.
DemographicsID	Identifies whether the demographics were captured on the current system or imported from another system. Initially set to 2.
StatisticsID	Identifies whether the statistics were captured by a COLLECT STATISTICS (QCD Form) or INSERT EXPLAIN statement. Initially set to 2.

TableStatistics

Function

Captures the statistics on a data sample for all the columns that are possible index candidates in the specified query.

TableStatistics also captures the detail statistics if you perform the SQL COLLECT STATISTICS (QCD Form) statement.

Table Definition

The following CREATE TABLE statement defines the TableStatistics table.

```
CREATE TABLE TableStatistics (
 MachineName    VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 TableName      VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 DatabaseName   VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 IndexName      VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC,
 ColumnName     VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 CollectedTime  TIMESTAMP(6) NOT NULL,
 SamplePercent  FLOAT NOT NULL,
 IndexType      CHAR(1) NOT NULL,
 IndexID        INTEGER,
 StatisticsInfo VARBYTE(16383),
 DataType       SMALLINT,
 DataLength     INTEGER,
 Attributes     VARCHAR(128),
 ModifiedTime   TIMESTAMP(6),
 ModifiedStats  VARBYTE(16383),
 QueryId        INTEGER,
 FieldPosition  BYTEINT,
 StatisticsID   INTEGER)
PRIMARY INDEX (MachineName, DatabaseName, TableName);
```

Attribute Definitions

The following table defines the TableStatistics table attributes.

Attribute	Definition
MachineName	The name of the system to which the table belongs.
TableName	Name of the table for which the row defines statistics.

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>DatabaseName</td><td>Name of the containing database or user for TableName.</td></tr>
<tr><td>IndexName</td><td>Name of the index, in case the index is named. Otherwise, IndexName is NULL.</td></tr>
<tr><td>ColumnName</td><td>Name of the column if the statistics are on a column.</td></tr>
<tr><td>CollectedTime</td><td>The timestamp value when the statistics were collected.</td></tr>
<tr><td>SamplePercent</td><td>The sample percentage of rows read to collect the statistics recorded in this row.</td></tr>
<tr><td>IndexType</td><td>Flag indicating whether the statistics represented by this row pertain to an index.

<table><tr><th>Code</th><th>Description</th></tr><tr><td>N</td><td>Statistics are for a non-indexed column.</td></tr><tr><td>P</td><td>Statistics are for a pseudo-index.</td></tr><tr><td>Y</td><td>Statistics are for the index specified by IndexID.</td></tr></table></td></tr>
<tr><td>IndexID</td><td>The unique identifier for the index to which the statistics represented by this row pertain.

<table><tr><th>IF IndexType has this value ...</th><th>THEN IndexID is ...</th></tr><tr><td>N</td><td>null.</td></tr><tr><td>P</td><td>the ID value generated by scanning the existing index set for the table and incrementing the largest value.</td></tr><tr><td>Y</td><td>the ID value assigned by the system.</td></tr></table></td></tr>
<tr><td>StatisticsInfo</td><td>Contains the column or index statistics.</td></tr>
<tr><td>DataType</td><td>Indicates the data type of the column. The value is the same that is returned in the datatype field in the PrepInfo CLIv2 parcel.</td></tr>
<tr><td>DataLength</td><td>The maximum length of the column in bytes.</td></tr>
<tr><td>Attributes</td><td>Set null and reserved for future use.</td></tr>
<tr><td>ModifiedTime</td><td>If the statistics have been modified, indicates the timestamp value when the last modification occurred.</td></tr>
</table>

Attribute	Definition
ModifiedStats	Contains user-modified column or index statistics. These statistics are used by the INITIATE INDEX ANALYSIS statement instead of the statistics in the StatisticsInfo column when you specify the USE MODIFIED STATISTICS option. You create the statistics in ModifiedStats using the "what if" features of the Teradata Index Wizard utility.
QueryID	The unique ID of the query. Set null when statistics are collected using COLLECT STATISTICS (QCD Form).
FieldPosition	Indicates the right-to-left position of the column within a composite index. Statistics are recorded in the row corresponding to FieldPosition 1 only. The value for other field positions are set null.
StatisticsID	Set to 1 if the demographics are captured by a COLLECT STATISTICS (QCD Form) or INSERT EXPLAIN statement. StatisticsID has different values if the demographics are imported rather than captured directly.

User_Database

Function

Describes User and Database identifiers to capture the identity of the user who submitted the query, the names of databases used in the query plan, and so on.

Table Definition

The following CREATE TABLE statement defines the User_Database table.

```
CREATE TABLE User_Database(
 UDB_Key     INTEGER NOT NULL,
 UDB_ID      INTEGER NOT NULL,
 MachineName VARCHAR(30) CHARACTER SET UNICODE
             NOT CASESPECIFIC NOT NULL,
 UDB_Name    VARCHAR(30) CHARACTER SET UNICODE
             NOT CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX PK_UDB_KEY ( UDB_Key );
```

Attribute Definitions

The following table defines the User_Database table attributes.

Attribute	Definition
UDB_Key	• Unique identifier for the user or database. • UPI for the table.
UDB_ID	Unique identifier for the user or database on the system on which the captured query was performed.
MachineName	Name of the production system on which the database identified by UDB_ID exists.
UDB_Name	Name of the user or database identified by UDB_ID.

UserRemarks

Function

Stores user comments about captured plans written using either the TSET or Visual EXPLAIN client utilities.

Table Definition

The following CREATE TABLE statement defines the UserRemarks table.

```
CREATE TABLE UserRemarks(
 QueryID     INTEGER,
 StepID      INTEGER,
 WorkloadID  INTEGER,
 RowType     CHAR(1) NOT NULL,
 SeqNumber   INTEGER NOT NULL,
 UpdTime     TIMESTAMP(6) NOT NULL,
 UserName    VARCHAR(30) CHARACTER SET UNICODE
                         NOT CASESPECIFIC NOT NULL,
 Remarks     VARCHAR(32000) CHARACTER SET UNICODE
                            NOT CASESPECIFIC NOT NULL)
PRIMARY INDEX (QueryID, UserName);
```

Attribute Definitions

The following table defines the UserRemarks table attributes.

Attribute	Definition
QueryID	The ID of the query about which remarks are being saved. Partial NUPI for the table.
StepID	The ID of the step about which remarks are being saved.
WorkloadID	The ID of the workload to which the query belongs.
RowType	Specifies which utility collected the remarks and whether they were saved or generated.
SeqNumber	A system-generated number (1 for the first row, 2 for second, and so on) to indicate the sequential order of remarks.

Code	Description
T	Comments generated automatically by the TSET utility during import.
V	Comments saved by the Visual EXPLAIN utility.

Attribute	Definition
UpdTime	The timestamp when the remark was last updated.
UserName	The UserName of the user who last updated the row. Partial NUPI for the table.
Remarks	The remarks of the step or query.

ViewTable

Function

Captures the DDL for any views used in a query.

Table Definition

The following CREATE TABLE statement defines ViewTable.

```
CREATE SET TABLE ViewTable(
 QueryID   INTEGER NOT NULL,
 ViewName  VARCHAR(30) CHARACTER SET UNICODE
           UPPERCASE NOT CASESPECIFIC NOT NULL,
 DBName    VARCHAR(30) CHARACTER SET UNICODE
           UPPERCASE NOT CASESPECIFIC NOT NULL,
 ViewText  VARCHAR(30000) CHARACTER SET UNICODE
           NOT CASESPECIFIC NOT NULL,
 SeqNumber BYTEINT NOT NULL,
 Complete  CHARACTER(1))
PRIMARY INDEX QueryID_ViewName_DBName (QueryID,ViewName,DBName);
```

Attribute Definitions

The following table defines the ViewTable attributes.

Attribute	Definition
QueryID	• Unique identifier for the query. • Partial NUPI for the table.
ViewName	• Name of the view. • Partial NUPI for the table.
DBName	• Name of the database. • Partial NUPI for the table.

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>ViewText</td><td>Stores the DDL view text.

<table><tr><th>IF ViewText is ...</th><th>THEN SeqNumber is set to ...</th></tr><tr><td><= 30 000 characters</td><td>1</td></tr><tr><td>> 30 000 characters</td><td>1 + (1-n), where n is the identifier for the fragment of ViewText stored in the row following the first fragment</td></tr></table>
When ViewText > 30 000 characters, only the first 30 000 are stored and the remaining text is stored in overflow rows within ViewTable. The overflow rows are identified by their SeqNumber value.</td></tr>
<tr><td>SeqNumber</td><td>The sequence of ViewText stored in this row.</td></tr>
<tr><td>Complete</td><td>Identifies whether ViewTable stores the complete view DDL or a truncated version.

<table><tr><th>Flag</th><th>Definition</th></tr><tr><td>F</td><td>ViewText DDL is truncated.

This case occurs when the size specified in the LIMIT SQL clause of the DUMP EXPLAIN or INSERT EXPLAIN statement that captured the view is less than the actual length of ViewText.

See "DUMP EXPLAIN" and "INSERT EXPLAIN" in the chapter "SQL Data Definition Language Statement Syntax" in Teradata RDBMS SQL Reference, Volume 4.</td></tr><tr><td>T</td><td>Full ViewText DDL is stored.

Overflow ViewText DDL is stored in additional rows in ViewTable. Individual overflow rows are identified by their respective SeqNumber.</td></tr></table></td></tr>
</table>

Workload

Function

Workload table has one entry for each workload defined in the database.

Table Definition

The following CREATE TABLE statement defines the Workload table.

```
CREATE TABLE Workload(
 WorkloadID       INTEGER NOT NULL,
 WorkloadName     VARCHAR(30) CHARACTER SET UNICODE
                  UPPERCASE NOT CASESPECIFIC NOT NULL,
 CreatedTimeStamp TIMESTAMP(6) NOT NULL,
 LastModified     TIMESTAMP(6) NOT NULL
 CreatorName      VARCHAR(30) CHARACTER SET UNICODE
                  NOT CASESPECIFIC)
UNIQUE PRIMARY INDEX(WorkLoadName),
UNIQUE INDEX(WorkLoadID);
```

Attribute Definitions

The following table defines the Workload table attributes.

Attribute	Definition
WorkloadID	• Internally generated unique (within the database) identifier for the workload. • USI for the table.
WorkloadName	• User-specified unique workload name. • UPI for the table.
CreatedTimeStamp	Timestamp when the workload was created.
LastModified	Timestamp when the workload was last modified.
CreatorName	Name of the user that created the workload.

WorkloadQueries

Function

Captures the queries that belong to the workload named by WorkloadID.

Table Definition

The following CREATE TABLE statement defines the WorkloadQueries table.

```
CREATE TABLE WorkloadQueries(
 WorkloadID           INTEGER NOT NULL,
 QueryID              INTEGER NOT NULL,
 Frequency            INTEGER NOT NULL)
PRIMARY INDEX(WorkLoadID)
UNIQUE INDEX(WorkloadID, QueryID);
```

Attribute Definitions

The following table defines the WorkloadQueries table attributes.

Attribute	Definition
WorkloadID	• The workload ID that identifies the data in the row. • NUPI for the table. • Partial USI for the table.
QueryID	• Unique qualifier for a query. Used to map it to the workload. • Partial USI for the table.
Frequency	Indicates the number of times the query is typically performed in the workload.

WorkloadStatus

Function

Maintains the status of workloads in the QCD.

This table is maintained by the Teradata Index Wizard.

Table Definition

The following CREATE TABLE statement defines the WorkloadStatus table.

```
CREATE TABLE WorkloadStatus (
 WorkloadID         INTEGER NOT NULL,
 RecommendationID   INTEGER,
 IndexNameTag       VARCHAR(30) CHARACTER SET UNICODE
                    NOT CASESPECIFIC,
 ValidatedSystem    VARCHAR(30) CHARACTER SET UNICODE
                    NOT CASESPECIFIC,
 ValidatedTimeStamp TIMESTAMP(6),
 ValidatedQCD       VARCHAR(30) CHARACTER SET UNICODE
                    NOT CASESPECIFIC,
 RecosAppliedSystem VARCHAR(30) CHARACTER SET UNICODE
                    NOT CASESPECIFIC,
 Remarks            VARCHAR(20000) CHARACTER SET UNICODE
                    NOT CASESPECIFIC)
UNIQUE PRIMARY INDEX (WorkloadID ,IndexNameTag);
```

Attribute Definitions

The following table defines the WorkloadStatus table attributes.

Attribute	Definition
WorkloadID	• Unique identifier for the workload. • Partial UPI for the table.
RecommendationID	Unique identifier for the set of indexes recommended for this workload after an index analysis.
IndexNameTag	• Name of the index recommendation as specified in the INITIATE INDEX ANALYSIS statement (see "INITIATE INDEX ANALYSIS" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*). • Partial UPI for the table.
ValidatedSystem	Name of the system on which index recommendations are last validated by the Teradata Index Wizard.

Attribute	Definition
ValidatedTimeStamp	Timestamp of the index validation.
ValidatedQCD	Name of the QCD on which the validation is performed by the Teradata Index Wizard.
RecosAppliedSystem	Name of the system on which index recommendations are last created by the Teradata Index Wizard.
Remarks	Free form textual remarks about the workload.

Chapter 5:

Database Foundations for the Teradata Index Wizard

This chapter describes the Teradata server-based architecture for the Teradata Index Wizard utility.

The Teradata Index Wizard helps you to reengineer existing databases by providing a method to propose optimal sets of secondary indexes to support particular SQL query workloads. This process permits you to analyze the potential performance enhancing effects of various secondary indexes on statistical representations of live data from your data warehouse.

In most cases, the data was probably not available at the time you performed the physical design for the database, so the effects of the secondary indexes you defined to support your query workloads were guessed at rather than tested empirically.

Loosely speaking, the Teradata Index Wizard tool permits you to revisit the Activity Transaction Modeling (ATM) phase of the transition from logical design to physical design (see the chapter "The Activity Transaction Modeling Process" in *Teradata RDBMS Database Design* for more information on ATM).

Detailed information about related tools and procedures is found in the following sources:

- Chapter 4: "Query Capture Facility"
- Chapter 6: "Target Level Emulation"
- "COLLECT DEMOGRAPHICS" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.
- "COLLECT STATISTICS (QCD Form)" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.
- "INITIATE INDEX ANALYSIS" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.
- "DUMP EXPLAIN" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.
- "INSERT EXPLAIN" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.
- "RESTART INDEX ANALYSIS" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.
- *Teradata Index Wizard User Guide*.
- Online help for Teradata Index Wizard utility.

- *Teradata Statistics Wizard User Guide*.
- Online help for Teradata Statistics Wizard utility.
- *Teradata Visual Explain User Guide.*
- Online help for Teradata Visual EXPLAIN utility.
- *Teradata System Emulation Tool User Guide.*
- Online help for Teradata System Emulation utility.
- *Teradata Manager User Guide.*
- Online help for Statistics Collection tool of Teradata Manager.

Teradata Index Wizard Overview

Introduction

The Teradata Index Wizard is used to reengineer or redesign an existing database by analyzing specific SQL workloads for opportunities to optimize performance through appropriate secondary index assignment. The analysis can produce recommendations to delete, as well as to add, indexes to the existing physical design.

General Procedure for Using the Teradata Index Wizard to Optimize Index Selection

The following table outlines the general procedure followed in performing index analysis using the Teradata Index Wizard:

Step	Activity Name	Activity Description
1	Identify workload.	Identify a set of SQL statements that constitute a workload. The following repositories are potential sources for identifying workload components: • Database Query Log (see "BEGIN QUERY LOGGING" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.) • Query Capture Database (see Chapter 4: "Query Capture Facility"). See "Workload Identification" on page 5-5 for details.
2	Define workload.	Perform the AddWorkload macro to define a new workload in a query capture database. See "Workload Definition" on page 5-8 for details.
3	Analyze indexes.	Perform an index analysis on the defined workload to produce a set of index recommendations. See "Index Analysis" on page 5-11 for details.
4	Validate indexes.	Validate the recommended indexes on your production system. This is an optional step in the process. See "Index Validation" on page 5-16 and "DIAGNOSTIC 'Validate Index'" in the chapter "Query and Workload Analysis Statements in *Teradata RDBMS SQL Reference, Volume 4* for details.

Step	Activity Name	Activity Description
5	Create indexes.	Apply the index recommendations to your production system. This is an optional step in the process. See "Index Application" on page 5-18 for details.

Constraints on the Output of the Teradata Index Wizard

The following constraints and restrictions apply to the use of the Index Wizard:

- By default, any query search conditions that default to full table scans cannot benefit from indexes; therefore, the Teradata Index Wizard does not generate index recommendations for such queries.
- If you generate index recommendations on a test system without emulating the production environment, then the index recommendations might not produce an optimal result when applied to the production system.

  You should always use Target Level Emulation (see Chapter 6: "Target Level Emulation") to emulate production environments when performing index analyses on a test system.
- The Teradata Index Wizard produces recommendations for secondary indexes only.

Workload Identification

Introduction

The objective of workload identification is to isolate the set of queries, called a workload, you want to analyze for the possibility of enhancing their performance by redefining the secondary indexes defined on the tables they access.

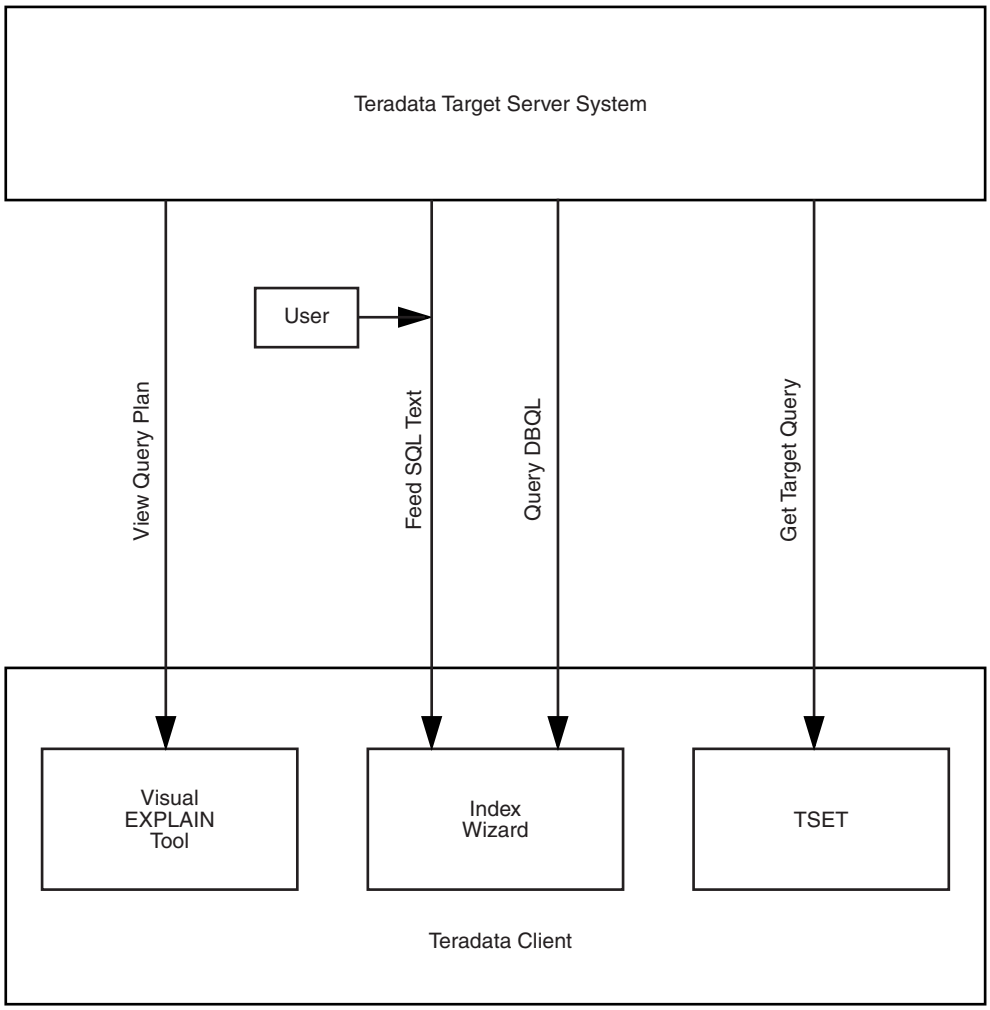
There are several methods for identifying workloads, including these:

- Ad hoc qualitative assessment of the performance of some queries. This is the “something is not right here” approach.
- Sift through the following data stores in search of candidate queries for performance enhancement:
  - The Database Query Log using the Teradata Index Wizard.
  - The Query Capture Database using tools like Visual EXPLAIN or Teradata System Emulation Tool (TSET).

After determining the set of queries to be analyzed, you can identify the workload using the Teradata Index Wizard tool.

Workload Identification Process

The process of identifying workloads is indicated by the following graphic.



GG02A006

Key	
Input Option	**Description**
View query plan	You can invoke the Index Wizard from a query plan viewed by means of the Visual EXPLAIN tool. The query used for index analysis is the active plan in Visual EXPLAIN at the time the Index Wizard is invoked.
Feed SQL text	You can directly type the text for the queries that are to make up a workload or you can load them from the server.

Key	
Input Option	**Description**
Query DBQL	You can fetch logged Database Query Log SQL text for index analysis by means of the Index Wizard.
Get target query	You can export a set of queries for analysis using TSET.

Workload Definition

Introduction

The objective of workload definition is to capture the set of query plans associated with the queries identified in the workload identification phase and define them to the Teradata Index Wizard.

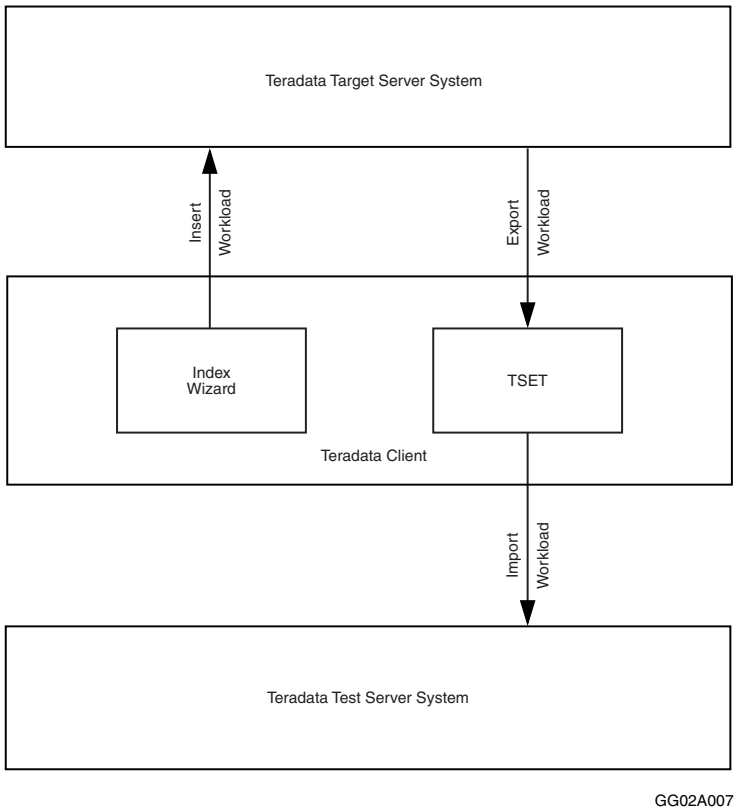
Workloads are defined to the Teradata Index Wizard using the query plans for the queries identified as a workload. These query plans must first be captured in the QCD using the INSERT EXPLAIN WITH STATISTICS statement (see "INSERT EXPLAIN" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*).

Workload Definition Macros

Teradata supplies a QCD with a set of macros to help you define workloads, modify those workloads, and delete them when they are no longer useful. These macros are normally invoked using the Teradata Index Wizard utility.

Workload Definition Process

The process of defining SQL workloads for subsequent index analysis is indicated by the following graphic:



Key	
Action	**Description**
Insert workload	The workload is defined in the specified QCD on the production system.
Export workload[a]	Once it has been created, the workload can be exported from the production system to the test system using TSET.
Import workload	In the second stage of this process, the workload is imported to the test system from the production system using either TSET or the Teradata Index Wizard.

a. These actions are optional and are relevant only if you intend to perform workload analyses on a test system.

Because workload analysis is very resource intensive, particularly with respect to memory and messaging, you should carefully evaluate whether to perform the analyses on a production system or, instead, to perform them on a test system using the Target Level Emulation facility (see Chapter 6: “Target Level Emulation”).

Mapping ATM Activities to Defining a Workload

Many phases of the Activity Transaction Modeling process (see the chapter "The Activity Transaction Modeling Process" in *Teradata RDBMS Database Design*) map well to the reengineering phases of workload definition. The following table indicates those mappings:

ATM Activity	Workload Definition Activity
Identify and define attribute domains and constraints for physical columns.	Define workload on tables and databases whose definitions exist in the data dictionary of a production environment. Because they already exist, they need not be identified.
Identify and model database applications.	Not applicable.
Model application processing activities to include their transactions and run frequencies.	Submit the SQL requests as transactions and run frequencies using the INSERT EXPLAIN statement. You determine the run frequency, which is a measure of how often the query is executed in the workload. Assume that DBQL queries are used as the workload input. In case a query is executed repeatedly, DBQL logs the query as many times as it is submitted. The Index Wizard client interface helps you to derive the run frequency information as the total count of the entries for the same queries logged in DBQL.
Model transactions to identify the tables used and the columns required for value or join access and estimating cardinalities	The workload analysis includes this activity. For the SQL requests submitted as workload, the system assimilates the information and stores it in a Query Capture Database.
Summarize value and join access information across all transactions.	The information stored in QCD as part of workload analysis is used to summarize the information.
Compile a preliminary set of data demographics by estimating table cardinalities and value distributions and assigning change ratings.	The workload analysis includes this activity. For the tables referenced in the SQL requests submitted as workload, the system assimilates the demographics information and stores it in the QCD. The processing includes a COLLECT STATISTICS (QCD Form) statement performed to obtain the information on columns identified as index candidates.

Index Analysis

Introduction

The objective of Index Analysis is to recommend a set of CREATE INDEX or DROP INDEX statements (or both) on the different tables referenced in the workload that provides the optimal improvement in the response time of that workload.

How Index Analyses Are Done

The following table describes the process used by the Index Wizard to perform an index analysis.

Stage	Process
1	The candidate search space and workload proposed index list are set null.
2	The workload candidate search space is built as follows:

Stage	Process
1	For each SQL statement in the workload:

Stage	Process
1	Regenerate the query plan. During this process, the Predicate Analyzer is invoked to build the statement candidate index list using the Candidate Index Enumerator.
2	Invoke the Index Search Engine to filter candidate indexes. To perform this task, the Index Search Engine analyzes the statement index candidate list, and removes the least promising indexes, producing the resultant statement proposed index list.
3	Add the statement proposed index list to the workload candidate search space.

Stage	Process
2 cont.	

Stage 2 cont.

Stage	Process
1 cont.	

Stage 1 cont.

Stage	Process
4	Determine if any table constraints have been violated.

IF a table constraint violation is …	THEN …
found	**1** Invoke the Index Search Engine to filter candidate indexes from the workload candidate search space. **2** Build a new workload proposed index list. **3** Invoke the Query Cost Analyzer to regenerate the query plan and compute costs by simulating the indexes identified as candidates by the Index Search Engine. Go to Stage 2.1.4.
not found	Go to Stage 2.2.

Stage	Process
5	Set workload candidate search space to workload proposed index list.

Stage 2

IF a checkpoint …	AND the specified checkpoint frequency…	THEN …
is specified	is hit	write the workload proposed index list to the AnalysisLog table.
not specififed		go to Stage 3.0

Stage	Process
3	Filter the workload proposed index list using the best indexes analyzed so far to produce the index recommendations.
4	Delete any remaining AnalysisLog entries.

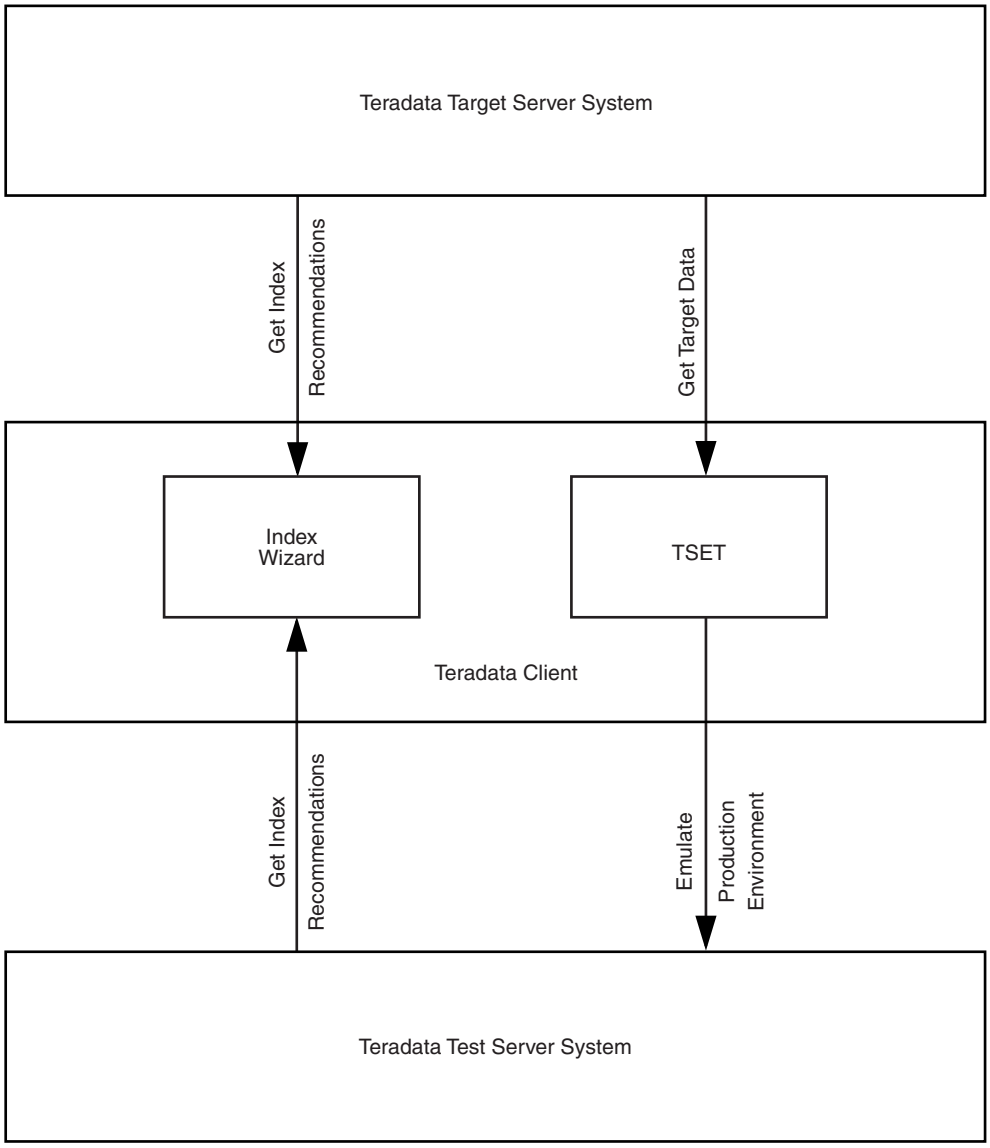
You can specify various index analysis parameters to control the choice of candidate indexes using the various boundary options of the INITIATE INDEX ANALYSIS statement (see “INITIATE INDEX ANALYSIS” in the chapter “Query and Workload Analysis Statements” in *Teradata RDBMS SQL Reference, Volume 4* for details). These boundary options only affect the index analysis phase of database query analysis, however, and not the query plan generated by the Optimizer.

Note that index recommendations are not restricted to just recommended indexes: they can also include recommendations to drop existing indexes. Index recommendations never include volatile tables that are accessed by the SQL statements making up the workloads

Index Analysis Process

The process of analyzing statistical data in order to develop a set of recommended indexes is indicated by the following graphic.

Note that if the analysis is performed on a production system, the two Get Index Recommendations steps collapse into a single step.



GG02A008

Key	
Activity	**Description**
Get target data	Use TSET to collect production system data such as environmental cost parameters and random AMP statistical samples.
Emulate the production environment	Use the Teradata Index Wizard client interface to set up the production system data at session level on a test system so it can emulate the production environment.
Get index recommendations	Perform the index analysis on the test system. Use the INITIATE INDEX ANALYSIS statement (see "INITIATE INDEX ANALYSIS" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*) to submit the SQL statement set to be analyzed. INITIATE INDEX ANALYSIS collects the index recommendations generated and saves them in the specified QCD. Note that index recommendations also include relevant CREATE INDEX and DROP INDEX statements plus the associated COLLECT STATISTICS statements for any recommended indexes.
Get index recommendations	When this procedure is performed on a production system, this step is identical to the previous step.

Index Validation

Introduction

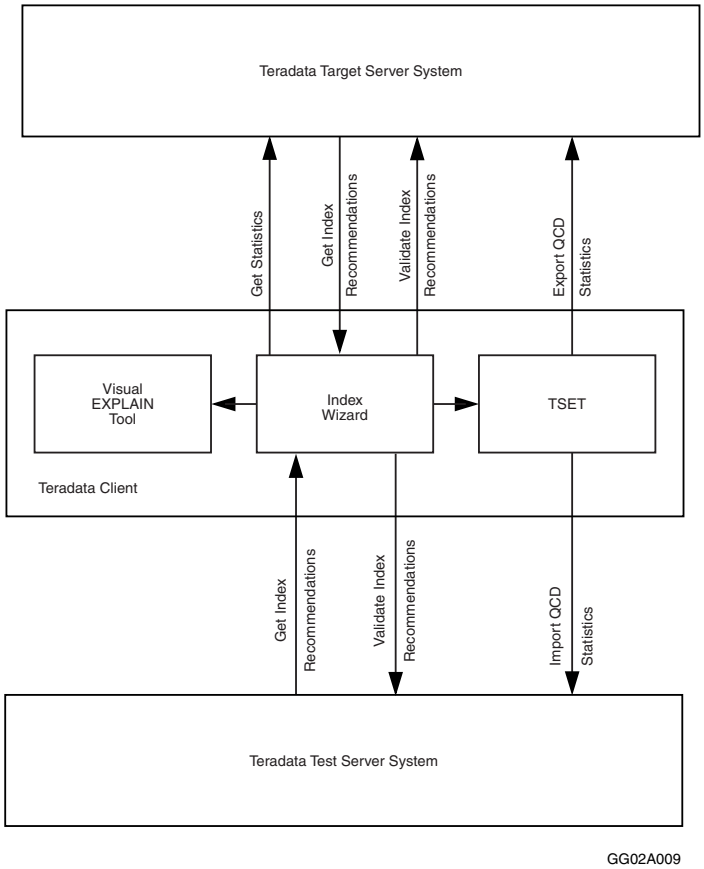
You can validate the indexes recommended by the Index Wizard by comparing query response latencies with the same workload against the existing index set against response latencies for the same query set and workload using the recommended index set. This is an optional, but recommended, step in the database query analysis process.

You can perform index validation either on a production system or on a test system, as indicated by the graphic.

See “DIAGNOSTIC ‘Validate Index’” in the chapter “Query and Workload Analysis Statements” in *Teradata RDBMS SQL Reference, Volume 4* for additional information about validating indexes recommended by the Teradata Index Wizard.

Index Validation Process

The process of validating recommended indexes is indicated by the following graphic.



<table>
<tr><th colspan="2">Key</th></tr>
<tr><th>Activity</th><th>Description</th></tr>
<tr><td>Get index recommendations</td><td>The Index Wizard utility retrieves the index recommendations from the QCD.</td></tr>
<tr><td>Get statistics</td><td>The SQL COLLECT STATISTICS (QCD Form) statement gathers sampled statistics from the production system for use in the validation process.

See “COLLECT STATISTICS (QCD Form)” in the chapter “SQL Data Definition Language Statement Syntax” in Teradata RDBMS SQL Reference, Volume 4 for more information.</td></tr>
<tr><td>• Validate recommendations
• Import/export QCD statistics</td><td>The actions required to complete these activities depend on whether the analysis is performed on a production system or a test system.

<table>
<tr><th>FOR this type of system …</th><th>The following actions must be undertaken …</th></tr>
<tr><td>Production</td><td>Rerun the entire workload after the previous activity in Index Validation has been performed.

The recommendations are made available to the Optimizer for generating the query plan.</td></tr>
<tr><td>Test</td><td>in the following order:

1 Export the statistics from the previous stage of Index Validation from the production system using TSET.

2 Import the exported statistics from the production system to the test system using TSET.

3 Rerun the entire workload.

The recommendations are made available to the test system Optimizer for generating the query plan.</td></tr>
</table>
</td></tr>
<tr><td>Compare the query plans with and without implementing the recommended indexes</td><td>Use the client Visual EXPLAIN tool to compare the query plans with and without incorporating the index recommendations to evaluate the extent of query processing enhancement produced by the recommendations.</td></tr>
</table>

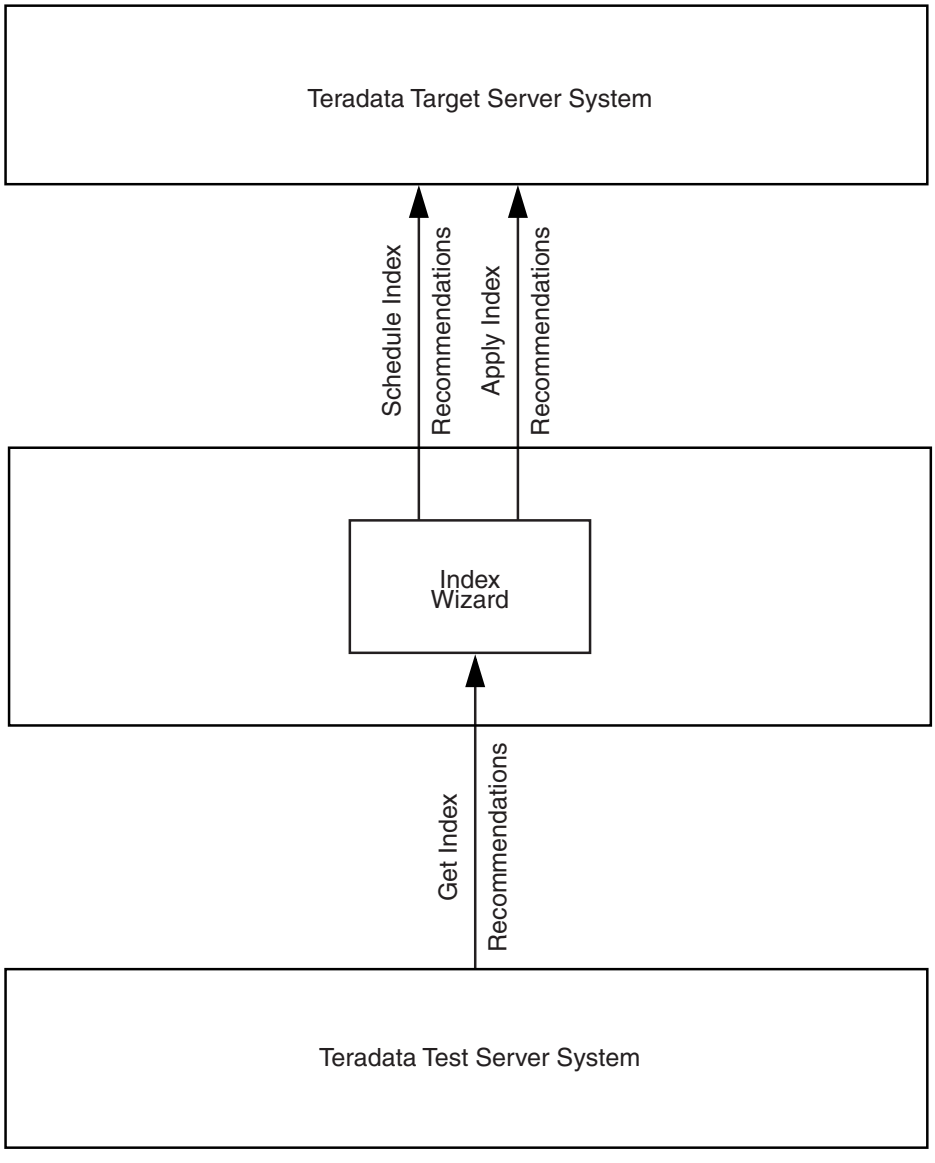
Index Application

Introduction

At this stage of the database query analysis, you apply the indexes recommended by the Index Analysis stage to the production database.

Index Creation Process

The process of applying the recommended indexes to the production system is indicated by the following graphic.



Key	
Activity	**Description**
Get index recommendations	Use the Teradata Index Wizard tool to retrieve index recommendations from the QCD.
Schedule application of index recommendations	Use the Index Wizard tool to schedule the application of the index recommendations.
Index recommendations	Perform the generated DDL statements to create the recommended index set on the production system. You can do this directly using the Teradata Index Wizard tool or you can schedule it for later using the TDQM API interface.

Chapter 6:

Target Level Emulation

This chapter describes Target Level Emulation, a set of tools used to set up a test system with environmental cost parameters and random AMP sample statistics from a production system having a different configuration. Then when a query is submitted on the test system, its Optimizer generates a query plan[1] identical to the plan that would have been produced on the production system. If you do not export the environmental costs and table statistics from the production system to your test system, then any EXPLAIN report produced for the same query is likely to differ on the two systems.

You can use the tool to validate and verify new queries in a test environment, ensuring that your production work is not disrupted by problematic queries.

The following general topics are described in this chapter:

- An overview of Target Level Emulation.
- Two basic procedures for enabling target level emulation and for capturing production system cost variables and applying them to a test system.

For additional information about environmental cost parameters, see "Environmental Cost Factors" on page 2-62.

For additional information about random AMP sampling, see "Random AMP Sampling" on page 2-35.

The SQL diagnostic statements that support Target Level Emulation are documented in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.

1 See the footnote to "What Is a Query Optimizer?" on page 2-3 for a definition of query plan.

An Overview of Target Level Emulation

Introduction

The Target Level Emulation facility permits you to emulate a target (production environment) system by capturing system-level environmental cost information and table-level random AMP sample statistics from that environment and storing it in the relational tables SystemFE.Opt_Cost_Table and SystemFE.Opt_RAS.Table.

You can then use the information from these tables together with appropriate column and index statistics to make the Optimizer on the test system generate query plans as if it were operating in the target environment rather than the test environment.

This feature produces a query plan for the emulated target system: it does *not* emulate the performance of that system.

Benefits of Target Level Emulation

This feature offers the following benefits to a DBA:

- Models the impact of various environmental changes on SQL statement performance.
- Provides an environment for determining the source of various Optimizer-based production database query problems using environmental cost data and random AMP sample-based statistical data.

Two Forms of Target Level Emulation

There are two forms of target level emulation: cost-based and random AMP sample-based. The two forms are orthogonal, but are definitely *not* mutually exclusive: they are meant to be used together as a tool for the precise analysis of production system query plans (see footnote 2 in Chapter 2 for a definition of a query plan) generated on much smaller test systems.

Environmental cost parameters are constant across a system configuration because there is only one set of environmental cost values per system. Table statistics, on the other hand, are different for each base table and do not vary as a function of system configuration. For example, the cardinality of a given table is the same whether that table is on a two AMP system or a 100 AMP system; however, environmental costs vary considerably as a function of system configuration.

These differences account for the sometimes dissimilar syntax of analogous SQL diagnostic statements for handling cost and random AMP statistical information.

Cost-based target level emulation has two possible forms: static and dynamic.

- Static emulation refers to setting environmental cost parameters at the SYSTEM level. Once set, the cost parameters persist across system restarts and reboots.
- Dynamic emulation refers to setting environmental cost parameters at the IFP, REQUEST, or SESSION levels (see "DIAGNOSTIC SET COSTS" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4* for definitions of what these terms mean in the context of Target Level Emulation).

  When you first set cost parameters, they are read dynamically from SystemFE.Opt_Cost_Table and copied to a memory-resident data structure. The Optimizer then initializes its cost parameters from this memory-resident structure on the fly. The cost parameters do *not* persist across system restarts and reboots.

  The advantage of dynamic target level emulation is that multiple users in multiple sessions can simultaneously emulate environmental cost parameters from multiple target systems on the same test system.

Random AMP sample-based target level emulation permits you to generate random AMP statistical samples on a production system and then export the captured data to a test system for detailed query analysis.

Related Systems

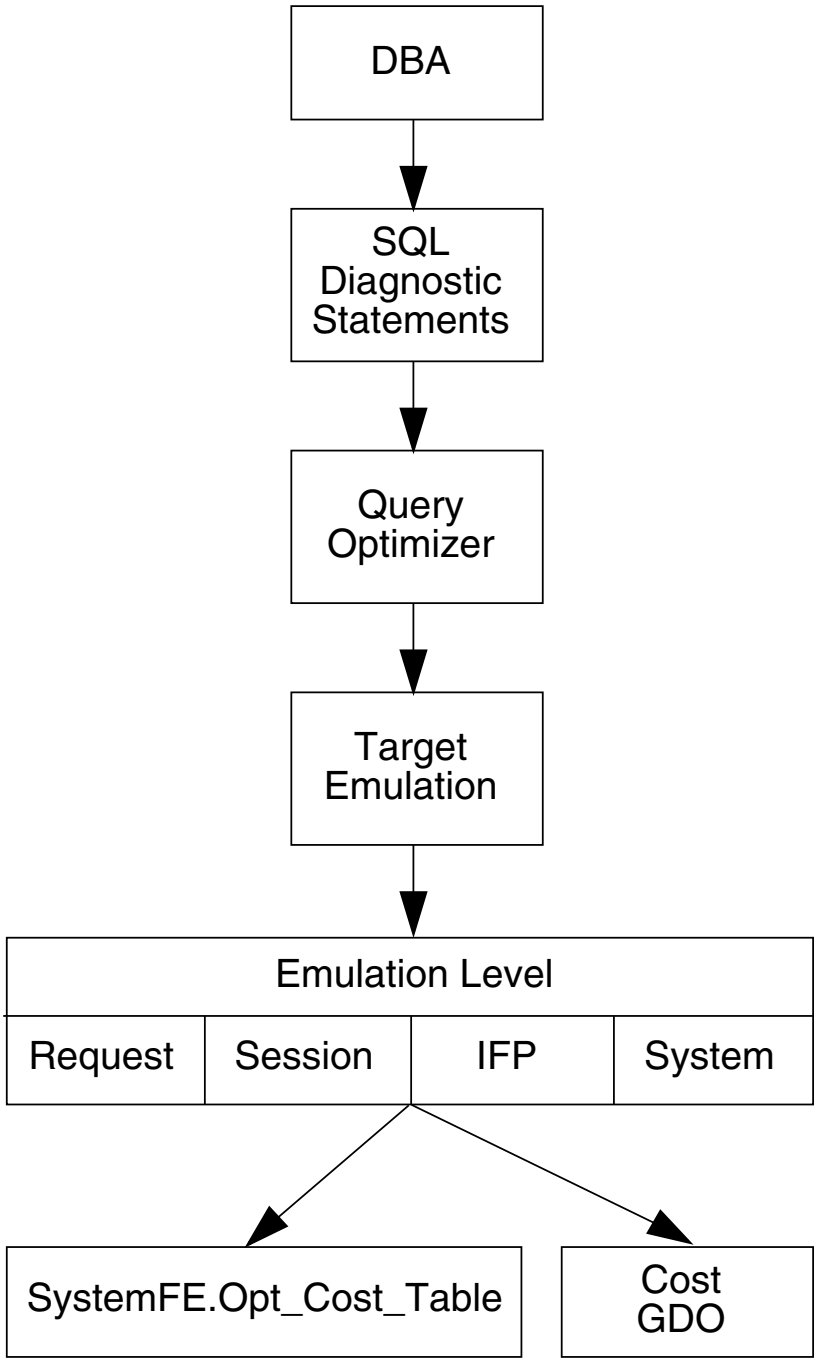
The target level emulation feature is closely related to other Teradata support tools, particularly those detailed in the following list:

- Query Capture Facility
- Visual Explain Tool
- Teradata System Emulation Tool (TSET)
- Teradata Index Wizard
- Teradata Statistics Wizard
- Statistics Collection tool (Teradata Manager)

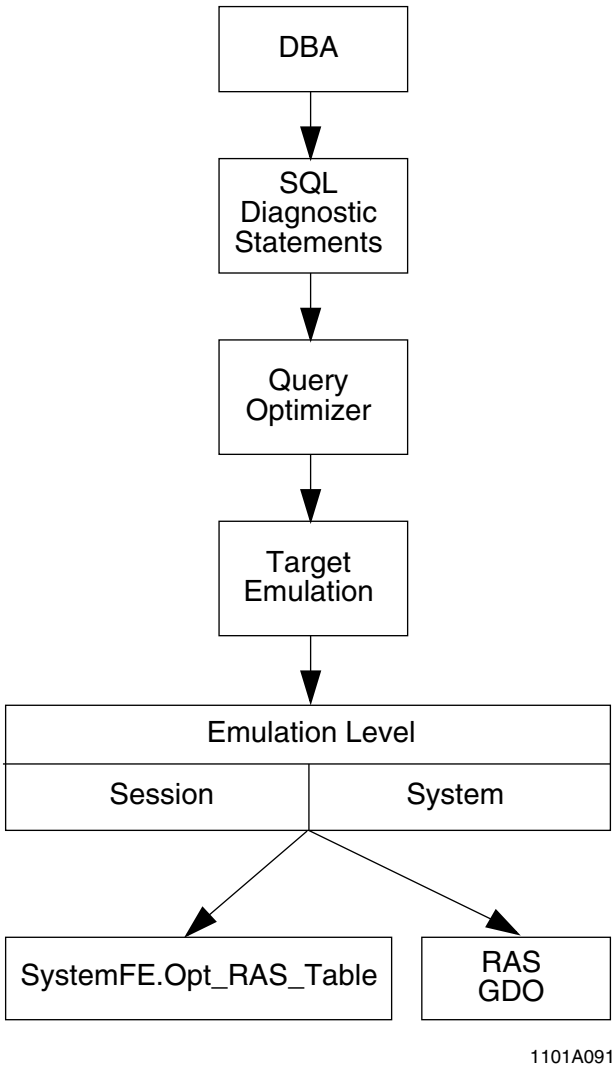
Benefits of Target Level Emulation

The principal benefit of target level emulation in the end user environment is that it ensures that queries are thoroughly debugged and optimized in a safe, but equivalent, emulated environment prior to introducing them into the production environment.

The following graphic indicates the work flow for environmental cost data capture and test.



The following graphic indicates the work flow for random AMP sample data capture and test:



Procedure to Enable Target Level Emulation

Introduction

Target Level Emulation is disabled by default and should *never* be enabled on a production system.

Before you can use target level emulation on a test system, you must enable it.

Procedure

The following table provides a high level procedure for performing this task:

Step	Action
1	Run the DIP utility script DIPSYSFE to create the optimizer cost values table SystemFE.Opt_Cost_Table and the random AMP sample statistical values table SystemFE.Opt_RAS_Table.
2	Use the DBSControl utility to enable Target Level Emulation on your test system by setting DBSControl general field 22, Target Level Emulation, to TRUE.

Procedures to Capture Target System Optimizer Cost Values and Random AMP Statistical Samples for a Test Environment

Introduction

The extraction of information from a target system and its subsequent application to a test system is a two-phase process, as indicated by the following table:

Stage	Process
1	Data capture phase. Target system optimizer variables are read from the Dictionary and various system tables and written to a staging area. This phase is initiated from the target system.
2	Application phase. The captured data in the staging area is exported to the appropriate Dictionary and system tables on the test system. This phase can be initiated from either the target or the test system.

You can use the Teradata System Emulation Tool (TSET) from a Windows client to perform this data capture and application.

If you do not use TSET, then you must follow a more circuitous path to capture target system optimizer variables and apply them to a test system. This topic presents both procedures.

Procedure Using TSET

Capturing data from a target production system and applying it to a test system is a two-phase process.

- The first phase reads the relevant data on the target system and then writes it to a staging area.
- The second phase exports the staging data to the test system and applies it to the appropriate data tables and system files.

The following table presents a high level overview of this process.

Step	Action
1	Depending on your requirements, perform one or both of the following steps: **Step** / **Action** **1** — Populate SystemFE.Opt_Cost_Table on the target system by running the DIAGNOSTIC DUMP COSTS statement. This operation extracts the Optimizer environmental cost value information from the target system and writes it to a relational table so it can be copied and used to drive the Optimizer on a test system. **2** — Populate SystemFE.Opt_RAS_Table on the target system by running the DIAGNOSTIC DUMP SAMPLES statement. This operation extracts random AMP statistics from the header of the specified table on the target system and writes it to a relational table so it can be copied and used to drive the Optimizer on a test system.
2	Log onto TSET from a client connected with the target system.
3	Create a target system profile to define the databases, tables, views, macros, and related environmental parameters or random AMP statistics to be captured for loading onto the test system.
4	Collect the Optimizer cost value information or random AMP statistics and any necessary object definitions and test queries from the target system and place it in a staging area from which the test system can import it. To do this, invoke the TSET Export function from the target system. This action writes the data created by DIAGNOSTIC DUMP COSTS or DIAGNOSTIC DUMP SAMPLES into a staging area.
5	Invoke the TSET Import function from a client connected with the test system to import the data from the staging area on the target system to SystemFE.Opt_Cost_Table or SystemFE.Opt_RAS_Table on the test system. The Import function permits you to select from several different load utilities to use to transfer the data.
6	If your test suite requires, manipulate the captured statistics using the Teradata Manager tool STATs.

Step	Action
7	Depending on your requirements, perform one or both of the following steps:

Step	Action
1	Run the DIAGNOSTIC SET COSTS SQL statement to specify the exact level of cost value information (and for which target configuration) you want to set. See "Mapping Target System Files to a Test System Optimizer Table and GDO" on page 6-12 for an illustration.
2	Run the DIAGNOSTIC SET SAMPLES SQL statement to specify the exact level of random AMP statistical sampling (and for which target configuration) you want to set. See "Mapping Target System Files to a Test System Optimizer Table and GDO" on page 6-12 for a comparable illustration.

Note that you can also use TSET to copy any of the following target system data to an appropriate test system.

- Object definitions
- Statistics
- QCD data
- Random AMP samples
- All available target system data, including cost parameters.

Procedure Not Using TSET

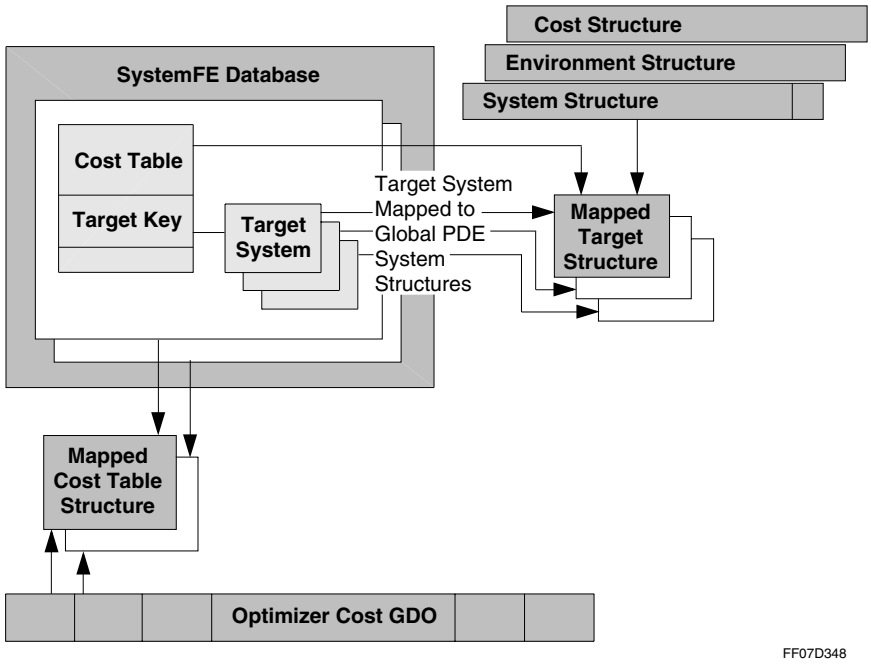
The following table provides a high level procedure for capturing Optimizer environmental cost information from a production system and applying that information to a test system.

Step	Action
1	Depending on your requirements, perform one or both of the following steps: **Step** / **Action** **1** — Populate SystemFE.Opt_Cost_Table on the target system by running the DIAGNOSTIC DUMP COSTS statement. This extracts the Optimizer environmental cost value information from the target system and writes it to a relational table so it can be copied and used to drive the Optimizer on a test system. **2** — Populate SystemFE.Opt_RAS_Table on the target system by running the DIAGNOSTIC DUMP SAMPLES statement. This extracts the random AMP statistical samples from the target system and writes them to a relational table so they can be copied and used to drive the Optimizer on a test system.
2	Depending on your requirements, perform one or both of the following steps: **Step** / **Action** **1** — Copy the Optimizer cost value information and any necessary object definitions and test queries from the target system to the test system. To do this, select rows from SystemFE.Opt_Cost_Table on the production system and export them to a file using an appropriate utility from the following list. • FastExport • CLIv2 • BTEQ SQL script **2** — Copy the random AMP statistical samples and any necessary object definitions and test queries from the target system to the test system. To do this, select rows from SystemFE.Opt_RAS_Table on the production system and export them to a file using an appropriate utility from the following list. • FastExport • CLIv2 • BTEQ SQL script

Step	Action
3	Depending on your requirements, perform one or both of the following steps: **Step** / **Action** 1 — Import the rows into SystemFE.Opt_Cost_Table on the test system using an appropriate utility from the following list. • FastLoad • MultiLoad • TPump • CLIv2 • BTEQ SQL script 2 — Import the rows into SystemFE.Opt_RAS_Table on the test system using an appropriate utility from the following list. • FastLoad • MultiLoad • TPump • CLIv2 • BTEQ SQL script
4	Populate the test system with appropriate test data imported from the production system for which testing is to be performed.
5	If your test suite requires, manipulate the captured statistics using the Teradata Manager tool STATs.
6	Depending on your requirements, perform one or both of the following steps: **Step** / **Action** 1 — Run the DIAGNOSTIC SET COSTS SQL statement to specify the exact level of cost value information (and for which target configuration) you want to set. See "Mapping Target System Files to a Test System Optimizer Table and GDO" on page 6-12 for an illustration. 2 — Run the DIAGNOSTIC SET SAMPLES SQL statement to specify the exact level of random AMP statistical sampling (and for which target configuration) you want to set. See "Mapping Target System Files to a Test System Optimizer Table and GDO" on page 6-12 for a comparable illustration.

Mapping Target System Files to a Test System Optimizer Table and GDO

The following graphic indicates the flow of how target system optimizer environmental cost parameters are mapped to test system structures that, in turn, cause the test system query optimizer to generate query plans that are based on the environmental cost parameters and data demographics of the emulated production system.



The flow for random AMP statistical samples is essentially identical after making the necessary changes to the node labels in the diagram.

Chapter 7:

Transaction Processing

This chapter discusses how Teradata processes transactions.

Topics include:

- "Database Transactions" on page 7-2
- "Locking Levels and Severities" on page 7-3
- "Lock Manager" on page 7-6
- "Default Lock Assignments and Lock Upgradeability" on page 7-8
- "DDL and DCL Statements, Data Dictionary Access, and Locks" on page 7-17
- "DML Statements and Locks" on page 7-18
- "Cursor Locking Modes" on page 7-19
- "Blocked Requests" on page 7-21
- "Deadlock Detection and Resolution" on page 7-24
- "Preventing Deadlocks" on page 7-26
- "Locking and Transaction Processing" on page 7-32
- "Upgrading System Locks" on page 7-33
- "Transaction Semantics: Operating in ANSI or Teradata Transaction Modes" on page 7-34
- "Rollback Processing" on page 7-41
- "Locking Issues With Tactical Queries" on page 7-43

Database Transactions

Definition: Transaction

A transaction is both a unit of work and a unit of recovery.

- As a unit of work, a transaction defines a limited set of SQL operations to be performed on a database. Either all of these operations must be performed or none of them can be performed. Depending on the situation, the limits can be defined either implicitly or explicitly.
- As a unit of recovery, a transaction defines a limited set of rollbacks to be performed on a database in order to change its current consistent state back to an earlier consistent state.

The general concept of transaction processing is encapsulated by their so-called ACID properties. ACID is an acronym for the following set of properties that characterize any database transaction:

- Atomicity
- Consistency
- Isolation
- Durability

The specific meanings of these expressions in terms of database transactions are defined in the following table:

Term	Definition
Atomicity	A transaction either occurs or it does not. No matter how many component SQL operations are specified within the boundaries of a transaction, they must all complete successfully and commit or they must all fail and rollback. There are no partial transactions.
Consistency	A transaction transforms one consistent database state into another. Intermediate inconsistencies in the database are not permitted.
Isolation	The operations of any transaction are concealed from all other transactions until that transaction commits.
Durability	Once a commit has been made, the new consistent state of the database survives even if the underlying system crashes.

Locking Levels and Severities

Introduction

Depending on the request being processed, Teradata places a certain default lock level on the object of the request, which can be one of the following database objects:

- Database
- Table
- View
- Row hash

The locking level determines whether other users can access the target object.

Locking severities and locking levels (see "Lock Manager" on page 7-6) combine to exert various locking severities. The more restrictive the combination, the greater the impact on system performance and the greater the delay in processing time.

Locking Severity and Relative Restrictiveness

The available lock severities, from most restrictive to least restrictive, are as follows.

Lock Severity	Description
Most Restrictive	
EXCLUSIVE	Placed only on a database or table when the object is undergoing structural changes (for example, a column is being created or dropped) or when a database is being restored by a host utility. An EXCLUSIVE lock restricts access to the object by any other user. You can also place this lock explicitly using the LOCK modifier.
WRITE	Placed in response to an INSERT, UPDATE, or DELETE request. A WRITE lock restricts access by other users (except for readers who are not concerned with data consistency and choose to override the automatically applied lock by specifying a less restrictive ACCESS lock). You can also place this lock explicitly using the LOCK modifier.
READ	Placed in response to a SELECT request. A READ lock restricts access by users who require EXCLUSIVE or WRITE locks. You can also place this lock explicitly using the LOCK modifier.

Lock Severity	Description
CHECKSUM	Placed in response to a user-defined LOCK FOR CHECKSUM when using cursors in embedded SQL. CHECKSUM locking is identical to ACCESS locking except that it adds checksums to the rows of a spool file to allow a test of whether a row in the cursor has been modified by another user or session at the time an update is being made through the cursor. See also "Cursor Locking Modes" on page 7-19 and "LOCKING Modifier" in the chapter "SQL Data Manipulation Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 6*.
ACCESS	Placed in response to a user-defined LOCK FOR ACCESS phrase. Permits the user to access for READ an object that may be already locked for READ or WRITE An ACCESS lock does not restrict access by another user except when an EXCLUSIVE lock is required. A user requesting access disregards all data consistency issues. Because ACCESS and WRITE locks are compatible, the data might be undergoing updates while the user who requested the access is reading it. Therefore, any query that requests an ACCESS lock might return incorrect or inconsistent results
Least Restrictive	

Compatibility Among Locking Severities

The following notation is used in this description of locking severity compatibilities:

Notation	Definition
Q	A locking queue associated with a database object.
L	A list of locks currently held.
`lock(<object>,<lock requested>`	A database object-locking severity requested pair

Each database object has an associated locking queue *Q* and list of currently held locks *L*. All transactions perform a locking operation before they access any database objects.

IF lock(<object>,<lock requested>) is …	THEN …
queued for any lock in *L*	the transaction is placed in *Q* and waits there as long as `lock(<object>,<lock requested>)` is queued.
granted	`lock(<object>,<lock requested>)` is added to *L* with `<lock requested>` and the transaction resumes processing

After the transaction finishes with an object, its lock is removed from *L*.

The following table summarizes the action taken when a requested locking severity competes with an existing locking severity:

Requested Lock Severity	Held Lock Severity				
	None	ACCESS	READ	WRITE	EXCLUSIVE
ACCESS or CHECKSUM	Lock Granted	Lock Granted	Lock Granted	Lock Granted	Request Queued*
READ	Lock Granted	Lock Granted	Lock Granted	Request Queued*	Request Queued*
WRITE	Lock Granted	Lock Granted	Request Queued*	Request Queued*	Request Queued*
EXCLUSIVE	Lock Granted	Request Queued*	Request Queued*	Request Queued*	Request Queued*

* If you specify NOWAIT, the transaction aborts instead of queueing.

A WRITE or EXCLUSIVE lock on a database, table, or view restricts all requests from accessing data except the one holding the lock.

Because a lock on an entire database can restrict access to a large quantity of data, the Lock Manager ensures that default database locks are applied at the lowest possible level and severity required to secure the integrity of the database while simultaneously maximizing concurrency.

Full-table WRITE locks on dictionary tables prevent contending tasks from using the dictionary, so the Lock Manager attempts to lock dictionary tables at the row hash level whenever possible.

NOWAIT

When you specify the NOWAIT option, a lock request that cannot be fulfilled immediately aborts the transaction. For details on how to use NOWAIT, see "LOCKING Modifier" in the chapter "SQL Data Manipulation Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 6.*

Lock Manager

Introduction

Any number of users and applications can simultaneously access data stored in a Teradata database.

The Teradata Lock Manager imposes concurrency control by locking the data being accessed by each transaction and unlocking the data when the transaction completes. This control ensures that the data remains consistent for all users.

While the Lock Manager imposes locks automatically, you can upgrade locks explicitly by using the SQL LOCKING modifier (see "LOCKING Modifier" in the chapter "SQL Data Manipulation Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 6*).

Locking Levels

The Lock Manager implicitly locks an object at the following levels:

This lock level ...	Locks ...
Most Restrictive	
Database	all rows of all tables in the database and their associated secondary index subtables.
Table	all rows in the base table and in any secondary index and fallback subtables associated with it.
View	all underlying tables accessed by the view.
Row Hash	the primary copy of rows sharing the same row hash value. A row hash lock permits other users to access other data in the table and is the least restrictive type of lock. A row hash lock applies to a *set* of rows that shares the same hash code. It does not necessarily, nor even generally, lock only one row. • A row hash lock is applied whenever a table is accessed using a unique primary index (UPI). • For an update that uses a unique secondary index (USI), the appropriate row of the secondary index subtable is also locked. • It is not necessary to lock the fallback copy of the row, nor any associated row, of a nonunique secondary index (NUSI) subtable.
Least Restrictive	

Lock Considerations

Teradata always makes an effort to lock database objects at the least restrictive level and severity possible to ensure database integrity while maintaining maximum concurrency. When determining whether to grant a lock, the Lock Manager takes into consideration both the requested locking severity and the object to be locked.

For example, a READ lock requested at the table level cannot be granted if a WRITE or EXCLUSIVE lock already exists on any of the following database objects:

- The database that owns the table
- The table itself
- Any rows in the table

A WRITE lock requested at the row hash level cannot be granted if a READ, WRITE, or EXCLUSIVE lock already exists on any of the following database objects:

- The owner database for the table
- The parent table for the row
- The row hash itself

In each case, the request is queued until the conflicting lock is released.

It is possible to exhaust Lock Manager resources. Any transaction that requests a lock when Lock Manager resources are exhausted aborts. In such cases, row hash locking for DDL statements can be disabled.

You can review all the active locks and determine which other user locks are blocking your transactions if you have the performance monitor MONITOR SESSION privilege and an application that uses the Performance Monitor/Application Programming Interface.

Default Lock Assignments and Lock Upgradeability

Introduction

The Lock Manager assigns locks to SQL statements by default. Depending on the assigned lock and the individual SQL statement, you can also upgrade or downgrade the default locking assignments using the LOCKING modifier (see "LOCKING Modifier" in the chapter "SQL Data Manipulation Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 6*).

Default Lock Assignments

The following table indicates some of the default lock assignments the Teradata Lock Manager applies:

SQL Statement	Access Type		Default Lock Type Assigned
	UPI or USI	NUSI or FTS	
SELECT	Row hash	Table	READ SHARE
INSERT	Row hash	Not applicable	WRITE
UPDATE	Row hash	Table	WRITE
MERGE	Row hash	• Not applicable for INSERT • Table for UPDATE	WRITE
DELETE	Row hash	Table	WRITE
CREATE TABLE DROP TABLE ALTER TABLE	Not applicable	Table	EXCLUSIVE
CREATE DATABASE DROP DATABASE MODIFY DATABASE	Not applicable	Database	EXCLUSIVE

You can change some default lock assignments using the LOCKING modifier (see "LOCKING Modifier" in the chapter "SQL Data Manipulation Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 6* for syntax and usage details). You can upgrade any lower severity lock to an EXCLUSIVE lock, but the only downgrade permitted is from a READ lock to an ACCESS lock.

You can change the status of user-specified locks redundantly, upgrade them, or downgrade them according to the following specification:

A change from this default assignment lock ...	TO this user-specified lock ...	IS ...
ACCESS[a]	ACCESS	redundant, but valid.
READ[b] SHARE	READ SHARE	
WRITE	WRITE	
EXCLUSIVE	EXCLUSIVE	
ACCESS	READ	a valid upgrade.
	WRITE	
	EXCLUSIVE	
READ SHARE	WRITE	
	EXCLUSIVE	
WRITE	EXCLUSIVE	
READ	ACCESS	a valid downgrade.

a. Even though a CHECKSUM lock is essentially identical to an ACCESS lock, you can only specify CHECKSUM locks using the LOCKING modifier; you cannot upgrade them to a higher severity lock. This is because Teradata does not specify CHECKSUM as a default lock severity.
b. READ and SHARE are synonyms.

The following table combines the information from the two previous tables to indicate associations between individual SQL DML statements and lock upgrades and downgrades at the row hash, view, and table database object levels:

This LOCKING modifier severity specification ...	Is available for this SQL DML statement ...
EXCLUSIVE	DELETE INSERT MERGE SELECT UPDATE
WRITE	SELECT
READ	SELECT
ACCESS	SELECT

The reason you can only specify the LOCKING FOR EXCLUSIVE modifier for the DELETE, INSERT, MERGE, and UPDATE statements is that the default lock severity for these statements is WRITE. You cannot downgrade a WRITE lock for these statements because doing so would compromise the integrity of the database. Because the SELECT statement does not change data, and therefore its actions cannot compromise database integrity, you are permitted to change its default locking severity to any other severity.

Releasing Locks

Except for locks placed during Host Utility (HUT) processing, users have no control over when locks are released.

The Lock Manager releases locks upon completion of any of the following:

- Implicit or two-phase commit transaction
- Outermost END TRANSACTION statement of an explicit transaction
- COMMIT or ROLLBACK of an ANSI mode transaction.

This occurs regardless of when you receive the data because the spool file may exist beyond the end of the transaction.

Guidelines for Maximizing Concurrency

The following set of guidelines set out some rules of thumb for maximizing concurrency with your database transactions.

- A SELECT statement that requests a READ lock against a table cannot run concurrently with an executing CREATE INDEX or ALTER TABLE … FALLBACK statement for the same table.

  Instead, specify a READ lock for the CREATE INDEX or ALTER TABLE statement to permit concurrency.
- If the CREATE INDEX or ALTER TABLE … FALLBACK LOCK modifier specifies WRITE (or if there is no LOCK modifier), then specify an ACCESS lock in your SELECT statement to permit concurrency.

  Note that the ALTER TABLE operation can be to add FALLBACK only; if other table attributes are added, then ALTER TABLE cannot run concurrently with SELECT.
- Because requests for WRITE locks can result in transactions being blocked and can result in deadlocks, you should consider running only read-only transactions (or access-only if a LOCK FOR WRITE clause is specified) concurrently with ALTER TABLE … FALLBACK or CREATE INDEX statements.

- Be aware that when running long transactions concurrently with CREATE INDEX or ALTER TABLE … FALLBACK, the CREATE INDEX or ALTER TABLE statement might not complete until the long running transactions have completed.
- To avoid the chance of deadlocks with other DML transactions or DDL statements, consider writing your DML transactions to immediately obtain the highest severity lock they will require rather than attempt to upgrade a less severe lock at a later time during transaction processing.

Example: No Deadlock

Introduction

The following example demonstrates how to optimize concurrency using the guidelines from "Releasing Locks" on page 7-10.

Table Definition

Assume the following table definition.

```
CREATE TABLE table_1 (
 column_1 INTEGER,
 column_2 INTEGER)
PRIMARY INDEX (column_1);
```

Problem Transactions

Now consider the following simultaneously running transactions.

Transaction Number	SQL Text
1	`LOCK table_1 FOR READ` `ALTER TABLE table_1, FALLBACK;`
2	`SELECT *` `FROM table_1;`

Transaction Processing

Assume these steps are taken in the following order:

Step	Process
1	The first transaction places a table-level READ lock on table_1.
2	The second transaction also places an table-level READ lock on table _1.
3	Both transactions access table_1 at the same time and run concurrently.
4	The first transaction builds the fallback concurrently with the SELECT in the second transaction, but does not complete until after the second transaction completes and releases its lock on table_1.

Example: Deadlock

Introduction

The following is an example of a deadlock situation.

Table Definition

Consider the following table definition.

```
CREATE TABLE table_1
 (
  column_1 INTEGER,
  column_2 INTEGER,
  column_3 INTEGER,
  column_4 INTEGER
 )
PRIMARY INDEX (column_1);
```

The following two transactions are running simultaneously.

Transaction Number	Request Number	SQL Text
1	1	LOCK table_1 FOR READ CREATE INDEX (column_3, column_4) ON table_1;
2	2	BT;
	3	SELECT * FROM table_1;
	4	UPDATE table_1 SET column_1 = <value-1> WHERE column_1 = <value-2>;
	5	ET;

Problem Transactions

Assume these stages are taken in the following order:

Stage	Process
1	Transaction 1 places a READ lock on table_1. This lock is in effect until the table header needs to change (after the creation of the index subtables is complete).
2	Request 3 places an READ lock on table_1.
3	Transaction 1 and request 3 can run concurrently when granted access to table_1.
4	Request 3 finishes but does not release the READ lock on table_1 until the end of the transaction.
5	Request 4 attempts to place a WRITE row hash lock on table_1. Its lock request is blocked because of the READ lock placed on table_1 by transaction 1.
6	Transaction 1 needs to upgrade its lock from READ to EXCLUSIVE. Its lock request is blocked because of the WRITE row hash lock placed by request 4.
7	At this point there is a deadlock situation: request 4 is waiting for transaction 1 to release its lock, and transaction 1 is blocked by request 4.

Problem Resolution

To avoid this deadlock, change your SQL in either of the following ways:

- Add the modifier LOCK table_1 FOR WRITE to the second transaction after request 2.

```
BT;
LOCK table_1 FOR WRITE
SELECT *
FROM table_1;
UPDATE table_1
SET column_1 = value_1
WHERE column_1 = value_2;
ET;
```

- Remove the LOCK modifier from the first transaction.

```
CREATE INDEX (column_3, column_4) ON table_1;
```

Example: Two Serial Transactions

Introduction

The following example shows two transactions, with the first transaction starting before the second transaction.

Table Definition

Consider the following table definition.

```
CREATE TABLE table_1
 (
  column_1 INTEGER,
  column_2 INTEGER,
  column_3 INTEGER,
  column_4 INTEGER
 )
PRIMARY INDEX (column_1);
```

Problem Transactions

The following two transactions are running simultaneously, with the first having started prior to the second.

Transaction Number	SQL Text
1	LOCK table_1 FOR READ CREATE INDEX (column_3, column_4) ON table_1;
2	SELECT * FROM table_1 WHERE column_3 = 124 AND column_4 = 93;

Each transaction places a table-level READ lock on table_1. The transactions obtain access to table_1 and run concurrently.

Note that the SELECT statement does not recognize the index being created by the CREATE INDEX statement.

Problem Resolution

To eliminate concurrency, change the coding of the first transaction to make its explicit lock exclusive.

Transaction Number	SQL Text
1	`LOCK table_1 FOR EXCLUSIVE` `CREATE INDEX (column_3, column_4) ON table_1;`

2	`SELECT *`
	`FROM table_1`
	`WHERE column_3 = 124`
	`AND column_4 = 93;`

The upgraded LOCK FOR EXCLUSIVE modifier in the first transaction blocks the table-level READ lock request on table_1 in the second transaction.

DDL and DCL Statements, Data Dictionary Access, and Locks

The performance of a DDL or DCL statement causes the Data Dictionary to be updated and appropriate locks to be placed on system tables while that statement is processing.

To improve concurrency, DDL and DCL processing use the finest granularity of locking that is practical and delay placing their locks for as long as possible.

DML Statements and Locks

When it processes DML statements, the Parser accesses the information it needs from the Data Dictionary tables using internal express-request transactions that place READ locks on row hash values. These locks are released when the data is returned to the parser.

Locks applied as a result of DML statement processing are listed in the following table:

DML Request	Updated Fields	Selection Criteria	Object Locked	Locking Severity
SELECT	Not applicable	UPI or USI[a]	Row hash	READ
		NUPI[b]	Set of rows	READ
		Any other[a]	Table	READ
DELETE[c]	Not applicable	UPI or USI	Row hash	WRITE
		NUPI	Set of rows	WRITE
		Any other	Table	WRITE
INSERT … SELECT	Not applicable	Select table		
		UPI or USI	Row hash	READ
		NUPI	Set of rows	READ
		Any other	Table	READ
			Insert table	WRITE
INSERT … [VALUES][c]	Not applicable	Not applicable	Primary row	WRITE
UPDATE[c]	Neither UPI nor USI	UPI or USI	Row hash	WRITE
	Neither NUPI nor USI	NUPI	Set of rows	WRITE
		Any other	Table	WRITE
	USI	USI	Table	WRITE
MERGE (Update)[d]	Neither UPI nor USI	UPI or USI	Row hash	WRITE
	Neither NUPI nor USI	NUPI	Set of rows	WRITE
		Any other	Table	WRITE
	USI	USI	Table	WRITE
MERGE (Insert) [c]	Not applicable	Not applicable	Primary row	WRITE

a. If a join or hash index includes the specified columns, then the entire index is READ-locked and the base table is not accessed.
b. If a join or hash index includes the specified columns, then a set of index rows is READ-locked and the base table is not accessed.
c. Inserts, updates, and deletes on a table with a join or hash index require WRITE locks on the index and READ locks on other tables associated with it if and only if the modified base table columns are also defined for the index.
d. MERGE exerts a "matching row hash lock" (a row hash lock based on the primary index value specified by the MATCH condition) whether performance of the MERGE results in an update, an insert, or neither.

Cursor Locking Modes

Introduction

Positioned cursors do not support specific locking levels; however, they expect the following rules to be observed.

- All actions involving a cursor must be done within a single transaction
- Terminating a transaction closes any open cursors

Locking Levels and Updatable Cursors

A SELECT statement performed when its associated cursor is opened causes Teradata to use either a table-level or row hash-level lock by default, depending on the constraint clause of the SELECT statement.

You can explicitly change this locking level by using the LOCKING modifier. The LOCKING modifier is supported for updatable cursors, and the CHECKSUM locking severity is designed especially for use with LOCKING.

Example

This example uses LOCK with CHECKSUM on table_1.

```
EXEC SQL
   REPLACE macro macro_2 AS
   (
   LOCK TABLE table_1 FOR CHECKSUM
   SELECT i, text
   FROM table_1
   )
;
```

How Cursors and Locks Interact

When a cursor is opened, Teradata generates a response pool table. This table identifies each data row that is a source for the response data in the spool table row.

The identifier so generated is used by Teradata to UPDATE or DELETE the data row when the application specifies such an action against WHERE CURRENT OF *cursor_name*.

IF the locking modifier is . . .	THEN Teradata . . .
ACCESS	does *not* check to ensure that the data row to be updated or deleted has not been modified since the response data was generated for the spool table. The target data row could be a completely new row if some other application has deleted the original source row and inserted a new row in its place.
CHECKSUM	inserts a checksum into each row of the spool file. This provides a mechanism for ensuring that the rows in the spool file have not been modified by another user or session at the time an update is being made through the cursor. Note that the CHECKSUM option does *not* guarantee that all conflicts will be detected. There is a small, but finite, possibility that a row created by update might have an identical checksum to the original, unmodified row. The CHECKSUM option uses ACCESS severity locks. CHECKSUM locking differs in that it also provides the checksums of the spool file rows.

Blocked Requests

Introduction

A request that is waiting in a lock queue is considered to be blocked. If you suspect that a request is blocked, you can use the Query Session utility to review the status of the session.

A compatibility with an EXCLUSIVE lock can result in a queue of several blocked requests, all of which must wait until the EXCLUSIVE lock is released.

Problem: Single Statement Transactions

When several requests that compete for the same table are submitted as separate, single statement transactions, the Lock Manager resolves their locking requirements as illustrated by the following table:

Stage	Process
1	Job_1 requires a READ lock on table_a. table_a is free, so the lock is granted and job_1 begins.
2	While job_1 is still running, job_2 requires a WRITE lock. This conflicts with the active READ lock, so the WRITE lock is denied and job_2 is queued.
3	Job_3 requires an ACCESS lock. ACCESS locks are compatible with both READ and WRITE locks (if job_1 completes, releasing the READ lock, then job_2 can begin whether or not job_3 still holds the ACCESS lock), so the ACCESS lock is granted and job_3 is allowed to run concurrently with job_1.
4	Job_4 requires a READ lock. This conflicts with the queued WRITE lock, so job_4 is queued behind job_2.
5	Job_5 requires an EXCLUSIVE lock. An EXCLUSIVE lock conflicts with all other locks, so job_5 is queued behind job_4.
6	Job_6 requires an ACCESS lock. This conflicts with the queued EXCLUSIVE lock, so job_6 is queued behind job_5.

Problem Resolution

Careful session scheduling can prevent such an endless queue of blocked requests.

For example, a request that needs an EXCLUSIVE lock can be submitted first or last, depending on how long it takes to process and its function in relation to other requests. It should be run first if other requests depend on its changes.

If a request that needs an EXCLUSIVE lock takes a lot of processing time, consider submitting it as a batch job to run during off hours.

Multistatement Transactions

Explicit multistatement transactions should also be reviewed for any scheduling concerns.

When competing locks are needed by multiple requests in a single transaction, the Lock Manager automatically upgrades the mode for each request, in turn, until the transaction is completed.

This handling protects an active transaction from being interrupted by new arrivals. However, a blocked queue can still result if the active transaction has many requests or demands excessive processing time.

For example, consider the following scenario:

Stage	Process			
	This transaction number …	Has this request …	That requires this lock …	With this result …
1	1	SELECT … FROM table_a	READ	the READ lock on table_a is granted and the SELECT request begins processing.
		INSERT INTO table_a …	WRITE	the WRITE lock on table_a is not granted and the INSERT into table_a request is blocked pending completion of the SELECT request.
2	2	INSERT INTO table_a …	WRITE	transaction 2 is queued because its request for a WRITE lock on table_a is not compatible with the READ lock already in place on table_a.

Stage	Process			
	This transaction number …	Has this request …	That requires this lock …	With this result …
3	1	SELECT … FROM table_a	None The READ lock is released.	processing of the SELECT request from Transaction 1 completes.
4		INSERT INTO table_a …	WRITE	the Lock Manager upgrades the lock on table_a for the Transaction 1 INSERT request
5	2	INSERT INTO table_a …	WRITE	transaction 2 remains queued and inactive, waiting for its WRITE lock request on table_a to be granted.

Deadlock Detection and Resolution

Introduction

When two or more transactions are competing for locks on the same data, a deadlock can occur in which none of the deadlocked transactions are able to proceed. This is sometimes referred to as a deadly embrace.

For example, a local deadlock occurs when two transactions that concurrently hold READ locks for the same data attempt to enter an update request.

Also, because requests are processing in parallel on multiple AMPs, it is possible for a global deadlock to occur.

For example, suppose transaction_a places a WRITE lock on object_x (for example the rows sharing the same hash value of a table) on AMP_1, and transaction_b places a WRITE lock on object_y on AMP_2. Both of these lock requests are granted.

Now, if transaction B attempts to place a write lock on object_x on AMP_1, it is blocked. In a similar manner, if transaction_a places a WRITE lock on object_y on AMP_2, it is also blocked. Neither transaction can complete.

Application Considerations

Applications must be written to handle deadlocks and deadlock aborts. When developing applications for relational database management systems, you must keep in mind that deadlocks are unavoidable. You should *expect* deadlocks to occur.

Because of this, you need to design your applications to anticipate and avoid deadlocks while at the same time taking care to plan for graceful recovery whenever a deadlock (or a system-initiated abort as the result of a deadlock) occurs.

A retry is usually all that is required. For example, BTEQ always retries by default.

If a multistatement transaction is involved, the entire transaction must be executed again, because Teradata rolls back the results of all processing performed up to the time of the abort. This behavior does not occur in ANSI mode because ANSI does not permit multistatement transactions.

Retrievals With Access Mode

Because an ACCESS lock is compatible with all locking modes except EXCLUSIVE, a user requesting an ACCESS lock might be allowed to read an object on which a WRITE lock is being held. This means that data could be retrieved by an ACCESS user while that same data is simultaneously being modified by a WRITE user. Therefore, any query that places an ACCESS lock can return incorrect or inconsistent results.

For example, assume a SELECT that uses a secondary index constraint is submitted with a LOCK FOR ACCESS phrase.

If the ACCESS lock is granted on an object being held for WRITE, the field value could change between the time the secondary index subtable key is located and the time the data row is retrieved (such a change is possible because a satisfied SELECT index constraint is not always double-checked against the base data row). This type of inconsistency may occur even if the data is changed only momentarily by a transaction that is later backed out.

Preventing Deadlocks

Introduction

For particular types of transactions, or for very large or urgent applications, you can reduce or prevent the chance of a deadlock by including the LOCK modifier in your SQL statements.

How the LOCK Modifier Can Help to Prevent Deadlocks

The LOCK modifier can be used to improve performance and reduce conflicts in the following ways:

- Using the NOWAIT option to abort a transaction if a lock cannot be granted immediately.
- Using the LOCK ROW FOR WRITE syntax to eliminate the chance of a deadlock during upgrading when multiple transactions select and then update the same row.
- Applying a higher or lower severity of lock than that normally applied by the Lock Manager.

Using the LOCK Modifier to Enhance Concurrency

To make more efficient use of system resources, you can use the LOCK modifier in a transaction to decrease or increase the restrictiveness of locks that would otherwise be placed by the system during the processing of transaction requests.

The following bullet points list some important facts about the LOCK modifier:

- The LOCK modifier is not ANSI SQL-99-compliant.
- The LOCK modifier can precede any Teradata SQL request or it can be used alone. It is typically used as a modifier, in which case it precedes the statement. For example, the following statement is valid:

```
LOCK TABLE tablename FOR READ;
```

- More than one LOCK modifier can appear in the same request.
- The CREATE VIEW and REPLACE VIEW statements allow the LOCK modifier to be specified as part of the view definition.

- You can use the LOCK modifier to specify the mode of lock to be placed on a database, table, or row hash before a request is processed. You can specify LOCK with the NOWAIT option to abort a transaction if a lock cannot be granted immediately.
- LOCK cannot prevent a lock of a higher mode from being imposed, and it does not affect objects that are already locked. You can precede your request with EXPLAIN to see the locks that will be set as each statement is executed[1].

The following example uses the LOCK phrase as both a modifier and as a statement:

```
LOCK TABLE table_a FOR READ
LOCK TABLE table_b FOR READ
SELECT …
FROM table_a, table_b
WHERE …;
LOCK TABLE tablename FOR WRITE
;
UPDATE …;
```

Dynamic Lock Upgrades and Deadlock

When the Optimizer specifies a primary or unique secondary index to process a SELECT statement, the Lock Manager applies a READ lock on the row hash value.

If the same transaction contains a subsequent DML statement based on the same index, the Lock Manager upgrades the READ lock to a WRITE or EXCLUSIVE lock.

If concurrent transactions simultaneously require this type of upgrade on the same row hash value, a deadlock can result.

For example, assume that two concurrent transactions use the same primary index value to perform a SELECT followed by an UPDATE, as follows (the example assumes the user is in Teradata mode):

1 Locks for row hash operations are not stated by EXPLAIN reports.

User	SQL Text
A	BEGIN TRANSACTION; SELECT y FROM table_a WHERE pi = 1; UPDATE table_a SET y = 0 WHERE pi = 1; END TRANSACTION;
B	BEGIN TRANSACTION; SELECT z FROM table_a WHERE pi = 1; UPDATE table_a SET z = 0 WHERE pi = 1; END TRANSACTION;

In this case, UserA and UserB are allowed to access the rows sharing the same row hash value simultaneously for READ during SELECT processing. When the UserA UPDATE statement requires a WRITE lock on the row hash value, the upgrade request is queued, waiting for the UserB READ lock to be released.

However, the UserB READ lock cannot be released because the UserB UPDATE statement also requires a WRITE lock on the row hash value, and that request is queued waiting for the UserA Read lock to be released.

You can avoid such deadlocks by preceding the transaction with a LOCK ROW [FOR] WRITE/EXCLUSIVE phrase. This phrase does not override any lock already held on the target table.

LOCK ROW is appropriate only for single table selects that are based on a primary index or unique secondary index constraint, as shown in the following example:

User	SQL Text
A	BEGIN TRANSACTION; LOCK ROW FOR WRITE SELECT y FROM table_a WHERE USI = 1; UPDATE table_a SET y = 0 WHERE USI = 1; END TRANSACTION;
B	BEGIN TRANSACTION; LOCK ROW WRITE SELECT z FROM table_a WHERE USI = 1; UPDATE tableA SET z = 0 WHERE USI = 1; END TRANSACTION;

In this example, the UserA request for a row hash WRITE lock is granted, which blocks the UserB request for a WRITE lock on that row hash value. The UserB transaction is queued until the UserA lock is released.

The UserA lock is held until the entire transaction is complete. Thus, the UserB 'LOCK ROW …' request is granted only after the UserA END TRANSACTION statement has been processed.

LOCK and the NOWAIT Option

If your transaction cannot wait in a queue, you can use LOCK with the NOWAIT option.

NOWAIT specifies that the entire transaction, (including one in ANSI mode), is to be aborted if the lock manager cannot place the necessary lock on the target object immediately upon receipt of a statement.

This situation is treated as a fatal error. The user is informed that the transaction was aborted, and any processing performed up to the point at which NOWAIT took effect is rolled back.

LOCK in View Definitions

The LOCK modifier can be specified in CREATE VIEW or REPLACE VIEW definitions. This provides a one-to-one mapping of base table columns to view columns. Thus, the ad hoc user need not worry about impacting transaction processing, and users can alter base table data without impacting any existing report-oriented queries.

Considerations for LOCK FOR ACCESS

A READ lock is normally placed on an object for a SELECT operation, which causes the request to be queued if the object is already locked for WRITE.

If an ad hoc query has no concern for data consistency, the LOCK modifier can be used to override the default READ lock with an ACCESS lock. For example:

```
LOCK TABLE tablename FOR ACCESS
SELECT...
FROM tablename...;
```

Be aware that the effect of LOCK FOR ACCESS is that of reading while writing, so dirty reads are likely to occur with this lock.

This can cause incorrect or inconsistent data to be returned, as follows:

- A select with an ACCESS lock can retrieve data from the target object even when another request is modifying that same object.

  Therefore, results from a request that applies an ACCESS lock can be inconsistent.
- The possibility of an inconsistent return is especially high when the request applying the ACCESS lock uses a secondary index value in a conditional expression.

  If the ACCESS lock is granted on an object being held for WRITE, the constraint value could change between the time the secondary index subtable is located and the time the data row is retrieved.

  Such a change is possible because a satisfied SELECT index constraint is not always double-checked against the base data row.

The LOCK ROW modifier cannot be used to lock multiple row hashes. If LOCK ROW FOR ACCESS is specified with multiple row hashes, the declaration implicitly converts to LOCK TABLE FOR ACCESS.

Example

The possibility of an inconsistent return is especially high when an ACCESS request uses a secondary index value in a conditional expression, because satisfied index constraints are not always rechecked against the retrieved data row.

For example, assuming that QualifyAccnt is defined as a secondary index, the following request:

```
LOCK TABLE AccntRec FOR ACCESS
SELECT AccntNo, QualifyAccnt
FROM AccntRec
WHERE QualifyAccnt = 1587;
```

could return the following result.

```
AccntNo  QualifyAccnt
-------  ------------
   1761          4214
```

In this case, the value 1587 was found in the secondary index subtable, and the corresponding data row was selected and returned. However, the data for account 1761 had been changed by the other user while this selection was in progress.

Returns such as this are possible even if the data is changed or deleted only momentarily by a transaction that is subsequently aborted.

This type of inconsistency can occur even if the data is changed only momentarily by a transaction that is later backed out.

Locking and Transaction Processing

Introduction

A lock placed as part of a transaction is held during processing of the transaction and continues to be held until one of the following events occurs:

- The transaction completes
- The transaction aborts and has completed its rollback
- The system restarts and aborted transactions have completed rollback

During system restart, only update transactions that were in progress at the time of the crash are aborted and rolled back. WRITE and EXCLUSIVE locks remain in place for those transactions until they are rolled back.

Implicit Transactions

In Teradata transaction mode, because an implicit (system-generated) transaction is taken as a single request, the Optimizer can determine what kinds of locks are required by the entire transaction at the time the request is parsed.

Before processing begins, the Optimizer can arrange any table locks in an ordered fashion to minimize deadlocks.

For a single statement transaction, the Optimizer specifies a lock on a row hash only while the step that accesses those rows is executing.

Explicit Transactions

When several requests are submitted as an explicit (user-generated) transaction, the requests are processed one at a time. This means that the Optimizer has no way of determining what locks will be needed by the transaction as a whole.

Because of this, locks are placed as each request is received, and all locks are held until one of the following events occurs:

- Completion of either of these events, depending on the mode, but regardless of when the user receives the data (the spool file might exist beyond the end of the transaction).
  - Outermost END TRANSACTION statement in Teradata mode.
  - COMMIT statement in ANSI mode.
- The transaction aborts.
- The system restarts.

Upgrading System Locks

Introduction

When necessary, the Lock Manager upgrades locks while processing system- or user-generated transactions. The most frequent upgrade is from a READ lock to a WRITE lock. This occurs whenever you select a row and then make an update request for the same row.

Rules for Upgrading Locks

Upgrading system locks helps to minimize deadlocks. The Lock Manager uses the following rules when it upgrades locks.

- If two transactions concurrently hold READ locks for the same data and the first transaction enters an update request, then its READ lock cannot be upgraded to a WRITE lock until the READ lock for the second transaction is released.
- If other transactions are awaiting locks for the same data when the first transaction enters its update request, its READ lock is upgraded before the waiting transactions are given locks. Thus, upgrading an existing lock has higher priority than does granting a new lock.

Lock upgrades are not necessarily handled in the same manner as lock queuing. A lock upgrade can occur when conflicting locks are required by multiple statements in a single transaction.

The current status of a particular session can be determined using the Query Session utility (see *Teradata RDBMS Utilities* for details on how to use QuerySession).

Transaction Semantics: Operating in ANSI or Teradata Transaction Modes

Introduction

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

- ANSI
- Teradata

Default Transaction Mode

Teradata transaction mode ensures compatibility with legacy applications, although it introduces a few caveats:

- When you upgrade an existing Teradata system, you might want to set the system default to Teradata transaction mode to provide compatibility for existing users and applications.
- New customers should consider setting the system default to ANSI transaction mode.

The default transaction mode for a session follows the system default set for that installation. The default mode can be overridden through use of the session options parcel which is submitted to Teradata with the connect or the logon/run parcel sequence.

Session Option: Changing Transaction Modes

To change the transaction mode for a session, do any of the following:

FOR this software ...	USE these commands or options ...	TO switch to this transaction mode ...
BTEQ	`.[SET] SESSION TRANSACTION ANSI`	ANSI
	`.[SET] SESSION TRANSACTION BTET`	Teradata
	Log off prior to entering this command. The command does not take effect until the next logon. See *Teradata BTEQ Reference* for more detail on using BTEQ commands.	

FOR this software ...	USE these commands or options ...	TO switch to this transaction mode ...
Preprocessor2	`TRANSACT(ANSI)`	ANSI

	`TRANSACT(BTET)`
	`TRANSACT(2PC)`
	`TRANSACT(COMMIT)`
		Teradata

	See *Teradata Preprocessor2 Programmer Guide* for more detail on setting Preprocessor options.	
CLIv2	`set tx_semantics = 'A'`	ANSI
	`set tx_semantics = 'T'`	Teradata
	`set tx_semantics = 'D'`	Server default
	See *Teradata Call-Level Interface Version2* for more detail on setting the tx_semantics field.	

Also see “Session Parameters” in the chapter “SQL Data Handling Fundamentals” in *Teradata RDBMS SQL Reference, Volume 1* for information about changing transaction modes.

Terminating Transactions

An application-initiated asynchronous abort causes full transaction rollback in both ANSI and Teradata transaction modes. Such a request is generated by any of the following:

- CLIv2 abort request
- TDP when the application terminates without proper session cleanup
- BTEQ with .ABORT

In both Teradata and ANSI transaction modes, implementation of ANSI transaction semantics includes the rule that if a session is terminated with an open transaction then any effects of that transaction are rolled back.

ANSI transaction mode only recognizes termination of a transaction by the explicit performance of a COMMIT or ABORT/ROLLBACK statement from the application. Teradata does not arbitrarily close a transaction unless its termination is required to preserve the integrity of the database.

In ANSI transaction mode, errors such as constraint violations on an INSERT or UPDATE statement do not roll back an offending transaction, they only roll back the current request.

Mode-Specific SQL Statement Restrictions

Except for the following statements, all SQL statements are valid in both transaction modes:

This SQL statement ...	Is not valid in this transaction mode ...
BEGIN TRANSACTION BT	ANSI
END TRANSACTION ET	
COMMIT [WORK]	Teradata

Session Pool Manager

The Session Pool Manager permits setting the Teradata or ANSI transaction mode for pooled sessions.

Two-Phase Commit Protocol

The 2PC protocol is supported for Teradata, but not for ANSI transaction mode sessions.

If you try to use the 2PC protocol while in ANSI transaction mode, the logon process aborts and an error is reported.

ANSI Session Mode

Definition

ANSI transaction mode is a state in which transaction processing adheres to the rules defined by the ANSI SQL-99 specification.

Apart from transaction semantics, you can write SQL code with explicit specifications to override defaults so that it performs identically in both ANSI and Teradata transaction modes.

Rules

The following rules are enforced in ANSI transaction mode.

- Transactions are always implicit.
- A transaction is opened by the first SQL statement executed in a session or by the first statement performed following the close of a transaction.
- A transaction is initiated when no transaction is currently active and a SQL statement is executed.
- The COMMIT [WORK] or ROLLBACK [WORK] statements close a transaction.

  In core-level ANSI SQL, you must specify the keyword WORK with the COMMIT and ROLLBACK statements. Since Teradata supports WORK as optional word for the ROLLBACK statement, it is also considered optional for the COMMIT statement. If you omit the keyword WORK and the SQL Flagger is enabled, then the statement is flagged as use of an extension.

- You can use the Teradata SQL ABORT statement in place of ROLLBACK. ABORT is a Teradata extension to the ANSI SQL-99 standard.
- Error results roll back only the request that evokes them and not the entire transaction.

  Teradata does not attempt to release any locks placed for a request that is rolled back.
- Control of character truncation of trailing non-blank characters causes errors.

  Use the SUBSTRING function to prevent such errors (see "SUBSTRING" in *Teradata RDBMS SQL Reference, Volume 5*).
- By default, character comparisons are always CASESPECIFIC.
- The default table type semantics for the CREATE TABLE statement is MULTISET.

  This means that duplicate rows are allowed when updating or inserting rows into tables.

Teradata Session Mode

Definition

Teradata transaction mode is a state in which transaction processing follows the rules defined by Teradata over years of evolution.

Teradata mode provides a means for conducting transaction processing by legacy applications.

Apart from transaction semantics, you can write SQL code with explicit specifications to override defaults so that it performs identically in both ANSI and Teradata transaction modes.

Rules

The following rules are enforced in Teradata mode:

- Transaction can be implicit or explicit.
- Explicit transaction boundaries must be specified using BEGIN TRANSACTION (or BT) and END TRANSACTION (or ET) statements (see "BEGIN TRANSACTION" and "END TRANSACTION" in *Teradata RDBMS SQL Reference, Volume 6*).
- Statement failures roll back the entire transaction.
- Errors roll back the entire transaction.
- Control of character truncation does not cause errors.
- The default in character comparison is NOT CASESPECIFIC.
- The default table type semantics for the CREATE TABLE statement is SET.

  This means that duplicate rows are not allowed when updating or inserting rows into tables.
- If an explicit transaction contains a DDL statement, then it must be the last statement in the transaction.

  Teradata enforces the rule that a DDL (Data Definition Language) statement must be the *last* statement of a transaction.

  Teradata treats the DATABASE and SET SESSION statements as DDL statements.
- Statements performed through a logon startup string follow the rules for transaction definition and termination and the rule that a DDL statement must be the last statement of a transaction.

Comparison of Transaction Rules in ANSI and Teradata Transaction Modes

The following table compares the transaction rules for ANSI and Teradata transaction modes:

Rule	ANSI Transaction Mode	Teradata Transaction Mode
Transaction declaration	Always implicit.	Either implicit or explicit.
Transaction initiation and termination	Transactions initiate when no transaction is active and an SQL statement is performed. Opened by one of the following: • First SQL statement performed in a session. • First statement performed after the previous transaction terminates. In both cases, the transaction terminates with either a COMMIT WORK or ROLLBACK WORK (or ABORT) statement.	Opened by one of the following: • Each statement performed in a session. • BEGIN TRANSACTION statement. If statements are grouped into explicit transactions in this way, they must be terminated by a matching END TRANSACTION statement. Transaction rollbacks are specified by an ABORT statement.
Error behavior	Errors roll back only the request that causes them, not the entire transaction. Locks placed by erroneous statements are not released.	Errors roll back the entire transaction.
Statement failure behavior	Not applicable.[a]	Statement failures roll back the entire transaction.
Default attribute for character comparisons	CASESPECIFIC	NOT CASESPECIFIC
Control of character truncation errors	Records errors for truncation problems with trailing non-blank characters.	Truncation problems do not cause errors.
Table type semantics	MULTISET	SET

Rule	ANSI Transaction Mode	Teradata Transaction Mode
Default TRIM function behavior	Trims both leading and trailing pad characters as if TRIM (BOTH FROM *expression*) were explicitly specified.	Trims only trailing pad characters as if TRIM(TRAILING FROM *expression*) were explicitly specified.
DDL statement placement	Not applicable.[b]	Must be last statement in transaction.
Statements performed through a logon startup string	Not applicable.	Must follow standard rules for: • Transaction definition • Transaction termination • DDL statement placement

a. Because each statement is an implicit transaction, there is nothing to roll back if a statement fails.
b. Because each statement is an implicit transaction, there is no first or last statement in a transaction.

Rollback Processing

Introduction

This section describes the differences in rollback processing for Teradata and ANSI transaction modes.

Application-Initiated Asynchronous Abort

An application-initiated asynchronous ABORT causes full transaction rollback in both transaction modes. This occurs by means of a CLIv2 abort request or by the TDP when the application terminates without proper session cleanup.

Teradata Transaction Mode

During transaction processing, either all requests are performed, or none are. Another way of stating this is to say that all transactions are atomic.

If, for any reason, a transaction cannot be completed successfully, or if it times out, the entire transaction aborts and rollback processing is performed.

Rollback processing, also called abort processing, performs the following actions in Teradata transaction mode:

Stage	Process
1	Rolls back all changes made to the database as a result of the transaction
2	Releases any locks applied as a result of statements in the transaction
3	Erases any partially accumulated results (spool files).

The rollback process constitutes transaction recovery. If the amount of work performed by a transaction is not properly controlled, the following things might occur:

- Locks applied on behalf of the transaction might block other sessions.
- If the system must restart during the transaction, rollback of the work already performed by the transaction might delay post-restart system availability.

ANSI Transaction Mode

ANSI only recognizes termination of a transaction by the performance of a COMMIT WORK or ABORT/ROLLBACK WORK statement performed by the application. Teradata does *not* terminate a transaction unless it needs to preserve the integrity of the database.

If a request errs, only the current request is rolled back, and not other requests previously performed in the transaction. The Lock Manager does not release locks placed for a rolled back request.

The entire transaction is rolled back when the current request is in one of the following states:

- Deadlocked.
- Aborted DDL statement.

  Either of the above situations are needed before the locks held by the transaction are released.
- Rejected request for a LOCK NOWAIT option.

Locking Issues With Tactical Queries

Introduction

Tactical queries deliver better response times and throughput using row hash locks whenever possible. Row hash locks are preferable for the following reasons:

- Higher concurrency

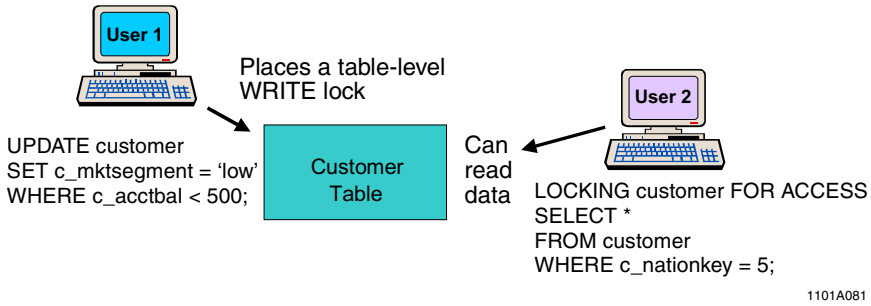
  If a lock is placed on only one or few rows, the other rows in the table can be accessed or updated by other users at the same time.
- Fewer resources required

  Row hash locks require less resources to apply because only a single AMP is engaged. When you apply table-level locks, work must be performed and coordinated across all AMPs in the system. Maintenance of table-level locks always requires two separate all-AMPs steps: the first to set the lock and the second to release it.
- Greater scalability

  When the fewest resources are marshaled to satisfy a request, then a greater number of similar requests can be processed in parallel. Throughput increases both as the number of concurrent users increases and as more nodes are added to the configuration.

ACCESS Locks and Tactical Queries

ACCESS locks provide greater throughput if updating by one group of queries and reading by another is occurring against the same tables. ACCESS locks permit you to have read access to an object that might already be WRITE- or READ-locked.



When you use ACCESS locks, there is a risk that the view of the data being accessed is inconsistent. Data in the process of being updated might be returned to a requestor as if it were consistent. If this is not acceptable for an application, then you should not use ACCESS locks.

Row Hash-Level Versus Table-Level ACCESS Locks

ACCESS locks can reduce wait times for queries, but they can also add unnecessary work if they are not handled carefully. You must understand the granularity of the lock and the nature of the query to ensure not to add unnecessary overhead to the workload.

For example, the modifier LOCKING TABLE customer FOR ACCESS requests a table-level lock and results in an all-AMPs operation even if there is only a single-AMP step in the query plan. Using this locking modifier can add two additional, unnecessary all-AMPs steps to the query plan and forces extra Dispatcher steps to be sent between the PE and the AMP.

In the following EXPLAIN report, note the separate step, Step 1, generated to perform the all-AMPs table-level lock, and the additional step, Step 3, that releases the table level lock across all AMPs:

```
EXPLAIN
LOCKING TABLE customer FOR ACCESS
SELECT c_name, c_acctbal
FROM customer
WHERE c_custkey = 93522;
```

```
Explanation
------------------------------------------------------------------------
  1) First, we lock CAB.customer for access.
  2) Next, we do a single-AMP RETRIEVE step from CAB.customer by
     way of the unique primary index "CAB.customer.C_CUSTKEY =
     93522" with no residual conditions.  The estimated time for this
     step is 0.03 seconds.
  3) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

For a single-AMP operation such as reading a single row using a primary index value, LOCKING ROW FOR ACCESS is always a better locking modifier to use than locking the entire table. You can see in the EXPLAIN report that a row-level ACCESS lock is applied and that only one AMP is used to process the query.

The following EXPLAIN text illustrates a row-level ACCESS lock:

```
EXPLAIN
LOCKING ROW FOR ACCESS
SELECT c_name, c_acctbal
FROM customer
WHERE c_custkey = 93522;
```

```
Explanation
------------------------------------------------------------------------
  1) First, we do a single-AMP RETRIEVE step from CAB.customer by
     way of the unique primary index "CAB.customer.C_CUSTKEY =
     93522" with no residual conditions locking row for access.  The
     estimated time for this step is 0.03 seconds.
```

Automatic Lock Escalation

Even if you explicitly specify a row hash-level ACCESS lock, the Optimizer automatically converts it to a table-level lock if the query plan requires an all-AMPs operation and there is only one table referenced in the query. Always check the EXPLAIN report to verify that row hash or, when appropriate, table-level ACCESS locks are issued for tactical queries.

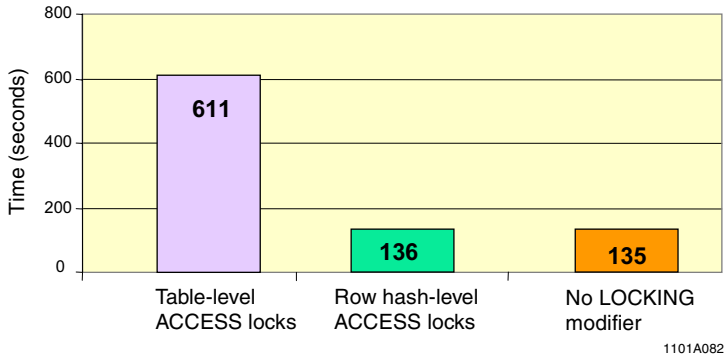
The following EXPLAIN report for an all-AMPs query shows that the Optimizer applies a table-level ACCESS lock even though the LOCKING modifier explicitly requests a row hash-level ACCESS lock.

```
EXPLAIN
LOCKING ROW FOR ACCESS
SELECT c_name, c_acctbal
FROM customer
WHERE c_nationkey = 15;
```

```
Explanation
------------------------------------------------------------------------
1) First, we lock CAB.customer for access.
2) Next, we do an all-AMPs RETRIEVE step from CAB.customer by way
   of an all-rows scan with a condition of ("CAB.customer.C_NATIONKEY = 15")
   into Spool 1, which is built locally on the AMPs. The input table will not be cached
   in memory, but it is eligible for synchronized scanning.  The size of Spool 1
   is estimated with high confidence to be 300,092 rows.  The estimated time for
   this step is 2 minutes and 8 seconds.
3) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
```

Even if more than one table is specified in the query, if a LOCKING ROW FOR ACCESS modifier has been specified, the Optimizer applies a table-level ACCESS lock for all tables undergoing all-AMP access in that query.

The following graph illustrates the cost of using table-level ACCESS locks compared to row hash-level ACCESS locks when the query itself only performs single-AMP operations.



The elapsed time represents the total time to perform 100 000 single-row SELECT statements using 20 sessions. When table-level ACCESS locks were specified in the query, total execution time for this workload increased by a factor of 5. The response times for the variables labeled Row hash-level ACCESS locks and No locking modifier are almost identical because when no locking

modifier was specified, the Optimizer applied a row hash-level READ lock in the background. This lock has the same overhead as the row-level ACCESS lock.

Some query tools make it difficult to specify an ACCESS lock modifier. In spite of this, you can enforce explicit ACCESS locking by querying views and placing the appropriate LOCKING modifier in their view definitions.

Row-Level ACCESS Locks and Join Indexes

The Optimizer propagates row-level ACCESS locks to join indexes where appropriate. Assume the following single table join index defined on the customer table. The UPI for the customer table is c_custkey. A query makes a request, specifying a value for c_name and requesting that a row-level ACCESS lock be applied to the customer table.

```
CREATE JOIN INDEX CustNameJI
AS SELECT c_name, c_acctbal, c_mktsegment, c_range
FROM customer
PRIMARY INDEX(c_name);

EXPLAIN
LOCKING ROW FOR ACCESS
SELECT c_acctbal, c_mktsegment, c_range
FROM customer
WHERE c_name =   'Customer#000000999' ;
```

```
Explanation
------------------------------------------------------------------------
1) First, we do a single-AMP RETRIEVE step from CAB.NAMEJI by way of the primary
index "CAB.NAMEJI.C_NAME = 'Customer#000000999'" with a residual condition of
("CAB.NAMEJI.C_NAME = 'Customer#000000999'") into Spool 1, which is built locally
on that AMP.  The input table will not be cached in memory, but it is eligible for
synchronized scanning locking row for access.  The size of Spool 1 is estimated
with high confidence to be 1 row.
```

The join index covers the query and provides the requested customer table data based on the specific c_name value specified. The Optimizer applies the requested row hash-level ACCESS lock to the join index row hash, not to the base table row hash.

Locking for Updates

A simple update query that specifies a primary index value for the table cues the Optimizer to apply a row hash-level WRITE lock to process the update. Row hash-level WRITE or row hash-level READ locks are not reported in the EXPLAIN text. This query updates the non-indexed order table column o_orderpriority by accessing a single row using the UPI defined on o_orderkey. Only a single AMP and a row hash lock are used to process this query.

```
EXPLAIN
UPDATE orders
SET o_orderpriority = 5
WHERE o_orderkey = 39256
```

```
Explanation
------------------------------------------------------------------------
  1) First, we do a single-AMP UPDATE from CAB.orders by way of
     the unique primary index "CAB.orders.O_ORDERKEY = 39256"
     with no residual conditions.
  -> No rows are returned to the user as the result of statement 1.
```

Locking for Complex Updates

The expression “complex update” is sometimes used as a synonym for join updates, but as used here, the meaning of the expression is broader. Complex updates include the following types:

- Statements that reference tables other than the table being updated.
- Statements that update the primary index.

When it detects complex updates being issued concurrently by several sessions, the Optimizer generates a plan to perform the individual update statements in series because each update applies a table-level lock.

The following query updates the non-indexed orders table column o_orderpriority by accessing a single row using the UPI defined on o_orderkey. The update also includes a subquery that qualifies the answer set based on the existence of lineitem rows with this same value for o_orderkey.

Because a join is specified in the WHERE clause, this is a complex update and two spool files are generated to accommodate the single-column update.

1. The first spool uses a merge delete against the target table to remove all the rows that are to be updated.
2. The second spool file uses a merge to insert the updated rows into the target table.

Because all AMPs are used to satisfy this query, the Optimizer automatically applies table-level WRITE locks.

The statement is a complex update because it has an EXISTS clause that references all rows from the lineitem table, while only the orders table is actually updated. If the subquery in the EXISTS clause had specified a lineitem primary index value, then the query could have been solved as a single-AMP operation.

```
EXPLAIN
UPDATE orders
SET o_orderpriority = 5
WHERE o_orderkey = 39256
AND EXISTS
  (SELECT * FROM lineitem
   WHERE l_orderkey = o_orderkey);
```

```
Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct CAB."pseudo table" for write on a
     RowHash to prevent global deadlock for CAB.orders.
  2) Next, we lock a distinct CAB."pseudo table" for read on a
     RowHash to prevent global deadlock for CAB.lineitem.
  3) We lock CAB.orders for write, and we lock CAB.lineitem for read.
  4) We do a single-AMP JOIN step from CAB.orders by way of the
     unique primary index "CAB.orders.O_ORDERKEY = 39256" with no
     residual conditions, which is joined to CAB.lineitem.
     CAB.orders and CAB.lineitem are joined using an
     inclusion merge join, with a join condition of (
     "CAB.lineitem.L_ORDERKEY = CAB.orders.O_ORDERKEY").  The
     input tables CAB.orders and CAB.lineitem will not be
     cached in memory.  The result goes into Spool 1, which is
     redistributed by hash code to all AMPs.  Then we do a SORT to
     order Spool 1 by the sort key in spool field1.  The size of Spool
     1 is estimated with low confidence to be 5 rows.  The estimated
     time for this step is 1 hour and 18 minutes.
  5) We do a MERGE DELETE to CAB.orders from Spool 1 (Last Use)
     via the row id.  New updated rows are built and the result goes
     into Spool 2, which is built locally on the AMPs.  Then we do a
     SORT to order Spool 2 by row hash.
  6) We do a MERGE into CAB.orders from Spool 2 (Last Use).
  7) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

Complex updates such as this one, with multiple all-AMPs steps, do not pose a performance issue when generated at low frequency. But if you need to perform a high volume of such queries, the all-AMPs operations and the table-level WRITE lock impair scalability as their volume increases.

Complex updates that are part of a tactical query are better handled when their individual statements are specified within a stored procedure. To perform this query, the stored procedure you would issue 2 single-AMP queries, each applying only single row hash locks. The following 2 EXPLAIN reports represent the SQL statements that could be written inside the stored procedure.

```
EXPLAIN
SELECT *
FROM lineitem
WHERE l_orderkey = 39256;
```

```
Explanation
-----------------------------------------------------------------------
  1) First, we do a single-AMP RETRIEVE step from CAB.lineitem by
     way of the primary index "CAB.lineitem.L_ORDERKEY = 39256"
     with no residual conditions into Spool 1, which is built locally
     on that AMP.  The input table will not be cached in memory, but it
     is eligible for synchronized scanning.  The size of Spool 1 is
     estimated with high confidence to be 4 rows.  The estimated time
     for this step is 0.15 seconds.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 0.15 seconds.
```

The stored procedure would only execute the second request if at least one lineitem row was returned from the first request.

```
EXPLAIN
UPDATE orders
SET o_orderpriority = 5
WHERE o_orderkey = 39256

Explanation
  1) First, we do a single-AMP UPDATE from CAB.orders by way of
     the unique primary index "CAB.orders.O_ORDERKEY = 39256"
     with no residual conditions.
```

The above two SQL statements would need to be written within an explicit transaction, using BEGIN TRANSACTION and END TRANSACTION statements to define the transaction boundaries.

Locking for Single Row Updates With Indexes

Single row inserts, updates and deletes are generally single-AMP operations. However, a different effort is expended, sometimes involving several or all AMPs, when indexes on the base table must also be maintained or when the indexes themselves are updated.

The following table indicates the extent of locking and the number of additional AMPs involved in single-row update operations when the base table has various types of indexes. Information is based on the assumption that each UPDATE statement specifies a value for the primary index in the WHERE clause. Updates to the row that do not involve the indexed columns are considered separately from updates that change those columns

Set of Objects Updated	Insert	Delete	Update Values in Non-Index Columns	Update Values in Index Columns
Base Table Only	1 row hash lock 1 AMP	1 row hash lock 1 AMP	1 row hash lock 1 AMP	Table lock All AMPs (PI update)
Base Table With 1 NUSI				
Base Table With 1 USI	1 additional row hash lock for USI 1 additional AMP	1 additional row hash lock for USI 1 additional AMP		Table lock All AMPs (USI update)
Base Table With 1 Join Index	Table lock on join index All AMPs for join index	Table or row hash lock on join index All or same AMP for join index[a]		Table or row hash lock on join index All or same AMP for join index[b]

a. If the DELETE statement carries the primary index value of the join index, Teradata applies a row-level lock to the join index structure and a single-AMP update takes place. If the join index has a primary index definition different from the base table, then because no primary index value for the join index is available, Teradata applies a table-level WRITE lock and performs an all-AMPs operation to delete from the join index structure.
b. If the UPDATE statement carries the primary index value of the join index, and the value being updated is either in the join index repeating group or anywhere in the select list if no repeating group is defined, only under these conditions does Teradata apply a row hash-level lock to the join index structure. Otherwise, the Lock Manager applies a table-level lock on the join index during the update and an all-AMPs operation is performed to implement it.

Several important conclusions can be made from this table:

- Simple base table insert, delete or update operations with no secondary or join indexes require minimal levels of locking and access the fewest AMPs.
- When you update a base table row but do not update the PI or USI values within that row, the query accesses only one or a few AMPs and performance should be scalable.
- If you update the index values, a query that includes a NUSI update still provides good scalability, but a query updating a USI value does not. This is because the USI update causes all AMPs to be accessed with a table-level lock.
- A row-at-a-time insert, such as those performed by TPump, into a table that carries only a single-table join index uses the more scalable row hash-level locks on the join index structure, and accesses only a single AMP when performing the join index maintenance.

- A single row insert on a table that carries a multitable join index places a table-level WRITE lock on the join index table unless the insert accesses the row using a primary index value, in which case a row hash-level WRITE lock is exerted.

  While this does not impact availability to the base table (the base table will only have a row hash-level lock), the WRITE lock prevents the join index from being updated by any other statement during its duration. In addition, any SELECT statements that might include that join index in their query plan must wait for the WRITE lock to be released.
- The table expresses the overhead for a *single* index of its type, but the documented overhead is additive. If multiple NUSIs exist on a table, all of the maintenance involved still takes place on the same AMP and there are no additional locks set, but an increasing number of subtable rows must be accessed and updated. This involves both additional physical I/O and additional CPU capacity.

  The number of AMPs involved in USI maintenance increases as a function of the number of USIs on a table. Each join index that base table underlies experiences either a row hash- or a table-level lock each time a participating column from that base table is updated and each time an entire row is inserted or deleted.

  Always run EXPLAIN reports on the base table UPDATE statement to verify the locking involved for its constituent join indexes.
- Other variations of indexing, like value-ordered NUSIs, follow the same behavior as traditional NUSIs in terms of the performance impact of their maintenance.

Appendix A:

Notation Conventions

This appendix describes the notation conventions used in this book.

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.
Spaces	Use one space between items, such as keywords or variables.
Punctuation	Enter all punctuation exactly as it appears in the diagram.

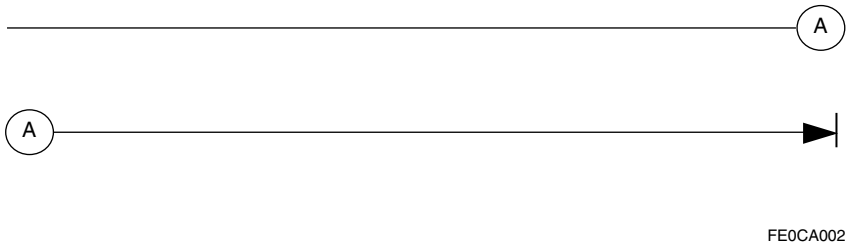
IF a word is shown in . . .	THEN it represents . . .
UPPERCASE LETTERS	a keyword. Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase. If a keyword is shown in uppercase, you may enter it in uppercase or mixed case.
lowercase letters	a 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.

Paths

The main path along the syntax diagram begins at the left, and proceeds, left to right, to the vertical bar, which marks the end of the diagram. Paths that do not have an arrow or a vertical bar only show portions of the syntax.

The only part of a path that reads from right to left is a loop.

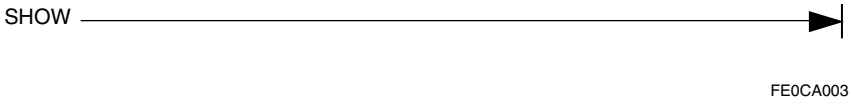
Paths that are too long for one line use continuation links. Continuation links are small circles with letters indicating the beginning and end of a link:



When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue.

Required Items

Required items appear on the main path:



If you can choose from more than one item, the choices appear vertically, in a stack. The first item appears on the main path:



Optional Items

Optional items appear below the main path:



If choosing one of the items is optional, all the choices appear below the main path:



You can choose one of the options, or you can disregard all of the options.

Abbreviations

If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.

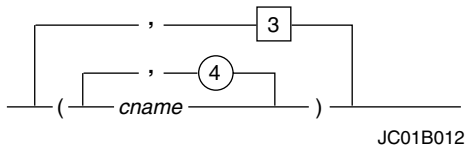


In the above syntax, the following formats are valid:

- SHOW CONTROLS
- SHOW CONTROL

Loops

A loop is an entry or a group of entries that you can repeat one or more times. Syntax diagrams show loops as a return path above the main path, over the item or items that you can repeat.



The following rules apply to loops:

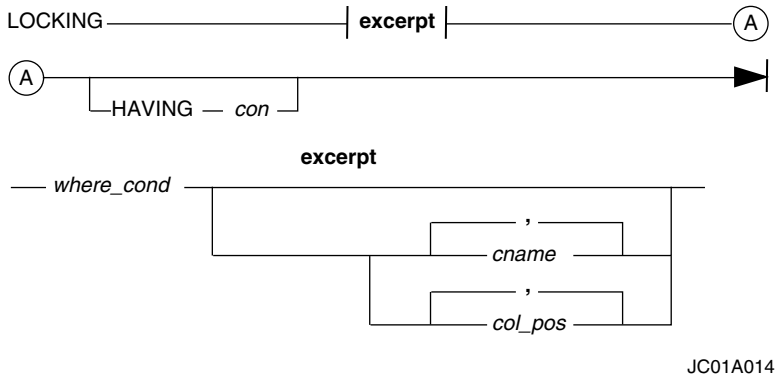
IF . . .	THEN . . .
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may enter *cname* a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must enter at least 3 groups of column names.
a separator character is required between entries	the character appears on the return path. If the diagram does not show a separator character, use one blank space. In the example, the separator character is a comma.

IF . . .	THEN . . .
a delimiter character is required around entries	the beginning and end characters appear outside the return path. Generally, a space is not needed between delimiter characters and entries. In the example, the delimiter characters are the left and right parentheses.

Excerpts

Sometimes a piece of a syntax phrase is too large to fit into the diagram. Such a phrase is indicated by a break in the path, marked by | terminators on either side of the break. A name for the excerpted piece appears between the break marks in boldface type.

The named phrase appears immediately after the complete diagram, as illustrated by the following example.



Character Shorthand Notation Used In This Book

Introduction

This book uses the UNICODE naming convention for characters. For example, the lowercase character ‘a’ is more formally specified as either LATIN SMALL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the book, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings.

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in Appendix B: “Japanese Character Sets," in *Teradata RDBMS International Character Set Support*.

Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a–z A–Z 0–9	Any	Any single byte Latin letter or digit.
a–z A–Z 0–9	Unicode compatibility zone	Any fullwidth Latin letter or digit.
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Used to indicate transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Used to indicate transition from multibyte to single byte KanjiEBCDIC.

Symbol	Encoding	Meaning
T	Any	Any multibyte character. Its encoding depends on the current character set. For KanjiEUC, code set 3 characters are sometimes shown preceded by "ss_3".
I	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by "ss_2", forming an individual multibyte character.
Δ	Any	Represents the graphic pad character.
Δ	Any	Represents either a single or multibyte pad character, depending on context.
ss_2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss_3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string "TEST", where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set

```
LMN<TEST>QRS
```

is represented as:

```
D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2
```

Pad Characters

The following table lists the pad characters for the various character data types.

Character Data Type	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20

Predicate Calculus Notation Used in This Book

Relational databases are based on the theory of relations as developed in set theory. Predicate calculus is often the most unambiguous way to express certain relational concepts.

Occasionally this book uses the following predicate calculus notation to explain concepts

.

This symbol ...	Represents this phrase ...
iff	If and only if
"	For all
$	There exists

Index

A

ACCESS lock
  row hash-level vs. table-level for tactical queries 7–44
  tactical queries 7–43
  tactical queries with join indexes 7–46
ACCESS lock, described 7–4
Activity Transaction Modeling process 5–10
AddWorkload macro 5–3
All AMPs statistical sampling 2–40
All-AMPs sampling 2–33
ALTER TABLE, locks and 7–10
AnalysisLog table 4–18
Apply
  component processes of 1–29
  described 1–29
ATM process 5–10
Attribute value skew 2–29

B

BEGIN QUERY LOGGING statement 5–3
Blocked locking requests 7–21
  multi-statement transactions 7–22
  Query Session utility 7–21
  single-statement transactions 7–21
Bucket, definition for interval histograms 2–27
Bushy search tree for join order optimization 2–68

C

Cache
  Data Dictionary 1–6
  request 1–23
  statement 1–23
Caching
  immediate 1–23
  non-immediate 1–24
CHECKSUM lock, described 7–4
CNF, see conjunctive normal form
COLLECT DEMOGRAPHICS statement 4–8, 4–10
COLLECT STATISTICS (Optimizer Form)
  statement 5–15
COLLECT STATISTICS statement 4–8, 4–10, 4–29
  COLLECT STATISTICS USING SAMPLE 2–22
  QCD form 5–10, 5–17
Collecting statistics 2–20, 2–42
Common step 1–19
Compressed interval histogram
  definition 2–28
  description 2–27
Concrete step 1–30
Concurrency control, lock manager and 7–6
Concurrency, locking 7–10
Conjunctive normal form 2–57
Correlated join 2–119
Cost functions, optimizer 2–59
Cost optimization
  definition 2–56
  path selection 2–57
Cost-based query optimization 2–5
CREATE INDEX statement 4–29, 5–15
  locks 7–10
Creating QCD tables
  procedure using BTEQ 4–9
  VEComp procedure 4–7
Cursors, positioned, locking modes 7–19
Cursors, updatable, locking modes 7–19

D

Data Dictionary
  cache 1–6
  How used by the Parser 1–6
Database lock, described 7–6
Database Query Analysis
  Index analysis 5–11
  index application 5–18
  Workload definition 5–8
Database Query Log 5–5, 5–7, 5–10
DataDemographics table 4–20
DBC.TableSizeX system view 4–21
DBC.TVFields, FieldStatistics column 2–30
DBQL 5–10
DBQM utility 5–19
DCL statements, locks and 7–17
DDL statements, locks and 7–17

Deadlocks
  access mode 7–25
  detection 7–24
  prevention 7–26
    example 7–11
    LOCKING FOR ACCESS 7–30
    LOCKING modifier
      NOWAIT 7–29
    LOCKING, view definitions 7–30
  prevention with LOCKING modifier 7–26
  resolution 7–24
Disjunctive normal form 2–57
Dispatcher
  described 1–31
  execution control 1–31
  request abort management 1–32
  response control 1–31
  transaction management 1–32
DML statements, locks and 7–18
DNF, see disjunctive normal form
DROP INDEX statement 4–29, 5–15
Duplicate rows
  MULTISET table kind option 7–37
  SET table kind option 7–38
Dynamic AMP sampling 2–35

E

Environmental cost factors 2–62
  external cost parameters 2–62, 2–63
  performance constants 2–62
Equal-height interval
  definition 2–28
Equal-height interval histogram 2–25
  definition 2–28
Exclusion join 2–110
  exclusion merge 2–111
  exclusion product 2–114
EXCLUSIVE lock, described 7–3

F

Field table 4–22

G

Generator
  component processes of 1–22
  described 1–21
  plastic steps and 1–21

H

Hash join 2–93
  HTMemAlloc DBSControl field 2–95
  SkewAllowance DBSControl field 2–95
High-biased interval 2–26
  definition 2–28
High-biased interval histogram 2–26
  definition 2–28
  loner 2–28, 2–29
Histogram
  compressed interval histogram 2–28
  contents 2–30
  definition 2–28
  equal-height interval 2–28
  equal-height interval histogram 2–28
  high-biased interval 2–28
  high-biased interval histogram 2–28
  how stored 2–30
HTMemAlloc 2–95

I

Inclusion join 2–116
  inclusion merge 2–116
  inclusion product 2–116
Inclusion merge join 2–116
Index
  ensuring use 2–60
  optimizer use for indexed access 2–60
Index analysis 5–11
Index application 5–18
Index Wizard utility 4–28, 4–69, 4–70, 5–3, 5–6, 5–8, 5–11, 5–15, 5–17, 5–19
  AnalysisLog table 4–18
  DataDemographics table 4–20
  how used 5–3
  IndexRecommendations table 4–27
Index_Field table 4–25
IndexColumns table 4–26
Indexed access 2–60
  cost optimization 2–60

IndexRecommendations table 4–27
INITIATE INDEX ANALYSIS 4–69
INITIATE INDEX ANALYSIS statement 4–18, 4–28, 4–29, 4–30, 5–15
INSERT EXPLAIN statement 4–8, 4–10, 5–10
  WITH STATISTICS 5–8
Interval
  definition for interval histograms 2–28
Interval histogram
  cardinality estimates 2–46
  compressed 2–27
  described 2–25
  equal-height 2–25
  high-biased interval 2–26
  storage 2–30
  terminology 2–27
  types 2–25
  values collected and computed 2–44

J

Japanese character code notation, how to read A–6
Join algorithms
  types 2–64
Join algorithms, see Join methods 2–80
Join index
  examples 3–33, 3–34, 3–35, 3–36, 3–37
  maintenance 3–39
  maintenance, examples 3–41, 3–43, 3–44, 3–46
Join methods 2–80
  correlated join 2–119
  exclusion join 2–110
  exclusion merge join 2–111
  exclusion product join 2–114
  hash join 2–93
  inclusion join 2–116
  inclusion merge join 2–116
  inclusion product join 2–116
  merge join 2–86
  minus all join 2–121
  nested join 2–97
  product join 2–82
  remote nested join 2–104
Join modes, see Join methods
Join order
  as a function of the number of tables to be joined 2–70
  search tree 2–67
Join order optimization
  bushy search trees 2–68
  left-deep search tree branches 2–69
  right-deep search tree branches 2–69
Join order search tree types
  bushy 2–68
  left-deep 2–67
Join planning 2–64
  evaluating join orders 2–72
  five-join lookahead 2–79
  join processing methods 2–64
  one-join lookahead 2–76
  process overview 2–72
  table lookahead 2–76
Join types
  local nested join 2–98
Joins
  methods 2–80
  Optimizer and join plans 2–66
  row elimination 2–71
Joins. See individual join methods

L

Large Table/Small Table join. See LT/ST join
Left-deep search tree branches for join order optimization 2–69
Left-deep search trees for join order optimization 2–67
Lock escalation
  automatic 7–45
Lock manager
  described 7–6
  locking levels 7–6
LOCK ROW modifier 7–30
Locks
  ACCESS 7–4
  blocked requests 7–21
  CHECKSUM 7–4
  compatibility among modes 7–5
  cursor locking modes 7–19
  deadlock detection 7–24
  EXCLUSIVE 7–3
  explicit transaction 7–32
  Lock Manager and 7–7
  maximizing concurrency 7–10
  modes 7–3
  object levels 7–6
  READ 7–3
  when released 7–10
  WRITE 7–3

Locks. See also Deadlocks
Loner 2–28
  definition 2–29
Lookahead
  five-join 2–79
  one-join 2–76
LT/ST join
  described 3–3
  examples 3–15
  indexed join
    reasonable 3–8
    unreasonable 3–8
  unindexed join
    reasonable 3–9
    unreasonable 3–9

M

Merge join 2–86
Minus All Join 2–121
MULTISET table kind option 7–37

N

Nested join 2–97
  local 2–98
  remote 2–104
NOWAIT, locking modifier 7–5, 7–29

O

Optimizer
  aggregate sharing 2–18
  bushy search trees 2–68
  cardinality estimation 2–46
  component processes of 1–18
  compressed interval histograms 2–27
  conjunctive normal form 2–57
  correlated join 2–119
  correlated subquery decorrelation 2–17
  cost estimation and statistics 2–59
  cost functions 2–59
  cost optimization 2–56
  database object substitution 2–19
  described 1–14
  disjunctive normal form 2–57
  environmental cost factors 2–62
  equal-height interval histograms 2–25
  evaluating join orders 2–72
  exclusion join 2–110
  exclusion merge join 2–111
  exclusion product join 2–114
  five-join lookahead planning 2–79
  hash join 2–93
  high-biased interval histograms 2–26
  inclusion join 2–116
  inclusion merge join 2–116
  inclusion product join 2–116
  indexed access 2–60
  interval histograms 2–25
  join methods 2–80
  join order as a function of the number of tables to be joined 2–70
  join order search trees 2–67
  join planning 2–64, 2–66
  join plans 2–60
  join processing methods 2–64
  join table lookahead 2–76
  left-deep search trees 2–67
  local nested join 2–98
  merge join 2–86
  minus all join 2–121
  nested join 2–97
  one-join lookahead planning 2–76
  parallel steps and 1–19
  parse tree representation of a query 2–8
  path selection 2–57
  predicate martialing 2–14
  predicate push down and pullup 2–16
  process overview 2–6
  product join 2–82
  query rewrite 2–14
  redundant join elimination 2–18
  remote nested join 2–104
  statistical profiles 2–44
  translation to internal representation 2–8
  view materialization 2–19
Optimizer statistics
  values collected and computed 2–44

P

Parallel step 1–19
Parse tree representation of a query 2–8
Parser
  component processes of 1–4
  major components of 1–2
Partition skew 2–29
Path selection 2–57

Plastic step 1–22
Predicate martialing 2–14
Predicate push down and pullup 2–16
Predicate table 4–35
Predicate_Field table 4–37
Product join 2–82

Q

QCD
  applications 4–4
  capacity planning 4–4
  defined 4–5
  information source 4–3
  non-compatibility with pre-5.0 releases 4–2
  physical model 4–6
  setting up 4–7
  ways to query 4–13
QCD database 5–3, 5–5, 5–8, 5–10, 5–15, 5–17, 5–19
  AnalysisLog table 4–18
  DataDemographics table 4–20
  Field table 4–22
  Index_Field table 4–25
  IndexColumns table 4–26
  IndexRecommendations table 4–27
  Predicate table 4–35
  Predicate_Field table 4–37
  QryRelX table 4–38
  Query table 4–40
  QuerySteps table 4–44
  Relation table 4–51
  SeqNumber table 4–57
  TableStatistics table 4–59
  User_Database table 4–62
  UserRemarks table 4–63
  ViewTable 4–65
  Workload table 4–67
  WorkloadQueries table 4–68
  WorkloadStatus table 4–69
QCD macros 4–14
QCD tables 4–17
QCF
  applications 4–4
QryRelX table 4–38
Queries, querying QCD 4–13
Query Capture Database. See QCD
Query Capture Facility. See QCF
Query cost optimization 2–56
Query optimization
  artificial intelligence, compared with 2–4
  compiler optimization, compared with 2–3
  objectives of 2–4
Query optimizer
  described 2–3
  need for 2–2
  parallels with artificial intelligence systems 2–4
  parallels with compiler optimization 2–3
  types of 2–5
Query rewrite 2–14
  aggregate sharing 2–18
  correlated subquery decorrelation 2–17
  database object substitution 2–19
  predicate martialing 2–14
  predicate push down and pullup 2–16
  redundant join elimination 2–18
  techniques 2–14
  view materialization 2–19
Query Session utility
  blocked lock requests 7–21
Query table 4–40
QuerySteps table 4–44

R

Random AMP sampling 2–20, 2–21, 2–35, 2–40, 2–60
  accuracy 2–37
  estimated statistics 2–59
READ lock, described 7–3
Recollecting statistics 2–42
Relation table 4–51
Request cache, see Statement cache
Residual statistics
  effects of cardinality on usefulness 2–38
  effects of number of distinct values on usefulness 2–38
  relative accuracy 2–37
Resolver
  component processes of 1–12
  described 1–11
Right-deep search tree branchess for join order optimization 2–69
Rollback processing
  ANSI mode 7–42
  Teradata mode 7–41
Row hash lock, described 7–6
Row hash locking 7–30
Rule-based query optimization 2–5

S

Sampled statistics
  all AMPs 2–20, 2–21
  random AMP 2–20, 2–21
Security checking
  described 1–13
SeqNumber table 4–57
Session management, blocked requests 7–21
Session modes
  ANSI 7–34
  ANSI rules 7–37
  Teradata 7–34
  Teradata rules 7–38
SET table kind option 7–38
Skew
  attribute value skew 2–29
  definition 2–29
  partition skew 2–29
SkewAllowance 2–95
Snowflake join, defined 3–4
Snowflake join. See LT/ST join
SQL statements
  BEGIN QUERY LOGGING 5–3
  COLLECT DEMOGRAPHICS 4–8, 4–10
  COLLECT STATISTICS 2–22, 4–8, 4–10, 4–29
  COLLECT STATISTICS (Optimizer Form) 5–15
  COLLECT STATISTICS (QCD Form) 5–10, 5–17
  CREATE INDEX 4–29, 5–15
  DROP INDEX 4–29, 5–15
  INITIATE INDEX ANALYSIS 4–18, 4–28, 4–29, 4–30, 4–69, 5–15
  INSERT EXPLAIN 4–8, 4–10, 5–8, 5–10
  LOCKING modifier 7–26
Star join
  examples 3–15
  indexes for, selecting 3–12
  Optimizer and 3–5
Star join. See LT/ST join
Statement cache
  described 1–23
  management structure 1–24
  matching process 1–25
  purging 1–27
  statement parsing and 1–23
Statistical profiles and the optimizer 2–44
Statistics
  accuracy 2–20, 2–40
  all AMPs sampling 2–20, 2–21, 2–40
  all-AMPs sampled 2–33
  collecting 2–20
  collection time as a function of cardinality 2–22
  collection time as a function of number of distinct values 2–22
  collection time as a function of system configuration 2–22
  performance overhead 2–21, 2–22
  policies for collecting 2–43
  purposes 2–20
  random AMP sampled 2–33
  random AMP sampling 2–20, 2–21, 2–35, 2–60
  ranking methods of collection for accuracy 2–40, 2–41
  reasons for collecting 2–20
  relative accuracy of various methods of collecting 2–40
  residual 2–37
  residual and cardinality 2–38
  residual and distinct values 2–38
  time overhead 2–21, 2–22
  values collected and computed 2–44
  when to collect 2–42, 2–43
  when to recollect 2–42
Statistics Wizard 2–24
Step, defined 1–19
Step. See Plastic step, Concrete step, Parallel step, Common step
Support tools
  Teradata System Emulation Tool (TSET) 6–3
  VEComp facility 6–3
  Query Capture Facility. See also QCF
Support tools, VEComp facility 4–1, 4–4
Syntaxer
  component processes of 1–10
  components 1–8
  described 1–8

T

Table lock, described 7–6
TableStatistics table 4–59
Tactical queries
  ACCESS locks 7–43
  automatic lock escalation 7–45
  complex update 7–47
  locking for complex update 7–47
  locking for simple update 7–46
  locking for single row hash update with indexes 7–49
  locking for update 7–46
  locking issues 7–43

row hash-level ACCESS locks and join indexes 7–46
row hash-level versus table-level ACCESS locks 7–44
Target Level Emulation Facility
benefits of emulation 6–2, 6–3
capturing cost data 6–2
capturing Optimizer environmental cost information 6–10
capturing random AMP sample statistical data 6–2
dynamic emulation 6–3
enabling 6–6
mapping files to a test system 6–12
static emulation 6–3
SystemFE.Opt_Cost_Table 6–3
target system 6–2
Target system (production system). See Target Level Emulation
Teradata Index Wizard utility 4–28, 4–69, 4–70, 5–3, 5–6, 5–8, 5–11, 5–15, 5–17, 5–19
AnalysisLog table 4–18
DataDemographics table 4–20
how used 5–3
IndexRecommendations table 4–27
Teradata Statistics Wizard 2–24
Teradata System Emulation Tool utility 5–5, 5–7, 5–9, 5–15, 5–17
Transaction modes
ANSI 7–34
ANSI rules 7–37
rollback processing in ANSI mode 7–42
rollback processing in Teradata mode 7–41
Teradata 7–34
Teradata rules 7–38
Transaction semantics 7–34
Transactions
lmplicit locking 7–32
multi-statement, blocked locking requests 7–22
rollback of 7–41
rules for DDL placement in Teradata mode 7–38
single-statement, blocked locking requests 7–21
Trees
bushy search 2–68
join order search 2–67
left-deep search 2–67
left-deep search branches 2–69
right-deep search branches 2–69
TSET utility 5–5, 5–7, 5–9, 5–15, 5–17, 6–3

U

Unicode, notation A–6
User_Database table 4–62
UserRemarks table 4–63

V

View lock, described 7–6
ViewTable 4–65
Visual EXPLAIN utility 5–5, 5–6, 5–17

W

Workload definition 5–8
Workload definition macros, AddWorkload 5–3
Workload table 4–67
WorkloadQueries table 4–68
WorkloadStatus table 4–69
WRITE lock, described 7–3