Preface

Purpose

SQL Request and Transaction Processing describes the 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 transaction processing environment.

This preface describes the organization of SQL Request and Transaction Processing and identifies information you should know before using it. This book should be used in conjunction with the other SQL volumes.

Audience

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

Supported Software Release

This book supports Teradata® Database 13.0.

Prerequisites

If you are not familiar with Teradata Database, you will find it useful to read Introduction to Teradata and SQL Fundamentals 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 SQL Quick Reference.
# Changes to This Book

<table>
<thead>
<tr>
<th>Release</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teradata Database 13.0</td>
<td>Updated information about collecting statistics on volatile tables.</td>
</tr>
<tr>
<td>March 2010</td>
<td></td>
</tr>
<tr>
<td>Teradata Database 13.0</td>
<td>• Clarified the way locks are placed for SELECT requests.</td>
</tr>
<tr>
<td>August 2009</td>
<td>• Revised the following EXPLAIN example: “SELECT with WHERE on nonunique index.”</td>
</tr>
<tr>
<td>Teradata Database 13.0</td>
<td>Corrected cross reference in chapter 3.</td>
</tr>
<tr>
<td>July 2009</td>
<td></td>
</tr>
<tr>
<td>Teradata Database 13.0</td>
<td>• Chapters 2 and 3 updated for NoPI tables.</td>
</tr>
<tr>
<td>April 2009</td>
<td>• Added new query rewrite modules to Chapters 1 and 2.</td>
</tr>
<tr>
<td></td>
<td>• Added several new Query Rewrite methods to Chapter 2.</td>
</tr>
<tr>
<td></td>
<td>• Updated cost profile descriptions in Chapter 2.</td>
</tr>
<tr>
<td></td>
<td>• Added new INCLUSION/EXCLUSION PRODUCT JOINs with DPE to chapter 3.</td>
</tr>
<tr>
<td></td>
<td>• Updated PRODUCT JOIN with DPE for cross terms in Chapter 3.</td>
</tr>
<tr>
<td></td>
<td>• Updated ROWKEY-BASED MERGE JOIN for cross terms in Chapter 3.</td>
</tr>
<tr>
<td></td>
<td>• Added new EXPLAIN text phrases to chapter 5.</td>
</tr>
<tr>
<td></td>
<td>• Added AnalysisStmts and XMLQCD QCD tables to Chapter 6.</td>
</tr>
<tr>
<td></td>
<td>• Added new Appendix B.</td>
</tr>
<tr>
<td></td>
<td>• Moved existing Appendix B to Appendix C.</td>
</tr>
<tr>
<td>Release</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Teradata Database 12.0</td>
<td>• Split old chapter 2 into two new chapters: chapter 2 covering basic query</td>
</tr>
<tr>
<td>September 2007</td>
<td>rewrite and query optimization, and chapter 3 covering join planning and</td>
</tr>
<tr>
<td></td>
<td>join optimization.</td>
</tr>
<tr>
<td></td>
<td>• Added coverage of request cache peeking to chapters 1 and 7.</td>
</tr>
<tr>
<td></td>
<td>• Added coverage of Query Rewrite subsystem to chapters 1 and 2.</td>
</tr>
<tr>
<td></td>
<td>• Added coverage of derived statistics to chapter 2.</td>
</tr>
<tr>
<td></td>
<td>• Added coverage of NumAllNulls and AvgAmpRPV statistics to chapter 2.</td>
</tr>
<tr>
<td></td>
<td>• Added coverage of statistics extrapolation to chapter 2.</td>
</tr>
<tr>
<td></td>
<td>• Documented increased number of histogram intervals from a maximum of</td>
</tr>
<tr>
<td></td>
<td>100 to a maximum of 200 in chapter 2.</td>
</tr>
<tr>
<td></td>
<td>• Added coverage of delayed and static partition elimination to chapter 2.</td>
</tr>
<tr>
<td></td>
<td>• Added additional coverage of cost profiles to chapter 2.</td>
</tr>
<tr>
<td></td>
<td>• Added the following join types and variations to chapter 3:</td>
</tr>
<tr>
<td></td>
<td>• Direct PPI Merge Join</td>
</tr>
<tr>
<td></td>
<td>• Product Join with Dynamic Partition Elimination</td>
</tr>
<tr>
<td></td>
<td>• Rowkey-Based Merge Join</td>
</tr>
<tr>
<td></td>
<td>• Single Window Merge Join</td>
</tr>
<tr>
<td></td>
<td>• Sliding Window Merge Join</td>
</tr>
<tr>
<td></td>
<td>• Added new EXPLAIN text phrases to chapter 5.</td>
</tr>
<tr>
<td></td>
<td>• Added coverage of the PartitionRecommendations and RangePartExpr QCD</td>
</tr>
<tr>
<td></td>
<td>tables to chapter 6.</td>
</tr>
<tr>
<td></td>
<td>• Added new columns to the QuerySteps table in chapter 6:</td>
</tr>
<tr>
<td></td>
<td>• Updated coverage of the Teradata Index Wizard in chapter 7.</td>
</tr>
<tr>
<td></td>
<td>• Added coverage of the Teradata Statistics Wizard to chapter 7.</td>
</tr>
<tr>
<td></td>
<td>• Added coverage of partition analysis to chapter 7.</td>
</tr>
<tr>
<td></td>
<td>• Added information about cost profiles to chapter 8.</td>
</tr>
<tr>
<td></td>
<td>• Added information about HUT locks to chapter 9.</td>
</tr>
</tbody>
</table>
Additional Information

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.info.teradata.com/">www.info.teradata.com/</a></td>
<td>Use the Teradata Information Products Publishing Library site to:</td>
</tr>
<tr>
<td></td>
<td>• View or download a manual:</td>
</tr>
<tr>
<td></td>
<td>1 Under Online Publications, select General Search.</td>
</tr>
<tr>
<td></td>
<td>2 Enter your search criteria and click Search.</td>
</tr>
<tr>
<td></td>
<td>• Download a documentation CD-ROM:</td>
</tr>
<tr>
<td></td>
<td>1 Under Online Publications, select General Search.</td>
</tr>
<tr>
<td></td>
<td>2 In the Title or Keyword field, enter CD-ROM, and click Search.</td>
</tr>
<tr>
<td></td>
<td>• Order printed manuals:</td>
</tr>
<tr>
<td></td>
<td>Under Print &amp; CD Publications, select How to Order.</td>
</tr>
<tr>
<td><a href="http://www.teradata.com">www.teradata.com</a></td>
<td>The Teradata home page provides links to numerous sources of information about Teradata. Links include:</td>
</tr>
<tr>
<td></td>
<td>• Executive reports, case studies of customer experiences with Teradata, and thought leadership</td>
</tr>
<tr>
<td></td>
<td>• Technical information, solutions, and expert advice</td>
</tr>
<tr>
<td></td>
<td>• Press releases, mentions and media resources</td>
</tr>
<tr>
<td><a href="http://www.teradata.com/t/TEN/">www.teradata.com/t/TEN/</a></td>
<td>Teradata Customer Education designs, develops and delivers education that builds skills and capabilities for our customers, enabling them to maximize their Teradata investment.</td>
</tr>
<tr>
<td><a href="http://www.teradataatyourservice.com">www.teradataatyourservice.com</a></td>
<td>Use Teradata @ Your Service to access Orange Books, technical alerts, and knowledge repositories, view and join forums, and download software patches.</td>
</tr>
<tr>
<td>developer.teradata.com/</td>
<td>Teradata Developer Exchange provides articles on using Teradata products, technical discussion forums, and code downloads.</td>
</tr>
</tbody>
</table>

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.teradata.com
References to Microsoft Windows and Linux

This book refers to “Microsoft Windows” and “Linux.” For Teradata Database 13.0, these references mean:

- “Windows” is Microsoft Windows Server 2003 64-bit.
- “Linux” is SUSE Linux Enterprise Server 9 and SUSE Linux Enterprise Server 10.
Preface

References to Microsoft Windows and Linux
# Table of Contents

**Preface** ................................................................. 3
  Purpose ................................................................. 3
  Audience ............................................................... 3
  Supported Software Release ........................................ 3
  Prerequisites .......................................................... 3
  Changes to This Book ................................................ 4
  Additional Information ................................................ 6
  References to Microsoft Windows and Linux ....................... 7

**Chapter 1: Request Parsing** ............................................ 17
  The SQL Parser ....................................................... 18
  Dictionary Cache .................................................... 22
  Request Cache ........................................................ 23
  Purging the Request Cache ......................................... 28
  Syntaxer ............................................................... 30
  Resolver .............................................................. 33
  Security Checking ................................................... 35
  Peeking at the Request Cache ....................................... 36
  Query Rewrite ........................................................ 54
  Optimizer ............................................................. 59
  Generator ............................................................. 66
  OptApply .............................................................. 68
  Dispatcher ............................................................. 70
# Chapter 2: Query Rewrite and Optimization

Query Rewrite ............................................................... 74
Typical Query Rewrite Techniques .................................. 90
Converting ANSI Join Syntax To Comma Join Syntax ............... 91
Projection Pushdown ...................................................... 92
OUTER JOIN-To-INNER JOIN Conversion ........................ 94
View Folding ............................................................... 96
Satisfiability and Transitive Closure .................................. 102
Predicate Simplification ................................................ 109
Predicate Pushdown and Pullup ....................................... 115
Eliminating Set Operation Branches .................................. 117
Eliminating Redundant Joins .......................................... 118
Pushing Joins Into UNION ALL Branches ......................... 121
Other Query Rewrites .................................................. 122
Predicate Marshaling .................................................... 123
View Materialization and Other Database Object Substitutions 125
Query Optimizers ........................................................ 126
Teradata Database Optimizer Processes ............................... 132
Translation to Internal Representation ................................ 134
Optimizer Statistics ....................................................... 140
How the AMP Software Collect Statistics .......................... 152
Interval Histograms ....................................................... 161
Sampled Statistics ....................................................... 175
Random AMP Sampling ............................................... 178
Comparing the Relative Accuracies of Various Methods of Collecting Statistics 188
When Should Statistics Be Collected Or Recollected? ............. 190
How the Optimizer Uses Statistical Profiles .......................... 193
Examples of How Initial Cardinality Estimates Are Made for Simple Queries Using Derived Statistics .............................................. 206
Interval Histograms ....................................................... 197
Statistical Inheritance ................................................... 216
Derived Statistics ........................................................ 217
Deriving Column Demographics ...................................... 217
Discovering Join Indexes ............................................... 231
Stale Statistics ............................................................. 251
Optimized Join Cardinality Estimates When There Is Single-Row Access to One of the Tables Using a Unique Index .............................................. 253
Using Extrapolation to Replace Stale Statistics ...................... 254
Chapter 3: Join Planning and Optimization

Optimizer Join Plans .................................................. 318
Join Geography ....................................................... 333
Evaluating Join Orders ............................................... 344
Lookahead Join Planning ........................................... 349
Partial GROUP BY Block Optimization ......................... 353
Join Methods ........................................................... 361
Product Join ............................................................. 363
Merge Join ............................................................... 372
Direct PPI Merge Join ............................................... 381
Rowkey-Based Merge Join ......................................... 383
Single Window Merge Join ......................................... 386
Sliding Window Merge Join ......................................... 389
Hash Join ................................................................. 397
Nested Join .............................................................. 405
Local Nested Join ...................................................... 407
Slow Path Local Nested Join ...................................... 408
Fast Path Local Nested Join ....................................... 411
Remote Nested Join .................................................. 412
Nested Join Examples ................................................ 415
Join Plan Without Nested Join ................................... 416
Join Plan With Nested Join ......................................... 417
Exclusion Join .......................................................... 418
Exclusion Merge Join ................................................ 419
Exclusion Product Join .............................................. 422
Inclusion Join ........................................................... 424
Inclusion and Exclusion Product Joins With Dynamic Partition Elimination ........ 425
Chapter 4: Join Optimizations

Star and Snowflake Join Optimization ............................................................. 442
LT/ST-J1 Indexed Joins .................................................................................. 448
LT-ST-J2 Unindexed Joins ............................................................................ 449
Miscellaneous Considerations for Star Join Optimization ........................ 450
Selecting Indexes for Star Joins ................................................................. 452
Star Join Examples ...................................................................................... 455
Cardinality and Uniqueness Statistics for the Reasonable Indexed Join Examples ................................. 457
Reasonable Indexed Join Plan Without Star Join Optimization ............. 458
Reasonable Indexed Join Plan With Star Join Optimization: Large Table  Primary Index Joined to Small Tables .......................................................... 460
Reasonable Indexed Join Plan With Star Join Optimization: Large Table  USI Joined to Small Tables .................................................................. 462
Reasonable Indexed Join Plan With Star Join Optimization: Large Table  NUSI Joined to Small Tables .................................................................. 464
Join Plan With Star Join Optimization: Large Table Subquery Join ........ 465
Cardinality and Uniqueness Statistics for the Reasonable Unindexed Join Examples .............................. 467
Reasonable Unindexed Join Without Join Optimization ......................... 468
Reasonable Unindexed Join With Join Optimization ............................... 469
Join Indexes ................................................................................................. 470
Maintenance of Join Index for DELETE, INSERT, and UPDATE .......... 474
General Method of Maintaining Join Index During DELETE ................... 476
Optimized Method of Maintaining Join Index During DELETE ............... 477
Maintaining Join Index During INSERT ..................................................... 478
General Method of Maintaining Join Index During UPDATE ................... 480
Optimized Method of Maintaining Join Index During UPDATE ............... 482
# Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN Request Modifier ................................................................. 484
EXPLAIN Confidence Levels ............................................................... 487
EXPLAIN Request Modifier Terminology .............................................. 493
EXPLAIN Modifier in Greater Depth ..................................................... 503
EXPLAIN: Examples of Complex Queries ............................................ 504
EXPLAIN Request Modifier and Join Processing .................................... 512
EXPLAIN and Standard Indexed Access ................................................. 517
EXPLAIN and Parallel Steps ................................................................. 520
EXPLAIN Request Modifier and Partitioned Primary Index Access .......... 522
EXPLAIN Request Modifier and MERGE Conditional Steps ................. 529
EXPLAIN and UPDATE (Upsert Form) Conditional Steps ....................... 534

# Chapter 6: Query Capture Facility

Compatibility Issues With Prior Teradata Database Releases .................. 546
Functional Overview of the Query Capture Facility .................................. 547
Query Capture Database ........................................................................ 549
Creating the QCD Tables ....................................................................... 551
Dropping the QCD Tables ...................................................................... 553
Querying QCD ....................................................................................... 555
QCD Query Macros and Views ............................................................... 556
QCD Table Definitions .......................................................................... 559
AnalysisLog ......................................................................................... 560
AnalysisStmts ...................................................................................... 562
DataDemographics ............................................................................... 564
Field ................................................................................................... 566
Index_Field ......................................................................................... 569
IndexColumns ...................................................................................... 570
IndexMaintenance ............................................................................... 571
IndexRecommendations ....................................................................... 573
IndexTable .......................................................................................... 578
JoinIndexColumns ............................................................................... 582
PartitionRecommendations .................................................................. 585
Predicate ............................................................................................. 589
Chapter 7: Database Foundations for the Teradata Index and Statistics Wizards

Teradata Index Wizard Overview ......................................................... 635
DBS Control Flags for Tuning the Workload Cache Limits for Index and Partition Analysis .......................................................... 640
Workload Identification ................................................................. 641
Workload Definition ................................................................. 643
Index and Partitioning Expression Analysis ....................................... 646
Viewing Index and Partitioning Expression Recommendations .......... 650
Index and Partitioning Expression Validation ..................................... 652
Index and Partitioning Expression Application .................................. 655
Effects of Request Cache Peeking on INSERT EXPLAIN and DUMP EXPLAIN Outcomes ................................................................. 657
An Extended Example .................................................................. 658
Examples of Partitioned Primary Indexes the Index Wizard/
INITIATE PARTITION ANALYSIS Might Recommend ......................... 664
Optimizer Statistics Recommendations and the Teradata Statistics Wizard .... 666
### Chapter 8: Target Level Emulation

- An Overview of Target Level Emulation .......................... 680
- Procedures to Enable Target Level Emulation and OCES DIAGNOSTIC Statements With Target Level Emulation .......................... 684
- Procedures to Capture Target System Optimizer Cost Values and Random AMP Statistical Samples for a Test Environment .................. 685
- Target Level Emulation and Cost Profiles .......................... 690
- Procedure To Capture and Install Costing Information Including the Active Cost Profile .......................... 691
- Mapping Target System Files to a Test System Optimizer Table and GDO .......................... 693

### Chapter 9: Locking and Transaction Processing

- Database Transactions .................................................. 696
- Transactions, Requests, and Statements .......................... 702
- Database Locks, Two-Phase Locking, and Serializability ....... 706
- Canonical Concurrency Problems .................................... 713
- Lock Manager ......................................................... 719
- Locking and Transaction Processing ............................... 722
- Teradata Database Locking Levels and Severities .............. 723
- Client Utility Locks and Teradata Database ........................ 732
- Default Lock Assignments and Lock Upgradeability ........... 739
- Blocked Requests ...................................................... 746
- Pseudo Table Locks .................................................. 750
- Deadlock .............................................................. 753
- Deadlock Detection and Resolution ................................. 755
- Preventing Deadlocks ................................................. 756
- Example: Transaction Without Deadlock ......................... 761
- Example: Transaction With Deadlock ............................. 762
- Example: Two Serial Transactions .................................. 764
- DDL and DCL Statements, Dictionary Access, and Locks .... 766
- DML Requests and Locks ............................................. 767
- Cursor Locking Modes ............................................... 772
- Transaction Semantics: Operating in ANSI or Teradata Session Modes ................................ 774
- ANSI Session Mode .................................................. 777
- ANSI Mode Transaction Processing Case Studies ................ 779
- Teradata Session Mode ............................................... 783
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teradata Mode Transaction Processing Case Studies</td>
<td>785</td>
</tr>
<tr>
<td>Comparison of Transaction Rules in ANSI and Teradata Session Modes</td>
<td>789</td>
</tr>
<tr>
<td>Rollback Processing</td>
<td>791</td>
</tr>
<tr>
<td>Locking Issues With Tactical Queries</td>
<td>793</td>
</tr>
<tr>
<td>References</td>
<td>801</td>
</tr>
<tr>
<td><strong>Appendix A: Notation Conventions</strong></td>
<td>813</td>
</tr>
<tr>
<td>Syntax Diagram Conventions</td>
<td>813</td>
</tr>
<tr>
<td>Character Shorthand Notation Used In This Book</td>
<td>819</td>
</tr>
<tr>
<td>Predicate Calculus Notation Used In This Book</td>
<td>821</td>
</tr>
<tr>
<td><strong>Appendix B: Content and Structure of XML Documents</strong></td>
<td>823</td>
</tr>
<tr>
<td>XML Plan Contents</td>
<td>824</td>
</tr>
<tr>
<td>XML Schema Used for the XMLPLAN Option</td>
<td>827</td>
</tr>
<tr>
<td>Example XML Optimizer Query Plan Document Produced by the XMLPLAN Option for BEGIN QUERY LOGGING</td>
<td>832</td>
</tr>
<tr>
<td>Sample Java Program For Processing XML Documents Like Those Produced by the XMLPLAN Option for BEGIN QUERY LOGGING</td>
<td>836</td>
</tr>
<tr>
<td>Related Topics</td>
<td>840</td>
</tr>
<tr>
<td><strong>Appendix C: References</strong></td>
<td>841</td>
</tr>
<tr>
<td>Query Processing References</td>
<td>842</td>
</tr>
<tr>
<td>Transaction Processing References</td>
<td>847</td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td>851</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>877</td>
</tr>
</tbody>
</table>
CHAPTER 1 Request Parsing

This chapter describes SQL request parsing, including the components of the SQL Parser that deal with query processing.
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 requests
- Control information
- Optional USING request modifier data

Major Components of the Parsing Engine

The Parsing Engine has six major and several lesser components related to query processing:

- The Parser
  - Request Cache
  - Syntaxer
  - Resolver
  - Security Checking
- Request Cache Peek
- Query Rewrite
  - Push projections into spooled views
  - Convert outer joins to inner joins
  - Fold views
  - Solve satisfiability and transitive closure
  - Push conditions into spooled views
  - Eliminate joins
- The Optimizer
  - Access planning
  - Join planning
  - Index Wizard
- The Generator
  - Steps generation
  - Steps packaging
• The Dispatcher
  • Execution control
  • Response control
  • Transaction and request abort management
  • Queue table cache management

The Parser also maintains a dictionary cache for the Resolver, Security, and the Optimizer and manages sessions between client applications and the Teradata platform.

**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 7: “Database Foundations for the Teradata Index and Statistics Wizards” and Teradata Index Wizard User Guide.

Parsing Engine Component Processes

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

1. The Syntaxer analyzes the high-level syntax of the statement for errors.
   
   If the syntax passes the check, then the SQL request 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 an SQL Request” on page 134).
   
   This skeletal parse tree is called a SynTree, which the Syntaxer then passes on to the Resolver. The SynTree is also referred to as the Black Tree for the query.

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 Request Cache Peek.
   
   The ResTree is also referred to as the Red Tree for the query.

3. The Request Cache Peek subsystem examines the ResTree for any parameterized request or CURRENT_DATE constant data that can be used to generate more specific query plans than are possible without having such resolved data. Request Cache Peek then passes the ResTree to Query Rewrite.

4. The Query Rewrite subsystem takes the ResTree from Request Cache Peek and rewrites the SQL text to prepare it for optimization.

   This subsystem includes the following specific types of query rewrite:

   - Converting outer joins to inner joins
   - View folding
     - Type 1 view folding
     - Type 2 view folding
   - Pushing projections into views
   - Pushing conditions into views
   - Satisfiability and Transitive Closure
   - Join elimination

   The Query Rewrite subsystem then passes the revised, semantically equivalent, ResTree’ to the Optimizer. See “Query Rewrite” on page 74 for further information.

5. The Optimizer analyzes the ResTree’ using various statistical and configuration data about the database and the system hardware components 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 request and attempts to optimize their placement to enhance performance and avoid deadlocks.

   The Optimized Parse Tree tree now transformed from a simple statement tree to a complex operation tree, is then passed to the Steps Generator for further processing.
This optimized version of the parse tree is referred to as the *White Tree*, or *Operation Tree*, for the request.\(^1\) When you perform an EXPLAIN of a request, the report the system produces is a verbal description of the White Tree the Optimizer produces for the request plus some additional information about non-costed steps that the system inserts into the White Tree for use by the Teradata Workload Manager.


6 The Steps Generator creates Plastic Steps from the White Tree. 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 the Request Cache and to Steps Packaging for further processing.

7 Steps Packaging adds context to the Plastic Steps by integrating various user- and session-specific information.

If any Data parcels\(^2\) were passed to the Parser via a parameterized request, then that data is also added to the steps tree. The final product of this process is referred to as Concrete Steps.

8 Steps Packaging passes the Concrete Steps to the Dispatcher for assignment to the AMPs.

9 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.

10 The Dispatcher manages any abort processing that might be required.

11 The Dispatcher receives the results of the AMP Steps from the BYNET and returns them to the requesting application.

12 End of process.

---

1. As the ResTree is transformed into the Operation Tree, it is sometimes referred to as a Pink Tree because at that intermediate point it is a mix of red and white, hence pink.

2. In this chapter, the term Data parcel always refers to a Data parcel set. A noniterated request is associated with only one Data parcel, while an iterated request is associated with multiple Data parcels. A request can also have no Data parcels associated with it.
Dictionary Cache

Introduction
To transform an SQL request 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 161 and “Environmental Cost Factors” on page 296), 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 DBS Control to Fine Tune Cache
For each PE, both the default and the maximum size of the dictionary cache is 1 Mbyte (1024 Kbyte). You can also control how many of its most recent 550 object references are retained in cache.

See Utilities for further information about fine tuning the dictionary cache.
Chapter 1: Request Parsing

Request Cache

Definition

The Request Cache stores certain categories of successfully parsed SQL requests and their plastic steps so they can be reused, eliminating the need to reparse the same request parcel (see “Peeking at the Request Cache” on page 36 for a description of the category of successfully parsed SQL requests that the system does not cache). The Request Cache is a PE-local buffer that stores the steps generated during the parsing of a DML request.

The Value of the Request Cache

The Request Cache is particularly useful for batch update programs that repeatedly issue the same requests with different data values because all requests submitted during a given session are routed to the same PE, and so access the same Request Cache.

The Request Cache is also useful in a transaction processing environment where the same DML requests are entered by a number of users using the same application program.

The Role of the Request Cache in Request Parsing

The Parser checks the Request Cache at the beginning of the parsing process, before the Syntaxer step, but after the Request parcel has been checked for format errors. If it finds a matching cached request, the Parser bypasses the Syntaxer, Resolver, Optimizer, and Generator steps, performs a security check (if required), and proceeds to the OptApply stage.

Note that the Request Cache Peek subsystem is a component of the Parser, and it, too, operates on all incoming requests. See “Peeking at the Request Cache” on page 36).

The Parser does not always cache nonprimary index parameterized queries that are sent with a Data parcel. Depending on the parameterized values, the Optimizer might choose a different query execution plan than what has been cached because reusing the cached plan might not always be the best choice. To deal with this possibility, the system plugs the parameterized constant values into a request before the Optimizer has to determine which query plan to use.

By exposing parameterized values before determining a query plan, it becomes possible to generate a more optimal plan for some categories of specific requests than could be generated without first peeking at those values. The result is identical to what would be achieved had you, for example, specified constants instead of USING variables in all points in the request where the Optimizer considers value predicates. See “Peeking at the Request Cache” on page 36 for details about how this is done.

3. The term parameterized is used instead of USING because it is possible to send both data and DataInfo parcels with a request without specifying a USING request modifier and to specify a USING request modifier with just a Data parcel and no DataInfo parcel. See Teradata Call-Level Interface Version 2 Reference for Channel-Attached Systems and Teradata Call-Level Interface Version 2 Reference for Network-Attached Systems for information about Data and DataInfo parcels.
Because cached requests and their plastic steps can be shared across logons, the Parser always makes a security check on them. 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.

When a request has a Data parcel (specified in Teradata SQL by a USING request modifier), the system caches it immediately. Whether a request uses a generic or specific query plan is determined by the Request Cache Peek subsystem (see “Peeking at the Request Cache” on page 36). The Request Cache Peek subsystem determines whether or not the query plan should also be cached.

The following examples show SQL statements that produce requests with Data parcels:

```sql
USING (a INTEGER, b INTEGER, c SMALLINT)
INSERT INTO tablex VALUES (:a, :b, :c);

USING (d SMALLINT, e INTEGER)
EXEC macroy (:d, :e);
```

If these requests were submitted as part of a data-driven iterative request, then multiple Data parcels would be involved (see Teradata Call-Level Interface Version 2 Reference for Channel-Attached Systems or Teradata Call-Level Interface Version 2 Reference for Network-Attached Systems for more information about iterated requests).

**Nonimmediate Caching**

The system does not immediately cache parameterized macros that do not specify USING request modifiers because they do not have Data parcels. The following macro does not have a Data parcel because it does not have a USING request modifier:

```sql
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 cached immediately. Instead, a hash value derived from the request text is stored in one of the first-seen entries in the cache management data structure.

If the same request is submitted to the Parser a second time, the request and its plastic steps are cached, and the system moves its entry from the first-seen area into one of the cache entries in the cache management data structure.

---

4. In this chapter, the term *Data parcel* always refers to a Data parcel set. A noniterated request is associated with only one Data parcel, while an iterated request is associated with multiple Data parcels. A request can also have no Data parcels associated with it.
The Request Cache Management Structure

Each Request Cache consists of several major components:

- A data structure to manage the cached requests
  - This structure contains information about one of the cached requests, plus a number of first-seen entries containing information about requests that have not yet been cached.
  - The structure is always memory-resident and its size changes dynamically so it is only as large as necessary to contain all the cached requests.
- The SQL text and plastic steps for the individual requests
  - The cached requests consist of the text of the SQL requests and, optionally, their plastic steps.
  - Individual cached requests are stored in PE memory.

Although the Request Cache is not shared among the PEs on a system, 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 given PE.

Criteria For Caching a DML Request

Teradata Database uses the following criteria to determine whether to cache a DML query plan or not:

<table>
<thead>
<tr>
<th>If a DML request has …</th>
<th>THEN its steps are …</th>
</tr>
</thead>
<tbody>
<tr>
<td>a USING request modifier</td>
<td>cached the first time the request is parsed if the request is USING-independent (see “Peeking at the Request Cache” on page 36). If the request is USING-dependent, then the steps might not be cached.</td>
</tr>
<tr>
<td>no USING request modifier</td>
<td>not cached the first time the request is parsed. If the request is parsed a second time, then the steps are cached.</td>
</tr>
</tbody>
</table>

The determination of whether the cached plan is specific to the request or generic is made by the Request Cache Peek subsystem (see “Peeking at the Request Cache” on page 36).
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.

To match a cached request, a new request must:

- Match the candidate cached SQL request text character-for-character. An exception is made only for values in the Data parcels of USING-independent requests.
- Have been submitted from a client with the same host format as the candidate cached request (see International Character Set Support for details).
- Use the same character set as the candidate cached request (see International Character Set Support for details).
- Use the same collation sequence as the candidate cached request (see “SET SESSION COLLATION” in SQL Data Definition Language and International Character Set Support for details).
- Match the date value returned by the CURRENT_DATE or DATE built-in functions if the request is specific to a resolved date. The Parser purges such plans when it encounters the same request for a changed CURRENT_DATE or DATE built-in function value.

Request Cache Matching Process

This topic 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, as seen in the following process:

1. The Parser creates a preliminary list of variables derived from the new request that consists of the following items:
   - SQL text hash value
   - Request parsing time
   - Request run time
   - Request error conditions
   - Request length
   - Various other flags
The Parser tests the preliminary variable list for the new request against the same information for the requests in the Request Cache.

<table>
<thead>
<tr>
<th>IF the preliminary information ...</th>
<th>THEN the Parser ...</th>
</tr>
</thead>
<tbody>
<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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<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 reuse.</td>
</tr>
</tbody>
</table>

End of process.
Chapter 1: Request Parsing
Purging the Request Cache

Purging the Request Cache

Introduction

Cached query plans remain in the Request Cache until the system spoils them, which means they are no longer valid. Whenever the Parser receives a DDL request, it broadcasts a spoil message to all PEs. DDL changes alter the database schema and can make previously valid query plans non-valid.

The spoil message causes PEs to delete only those cached requests (and their corresponding entries in the cache management data structure) that reference the database object that is changed by the DDL request.

Cached plans are also purged if they are specific to a resolved date value that no longer matches the date returned by the CURRENT_DATE or DATE built-in functions. The system purges such plans when the same request is seen for a changed CURRENT_DATE or DATE built-in function.

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 Request 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.

Nonexempt Requests

All cached requests that are not marked exempt are purged periodically. The purge times are phased among the PEs so that all are not purged simultaneously.

Purging a Full Cache

The maximum number of entries possible in the Request Cache depends on the setting for the MaxRequestsSaved flag in the DBS Control record (see Utilities for details). The default is 600 entries, with minima and maxima at 300 and 2,000 entries, respectively. You can increase the values in increments of 10 entries.

When all the Request Cache entries are full, the Parser uses a least-recently-used algorithm to determine which requests 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 Request Cache Entries

Periodically, the system purges the Request Cache 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 Request Cache Entries

The system purges Request Cache entries 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 request are discarded.
- A data definition request (for example, an ALTER TABLE request) is submitted for a table that has been specified by a cached request.
  In this case, the cached steps for the request are discarded.
Chapter 1: Request Parsing
Syntaxer

**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, also called the Black Tree, which it then passes on to the Resolver.

A parse tree is a data structure used by the 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 and from derived statistics. The Syntaxer does not transform the query text in any way.

The parse tree also permits a relatively simple transformation of the Request parcel by Query Rewrite and the Optimizer 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 for a request.

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.
A block diagram of Syntaxer activity is shown in the following illustration:
**Syntaxer Component Processes**

1. The Syntaxer checks the Request Cache to determine if it contains an identical Request parcel.

<table>
<thead>
<tr>
<th>IF an identical parcel is ...</th>
<th>THEN the Syntaxer ...</th>
</tr>
</thead>
<tbody>
<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 set specified in the USING request modifier. Plastic steps are flexible, thereby allowing different sets of values from USING request modifiers 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>
</tbody>
</table>

2. If no matching parcel is found in the Request 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. The Syntaxer produces the SynTree as its output and passes it to the Resolver.

3. End of process.

**SQL Flagger Activity**

When the SQL Flagger is enabled, the Syntaxer is the component of the Parser that performs 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, also called a Red Tree, which it passes to Security, Request Cache Peek, and then to Query Rewrite. The Resolver does not transform the query text in any way.

Block Diagram of Resolver Activity

The following graphic shows a block diagram of Resolver activity:

```
From Syntaxer

SynTree

Resolver

Check Data Dictionary Cache

Is Data Dictionary Information In Cache? 

N

Data Dictionary

Y

ResTree

To Request Cache Peek
```
Resolver Component Processes

1. The Resolver takes the SynTree as its input from the Syntaxer.

2. 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.

3. The Resolver refers to the data dictionary to verify all names and access rights and to convert those names to their equivalent numeric IDs.

4. 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.
   
   Note that the Resolver spools all views unconditionally.

5. If a Request parcel contains views or macros, 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.

6. The Resolver produces the ResTree as its output and passes it first to Security and then to Request Cache Peek.
   
   At this point, the system peeks at any parameterized values or DATE or CURRENT_DATE built-in function values that can be evaluated for specific optimization opportunities. See “Peeking at the Request Cache” on page 36 for details.

7. End of process.

SQL Flagger Activity

Some additional SQL Flagger 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, the system validates user logon security rights for every request processed, including those that have been cached in the Request Cache. Database object access rights for cached requests are not revalidated when requests are processed from the Request Cache.

When this process completes, the Parser passes the ResTree to the Request Cache Peek subsystem.

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

- *Security Administration*
- *Database Administration*
- *Data Dictionary*
Peeking at the Request Cache

Introduction

The system has the ability to peek at the parameterized values of a request, and then to resolve those variables in the Resolver. This enables the Optimizer to generate a specific, uncached plan for such requests when the system determines that to be the best way to handle the request rather than always generating a cached generic plan. Request Cache Peek also resolves DATE and CURRENT_DATE values for all cacheable requests, including those for which a specific query plan is generated.

Terminology

This topic introduces some of the terminology that is specific to the Request Cache Peeking subsystem.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
</table>
| Specific Plan   | An Optimizer plan generated by peeking at parameterized values that applies only to a single instance of that query.  
                  | The values obtained from peeking at a parameterized request can then be used to tailor a very specific query plan for that request.  
                  | Specific query plans apply only to parameterized-dependent requests.                                                                      |
| Generic Plan    | An Optimizer plan generated without peeking at parameterized values that applies to most, if not all, instances of that query.          
                  | Generic query plans apply only to parameterized-independent requests.                                                                        |

The presence of parameterized data in a request does not automatically determine that the Optimizer will generate a specific request plan for it. There are many parameterized queries for which the query plan generated by the Optimizer does not depend on parameterized values. Such queries do not gain any added performance benefit from generating specific query plans as the result of Request Cache peeking. Furthermore, the favorable impact of not caching a plan can be high if a request has a short execution time.

For example, consider the following query that specifies a USING request modifier whose single variable \( x \) is also used to specify a parameterized predicate condition \( x \) in its WHERE clause.

```sql
USING (x INTEGER)
SELECT *
FROM table_1
WHERE column_1 = :x;
```

If there is a UPI defined on `table_1.column_1`, then the access path selected by the Optimizer is independent of the value of \( x \) because irrespective of its value, assuming that value specifies a valid primary index, only one access path is possible.

5. Such as USING request modifier values, which are user-supplied constant data (see “USING Request Modifier” in SQL Data Manipulation Language for information about the USING request modifier).
Date-based Request Cache peeking, which resolves the current date from DATE or CURRENT_DATE built-in functions when either of those functions is specified in the request, can be used to generate either specific or generic query plans, depending on several other factors that might be present in the query text. The following table explains the terminology used for these date-specific query plans.

<table>
<thead>
<tr>
<th>FOR this type of parameterized request with a resolved DATE or CURRENT_DATE value and for which this type of query plan is generated ...</th>
<th>THE plan is referred to as a ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic</td>
<td>DateSpecific generic plan.</td>
</tr>
<tr>
<td>specific</td>
<td>DateSpecific specific plan.</td>
</tr>
</tbody>
</table>

If your system is enabled with Teradata Active System Management category 3, and a request belongs to a workload defined with an enforcement priority level of tactical, then it is considered to be a HiPriority request (see Teradata Dynamic Workload Manager User Guide for details). Tactical queries are defined as requests whose per-AMP CPU time is less than or equal to one second.

In the context of the Request Cache Peek subsystem, the following definitions apply:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized PK statement</td>
<td>A single SQL statement with a set of parameterized variables that are used in equality predicate conditions on an NPPI, PPI, or USI of a single table with no ORed conditions. The term PK, which commonly represents Primary Key, in this case represents a primary or unique secondary index column set.</td>
</tr>
</tbody>
</table>
| Parameterized PK request | A request that contains only the following:  
  • Parameterized PK statements  
  • Null statements  
  See SQL Data Manipulation Language for a definition of null statements. |

Request Cache peeking does not impact the treatment of parameterized PK requests, so the Optimizer does not generate a specific plan for those cases.
The following table indicates whether various types of parameterized request are parameterized-dependent or parameterized-independent:

<table>
<thead>
<tr>
<th>The Parser considers this type of parameterized request...</th>
<th>To be...</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple parameterized</td>
<td>parameterized-independent.</td>
<td>The request consists of PK statements and other statements without specifying parameterized variables in any of its predicates. For example, consider the following table definition: CREATE TABLE emp ( emp_id INTEGER, dept_id INTEGER) UNIQUE PRIMARY INDEX (emp_id); The following multistatement parameterized request is parameterized-independent, even though the second statement in the request is a parameterized PK statement, because the x variable is specified on a UPI, so the access path is independent of the value of x, which means that it is also an exempt request (see “Exempt Requests” on page 28): USING (x INTEGER) SELECT * FROM emp WHERE dept_id = 10 ;SELECT * FROM emp WHERE emp_id = :x AND dept_id = 12;</td>
</tr>
<tr>
<td>parameterized PK</td>
<td>parameterized-independent.</td>
<td>By definition.</td>
</tr>
<tr>
<td>iterated</td>
<td>parameterized-independent.</td>
<td>By definition.</td>
</tr>
<tr>
<td>all other parameterized requests</td>
<td>parameterized-dependent.</td>
<td>By definition.</td>
</tr>
</tbody>
</table>

**About Request Cache Peeking**

The word *peeking* means looking at the parameterized values from a Data parcel and evaluating date-based built-in function constant values during query parsing, then using those values to investigate all potential optimization opportunities such as:

- Satisfiability and transitive closure (see “Satisfiability and Transitive Closure” on page 102)
- Optimum single table access planning
- Partition elimination (see “Partition Elimination” on page 298)
- Using covering secondary, hash, and join indexes in place of base tables (see “Query Rewrite” on page 74)
Peeking facilitates the optimization of certain categories of queries by inserting data values that are specific to those queries into the parse tree at an earlier stage of request processing than would otherwise be done. The system always caches generic query plans because reusing a cached query plan saves parsing and optimization time. However, reusing a cached generic query plan is not the best approach to executing a query that provides constant literals that can be used to produce a specific query plan tailored closely to that specific request.

The system does not cache specific query plans for requests that are parameterized-dependent (see “Terminology” on page 36 for the definition of parameterized-dependent and parameterized-independent requests), because those plans cannot be reused for otherwise identical queries that have different sets of parameterized or built-in function-supplied constant values, but it does cache the other information generated for specific query plans. This information includes its SQL text hash, its host character set, its estimated execution costs, its parsing time, and its run time. Request Cache Peeking directly impacts only those requests that are not cacheable (see “Request Cache” on page 23).

When the system determines that a query plan should not be cached, the parameterized request and DATE or CURRENT_DATE built-in function values can be used to optimize the request instead of treating that data as parameters with unknown values. In other words, Request Cache Peeking uses the literal data values from a parameterized request and date-related built-in functions, or both, to ensure that the Optimizer generates an optimal, uncached plan for that particular query rather than generating a generic plan and then caching it for future reuse.

A specific query plan generated by the Optimizer should be an optimal plan, and its runtime performance should be significantly better than the execution time for an equivalent generic query plan. However, there can be cases when the runtime costs of a specific plan and its equivalent generic plan do not differ significantly. The impact of this failure to achieve significantly better performance from a specific request plan is especially high for those cases where the parsing cost is high. In such cases, the system must examine the performance implications of regenerating the specific plan every time.

The system monitors all such queries for their parsing and run times and keeps the information in the Request Cache to use when it must decide between generating a specific or generic plan for a request.

The first time a parameterized request is submitted to a PE, determined by not finding the request in the Request Cache, the Request Cache Peek system peeks at the parameterized values for the request and instructs the Optimizer to generate a specific query plan only if the query is not a parameterized PK request. Specific query plans are not cached because they cannot be reused for different parameterized or built-in date function values.

The Optimizer always generates specific plans before it generates generic plans to ensure that queries submitted to the system only once are provided an optimal request plan.

If the parsing cost of a specific plan is a fraction of its execution time, then all subsequent requests execute specific plans (see “Request Cache Peek Component Process” on page 45 for details). This is because if the overhead of parsing is small, the impact of not caching the request is insignificant.
Otherwise, when the system encounters a query for the second time (determined by interrogating the Request Cache), the Optimizer generates a generic plan for it. If the estimated cost of a previously executed specific plan is significantly less than the estimated cost of the equivalent generic plan, then the system reparses the request and instructs the Optimizer to generate and execute a specific plan for it. (see “Request Cache Peek Component Process” on page 45 for details).

This avoids executing a bad *generic* plan for the cases where it is possible to generate a good *specific* plan. The system compares the plan cost estimates only if the CompareEstimates flag in the cost profile is enabled (CompareEstimates is *disabled* by default). Otherwise, the system executes the generic plan, measures its parsing and run times, and caches them in the Request Cache.

The decision to cache a generic plan is based on the respective CPU run times of the specific and generic versions of the plan. The performance of some parameterized queries that are submitted only twice during the unspoiled lifetime of the Request Cache might be negatively affected by this because the request must be parsed on both its first and second submissions.

For tactical queries, the system caches the generic plan and always uses it (see “Request Cache Peek Component Process” on page 45 for details).

For HiPriority queries, the system compares elapsed times instead of CPU times to determine whether the generic plan should be cached or not (see “Request Cache Peek Component Process” on page 45 for details).

See “Block Diagram of Request Cache Peek Activity” on page 41 and “Request Cache Peek Component Process” on page 45 for details.

### Enabling and Disabling Request Cache Peeking

You can control the availability of Request Cache Peeking on your system by setting the DisablePeekUsing performance flag in the DBS Control record (see *Utilities* for details).

<table>
<thead>
<tr>
<th>IF the value of DisablePeekUsing is set to ...</th>
<th>THEN Request Cache Peeking is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>enabled for your system.</td>
</tr>
<tr>
<td></td>
<td>This is the default.</td>
</tr>
<tr>
<td>TRUE</td>
<td>disabled for your system.</td>
</tr>
</tbody>
</table>

The setting for DisablePeekUsing does *not* control whether the system caches requests that contain constant literal data obtained from date-based built-in functions only, without specifying a parameterized request. In this case, the Optimizer always generates and caches generic plans for queries that specify the built-in functions DATE or CURRENT_DATE, but when the date value changes, the system flushes any cached plans with dependencies on that date value, and the Optimizer generates a new plan using the new date value.
Block Diagram of Request Cache Peek Activity

The following graphic shows a block diagram of Request Cache Peek activity:

1. **Request Parcel**
   - **Check Request Cache**
   - **Format Errors?**
     - **N**
     - **Y**
   - **Is Parcel In Cache?**
     - **N**
     - **Y**
       - Instruct the Optimizer to Use the Generic Plan
       - **Is Generic Plan > Threshold for Specific Plan?**
         - **N**
         - **Y**
           - **Instruct the Optimizer to Generate and Execute a Generic Plan**
           - **To OptApply**
           - **Is Cached Plan Marked as Specific Always?**
             - **Y**
             - **To OptApply**
             - **N**
               - **To Query Rewrite**
               - **To Application**

   - **Instruct the Optimizer to Generate and Execute a Specific Plan**
   - **Capture and Cache Parsing Time and CPU Run Time**
   - **Mark Cache Entry as Specific Always**
   - **Reparse Request and Instruct the Optimizer to Generate and Execute a Specific Plan**

2. **To Application**
Chapter 1: Request Parsing
Peeking at the Request Cache

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
</tr>
<tr>
<td>Is This a High Priority Request With (Parse Time for Parsing Specific Plan and Elapsed AMP Time For Specific Plan) &gt; Threshold?</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Always Use Cached Generic Plan</td>
</tr>
<tr>
<td>To OptApply</td>
</tr>
<tr>
<td>N</td>
</tr>
<tr>
<td>Is Single AMP CPU Time for Generic Plan &lt; Threshold?</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Always Use Cached Generic Plan</td>
</tr>
<tr>
<td>To OptApply</td>
</tr>
<tr>
<td>N</td>
</tr>
<tr>
<td>Is CPU Path Time for Parsing Request With Specific Plan &gt; Threshold?</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Is Total All-AMP CPU Time &gt;= Threshold?</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Always Use Cached Specific Plan</td>
</tr>
<tr>
<td>To OptApply</td>
</tr>
<tr>
<td>N</td>
</tr>
<tr>
<td>Always Use Cached Generic Plan</td>
</tr>
<tr>
<td>To OptApply</td>
</tr>
<tr>
<td>Always Use Cached Specific Plan</td>
</tr>
<tr>
<td>To OptApply</td>
</tr>
</tbody>
</table>

| 1101A502 |
Request Cache Peek Component Process Terminology

The following table defines the terms used in the stages of the Request Cache Peek component processes:

<table>
<thead>
<tr>
<th>Key to Terms Used in the Request Cache Peek Process Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Factors</td>
</tr>
<tr>
<td>The following algorithmic factors are measured by the system.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pset</td>
<td>Elapsed time for parsing a request for a specific plan. Elapsed time is defined as the clock time a request takes to complete its execution.</td>
</tr>
<tr>
<td>Rset</td>
<td>AMP elapsed time for the specific plan.</td>
</tr>
<tr>
<td>Rget</td>
<td>AMP elapsed time for the generic plan.</td>
</tr>
<tr>
<td>Pscpu</td>
<td>CPU parsing time for a request using the specific plan. CPU parsing time is defined as the CPU time taken to parse a request, generate a query plan for it, and pass that plan to the Dispatcher.</td>
</tr>
<tr>
<td>Rscpu</td>
<td>Total all-AMPs CPU time for the specific plan.</td>
</tr>
<tr>
<td>Rgcpu</td>
<td>Total all-AMPs CPU time for the generic plan.</td>
</tr>
<tr>
<td>PerAMPRgcpu</td>
<td>CPU time per single AMP for the generic plan.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cost Profile Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following algorithmic factors are defined in your Cost Profile.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TacticalResp1</td>
<td>Runtime CPU cost per AMP that determines whether a query is treated as a tactical or decision support request. The value is derived from the CPU time used for query execution in the AMPs. By definition, a request whose per-AMP CPU time ≤ 1 second is considered to be tactical and a request whose per-AMP CPU time &gt; 1 second is considered to be a decision support request.</td>
</tr>
<tr>
<td>TacticalResp2</td>
<td>Runtime CPU cost per AMP that determines whether a query is treated as a tactical or decision support query when it is submitted as a HiPriority request. The value is derived from the CPU time used for query execution in the AMPs.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HighParsingPTThreshold</td>
<td>Threshold percentage of the parsing cost compared to the runtime CPU cost on which a determination is made on whether the request has a high parsing cost.</td>
</tr>
<tr>
<td>HighParsingRTThreshold</td>
<td>Threshold multiplication factor for the determination of runtime benefits for a query that has a high parsing cost. If the request has a high parsing cost, then the runtime CPU times of specific and generic plans should differ by at least this value multiplied times the specific CPU parsing time.</td>
</tr>
<tr>
<td>HighParsingPTThreshold</td>
<td>Threshold multiplication factor for the determination of runtime benefits for a query that has a low parsing cost. If the request has a low parsing cost, then the runtime CPU times of specific and generic plans should differ by at least this value multiplied times the specific CPU parsing time.</td>
</tr>
<tr>
<td>UseHiPriority</td>
<td>A flag that enables or disables the HiPriority-based decisions in the caching algorithm.</td>
</tr>
<tr>
<td>ElapsedTimeThreshold</td>
<td>Threshold multiplication factor by which the elapsed time of a specific plan execution (the sum of its parsing and run times) should exceed the elapsed time of the equivalent generic plan.</td>
</tr>
<tr>
<td>EstimateCostFilter</td>
<td>Threshold factor by which the estimated cost of a specific plan should be better than the estimated cost of the equivalent generic plan. Used for comparison of estimated costs.</td>
</tr>
<tr>
<td>CompareEstimates</td>
<td>A Cost Profile flag (see “Cost Optimization” on page 216) that enables or disables the estimate-based comparisons for deciding between using the generic and specific plan for a request. This flag is disabled by default. Use this flag to tune those cases where the system executes a bad generic plan the second time a cached query is encountered.</td>
</tr>
</tbody>
</table>
Request Cache Peek Component Process

The following process outlines the stages the Request Cache Peek subsystem follows to determine whether a generic or specific plan should be used and cached for a given request.

<table>
<thead>
<tr>
<th>Stage</th>
<th>IF ...</th>
<th>THEN the request is processed as ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>• the request is submitted for the first time AND • the specific plan projects that the parsing time is less than one percent of the runtime</td>
<td>Specific Always&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>2</td>
<td>• the request that was submitted in Stage 1 is submitted a second time AND • the CompareEstimates cost profile flag is set AND • the generic plan estimate &gt; (EstimateCostFilter cost profile flag value * specific plan estimate)</td>
<td>Specific Always</td>
</tr>
<tr>
<td>3</td>
<td>• the request is HiPriority AND • the UseHiPri cost profile flag is set AND • the total elapsed time for the specific plan &gt; (ElapsedTimeThreshold cost profile flag value * elapsed runtime of the generic plan)</td>
<td>Generic Always&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>4</td>
<td>PerAMPRgcpu ≤ (TacticalResp1 value * 1,000)</td>
<td>Generic Always</td>
</tr>
<tr>
<td>5</td>
<td>• the request is HiPriority AND • PerAMPRgcpu ≤ (TacticalResp2 value * 1,000)</td>
<td>Generic Always</td>
</tr>
<tr>
<td>6</td>
<td>• the CPU path time for parsing the specific plan &gt; (HighParsingPTThreshold cost profile flag value * (specific plan parsing time + runtime)) AND •</td>
<td>Specific Always</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- generic plan runtime - specific plan runtime ≥ (HighParsingRTThreshold cost profile flag value * specific plan parsing time)</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>generic runtime - specific runtime ≥ (LowParsingRTThreshold cost profile flag value * specific plan parsing time)</td>
</tr>
<tr>
<td>8</td>
<td>anything else</td>
<td>Generic Always</td>
</tr>
</tbody>
</table>

<sup>a</sup> Specific Always means that parameterized values are resolved and evaluated in the query plan for all subsequent requests.

<sup>b</sup> Generic Always means that parameterized values are not resolved until the concrete steps for the request are generated, and the cached generic plan is used for all subsequent requests.
Replacing Parameterized Variables

The system does not replace parameterized variables for parameterized PK requests because they have a very short run time. The intent of eliminating PK requests from consideration is that it is known a priori that such queries have a short run time, and therefore caching such plans is beneficial. Additionally, PK requests are USING-independent by definition.

Other request types, such as few-AMP requests, typically have a short run time and should benefit from caching as a general rule. However, it is not possible to determine whether such requests qualify for caching without first optimizing them, so they are not excluded from the Request Cache peek logic.

If the cost of a specific plan is better than the runtime cost of a generic plan as determined by their respective parsing and runtime costs, or if the estimate of a specific plan indicates significant cost savings over the estimate for the equivalent generic plan, then the system generates a specific plan for subsequent requests.

If a request is not a parameterized PK request, then the system replaces USING variables in all the conditions in the request, including the select list of a subquery or derived table.

The system uses the following rule to decide which parameterized request variables are replaced in the request:

<table>
<thead>
<tr>
<th>IF a request is this type ...</th>
<th>THEN replace its USING variables in all places except ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>the outermost SELECT list.</td>
</tr>
<tr>
<td></td>
<td>This applies for SELECT requests that specify SET operators such as GROUP BY, UNION, INTERSECT, and MINUS.</td>
</tr>
<tr>
<td>anything else</td>
<td>when it occurs in a parameter assignment list.</td>
</tr>
</tbody>
</table>

Resolving the DATE or CURRENT_DATE Value for Optimization

The system resolves the DATE or CURRENT_DATE built-in function value for all queries, whether parameterized or not, and replaces the function with the actual date before the request reaches the Optimizer. This helps to generate a more optimal plan in cases of partition elimination, sparse join indexes, and NUSIs that are based on DATE or CURRENT_DATE.

The rule to replace DATE or CURRENT_DATE in the resolved parse tree is the same as is followed for replacing parameterized request variables (see “Replacing Parameterized Variables” on page 46).

The DATE or CURRENT_DATE value is resolved in the Resolver phase and replaced with the value of the actual date. The SQL text otherwise remains the same. The system also notes the timestamp and uses it in OptApply instead of getting a new timestamp to build the USING row, for example. This synchronizes the timestamps for DATE, CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP.

If the system finds a matching entry in the Request Cache pertaining to a previous date, the entry is marked for purging and the plan is regenerated as if the request is seen for the first time.
The EXPLAIN text for queries having a resolved DATE or CURRENT_DATE value shows the resolved date. For example, assuming the following EXPLAIN SELECT request was submitted on February 2, 2007, the EXPLAIN text looks like the following example, with the relevant phrases highlighted in boldface type:

```
EXPLAIN SELECT *
FROM es
WHERE process_dt = CURRENT_DATE;
```

**Explanation**

---

1) First, we lock a distinct GL."pseudo table" for read on a RowHash to prevent global deadlock for GL.es.
2) Next, we lock GL.es for read.
3) We do an all-AMPs RETRIEVE step from a single partition of GL.es with a condition of ("GL.es.process_dt = DATE '2007-02-20'") with a residual condition of ("GL.es.process_dt = DATE '2007-02-20'") into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 1 row. The estimated time for this step is 0.03 seconds.
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 0.03 seconds.

**Miscellaneous Considerations for Request Cache Peeking**

The items in the following list also affect Request Cache peeking:

- If a USING request modifier is being explained, or submitted with an INSERT EXPLAIN or DUMP EXPLAIN request, the system peeks at the parameterized variables if a Data parcel is also provided in the request.
  
  If no Data parcel is submitted, then the system does not peek at the parameterized variables, and the system generates a generic plan for the request.

- If a parameterized request is submitted under a MultiLoad or FastLoad sessions, the system does not peek at the parameterized variables and current date, so there is no impact on request caching.

- If a parameterized request is submitted under a FastExport session, the system peeks at the parameterized variables and current date for a SELECT request in order to generate a more optimal plan for exporting the data.
  
  Note that requests submitted in a FastExport session are not cached, so peeking at parameterized request variable values does not impact caching behavior.

- If the Parser does not peek at the value of any of the parameterized request variables of a parameterized request as, for example, when a USING variable is in the outermost select list and is not specified in the WHERE condition, then there is no impact on caching.
Usage Considerations

The Request Cache stores the query plans of successfully parsed SQL DML requests so they can be reused when the same request is resubmitted. The Request Cache contains the text of the SQL request and its plastic steps. The plastic steps are referred to as the request plan.

If Request Cache peeking is not enabled, when a parameterized request is seen and successfully parsed, it is cached immediately. Request Cache peeking changes the behavior of the Request Cache for requests submitted in CLIv2 Execute or Both Prepare and Execute modes, as follows:

- If a request specifies a DATE or CURRENT_DATE built-in function, then the request plan is generated for the specific date when the request is parsed. The specific date, when exposed during the optimization process, causes the Optimizer to generate a more optimal plan where applicable.

For example, if the partitioning expression is date-based, and the condition on the partition column in the predicate specifies the CURRENT_DATE built-in function, then Optimizer performs partition elimination, generating a more optimal plan.

Assume that table t4 is partitioned on the date column d.

```sql
EXPLAIN SELECT *
FROM t4
WHERE d BETWEEN DATE '2005-02-01' AND CURRENT_DATE;
```

*** Help information returned. 12 rows.
*** Total elapsed time was 1 second.

The following is the EXPLAIN text output when the date is not replaced:

```
Explanation
------------------------------
1) First, we lock a distinct MYDB."pseudo table" for read on a RowHash to prevent global deadlock for MYDB.t4.
2) Next, we lock MYDB.t4 for read.
3) We do an all-AMPs RETRIEVE step from 23 partitions of MYDB.t4 with a condition of ("(MYDB.t4.d <= DATE) AND (MYDB.t4.d >= DATE '2005-02-01')") into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.09 seconds.
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 0.09 seconds.
```

Note that the system must scan 23 partitions to process this query without peeking at the value of CURRENT_DATE (the relevant text is highlighted in boldface type).

If Request Cache peeking is enabled, the following is the EXPLAIN text output for the same request. Notice the partition elimination in the EXPLAIN output that indicates 9 fewer partitions to scan due to Request Cache peeking (the relevant text is highlighted in boldface type).

```sql
EXPLAIN SELECT *
FROM t4
WHERE d BETWEEN DATE '2005-02-01' AND CURRENT_DATE;
```

*** Help information returned. 13 rows.
*** Total elapsed time was 1 second.
Explanation

1) First, we lock a distinct MYDB."pseudo table" for read on a RowHash to prevent global deadlock for MYDB.t4.
2) Next, we lock MYDB.t4 for read.
3) We do an all-AMPS RETRIEVE step from 14 partitions of MYDB.t4 with a condition of "(MYDB.t4.d <= DATE '2006-03-02') AND (MYDB.t4.d >= DATE '2005-02-01')" into Spool 1 (group_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.08 seconds.
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 0.08 seconds.

Date-based request plans are stored in the Request Cache along with the date. The system uses this information to process a resubmitted request as described in the following table.

The system first recomputes the date.

<table>
<thead>
<tr>
<th>IF the recomputed date ...</th>
<th>THEN the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>matches the cached date</td>
<td>cached plan is reused.</td>
</tr>
<tr>
<td>does not match the cached date</td>
<td>cached plan is purged and a new plan is generated for the changed date. The new plan is then cached.</td>
</tr>
</tbody>
</table>

• When Request Cache peeking is enabled, requests are cached or not according to the actions described by the following table.

<table>
<thead>
<tr>
<th>IF a parameterized DML request is ...</th>
<th>THEN it is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameterized-independent</td>
<td>cached immediately.</td>
</tr>
<tr>
<td>parameterized-dependent</td>
<td>parsed with the peeked parameterized values exposed, and the Optimizer then generates a request plan that is specific to those parameterized values. In this case, the request plan is not cached, but its request text is. When the same parameterized-dependent request is resubmitted, the Optimizer generates a generic plan for it.</td>
</tr>
</tbody>
</table>

The generic plan is executed if either of the following statements is true:

• The estimates are not compared
• The estimate for the specific plan does not indicate any benefit when compared to the estimate for the generic plan.

Otherwise, the system reparses the request and generates a specific plan for it.
This means that a generic plan, if executed, is always cached for subsequent reuse if either of the following statements is true:

- Its runtime CPU cost is very small
- Execution of the specific plan does not provide enough runtime CPU cost benefit to compensate for the parsing CPU cost.

In all other cases, the Optimizer generates the specific plan for all requests until the cache is purged. This includes case when executing the generic plan fails, such as when it aborts or returns a fatal error during execution.

A performance benefit is expected to be seen when the system decides to generate specific plans for all subsequent requests.

You can submit a MONITOR SESSION request to determine the number of Request Cache hits in a session. You can then use that information to determine whether a generic plan has been cached and used.

A specific plan is generally more optimal than its equivalent generic plan because its parameterized values have been substituted as literals. There are, however, some queries for which the specific plan cost is the same as that for the equivalent generic plan. The logic that replaces the parameterized values to generate specific plan cannot distinguish between queries for which the estimated specific and generic plan times are identical.

The request plans for the following two queries are identical, and peeking at parameterized values cannot help to generate more optimal plans for them. But queries such as the previous SELECT example can benefit from Request Cache peeking if a join index like the following is defined on the target base table for the query.

For example, consider the following table definition:

```sql
CREATE SET TABLE MYDB.t2, NO FALLBACK, NO BEFORE JOURNAL, NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  i INTEGER,
  j INTEGER,
  k INTEGER,
  l INTEGER,
  c CHARACTER(400) CHARACTER SET LATIN NOT CASESPECIFIC DEFAULT 'a')
PRIMARY INDEX (i)
INDEX (j);
```

The statistics collected on the table show the following number of unique values for the NUSI column `j` and the NUPI column `i`:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Unique Values</th>
<th>Column Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/02/17</td>
<td>12:51:22</td>
<td>1,537</td>
<td>j</td>
</tr>
<tr>
<td>06/02/17</td>
<td>12:56:10</td>
<td>60</td>
<td>k</td>
</tr>
</tbody>
</table>
Now consider the following query against table \( t_2 \):

```sql
EXPLAIN USING (a INTEGER) SELECT *
FROM t2
WHERE j = 58
AND k = 100000+i;
```

*** Help information returned. 14 rows.
*** Total elapsed time was 1 second.

Explanation

1) First, we lock a distinct MYDB."pseudo table" for read on a RowHash to prevent global deadlock for MYDB.t2.
2) Next, we lock MYDB.t2 for read.
3) We do an all-AMPs RETRIEVE step from MYDB.t2 by way of an all-rows scan with a condition of \("(MYDB.t2.k = (100000 + MYDB.t2.i)) AND (MYDB.t2.j = 58)"\) into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 10 rows. The estimated time for this step is 0.17 seconds.
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 0.17 seconds.

And consider the following query, which is also made against table \( t_2 \):

```sql
EXPLAIN USING (a INTEGER) SELECT *
FROM t2
WHERE j = 58
AND k =:a + i;
```

*** Help information returned. 14 rows.
*** Total elapsed time was 1 second.

Explanation

1) First, we lock a distinct MYDB."pseudo table" for read on a RowHash to prevent global deadlock for MYDB.t2.
2) Next, we lock MYDB.t2 for read.
3) We do an all-AMPs RETRIEVE step from MYDB.t2 by way of an all-rows scan with a condition of \("(MYDB.t2.k = (:a + MYDB.t2.i)) AND (MYDB.t2.j = 58)"\) into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 10 rows. The estimated time for this step is 0.17 seconds.
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 0.17 seconds.

Now create table \( t_3 \) as a copy of table \( t_2 \) with data and consider the following join index definition:

```sql
CREATE JOIN INDEX j1 AS
SELECT *
FROM t3
WHERE j > 58
AND k > i+3;
```
Consider the following query and assume the USING value for \(a1\) is 4:

```sql
EXPLAIN USING (a1 int) SELECT *
FROM t3
WHERE j = 80
AND k = i+:a1;
```

*** Help information returned. 14 rows.
*** Total elapsed time was 1 second.

You can see that the value 4 has been explicitly inserted in the text for Step 3, where it is highlighted in boldface type.

**Explanation**

1) First, we lock a distinct MYDB."pseudo table" for read on a RowHash to prevent global deadlock for MYDB.J1.
2) Next, we lock MYDB.J1 for read.
3) We do an all-AMPs RETRIEVE step from MYDB.J1 by way of an all-rows scan with a condition of "(MYDB.J1.k = (MYDB.J1.i + 4)) AND (MYDB.J1.j = 80)" into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 126 rows. The estimated time for this step is 0.14 seconds.
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 0.14 seconds.

The following performance benefits are realized from Request Cache peeking:

- The system generates specific plans for all requests if either of the following conditions are true:
  - The system compares the plan estimates and the estimated cost of the specific plan is less than the estimated cost of the generic plan.
  - Both the parsing and runtime CPU costs of the specific plan is smaller than the runtime CPU cost of the equivalent generic plan.
- The values for the DATE or CURRENT_DATE built-in functions are expanded and passed to the Optimizer for optimal plan generation. This is done even if the request does not specify parameterized data.
- If a specific plan does not provide a performance benefit, the system instead caches the equivalent generic plan and reuses it for subsequent requests.

The following potential performance liabilities are also realized from Request Cache peeking:

- In some cases, estimates of generic and specific plans are compared. If the generic plan is worse than the equivalent specific plan, the system reparses the request and generates a specific plan for it.
- A request that is classified as parameterized-dependent might not always generate a different specific plan. Such requests are also subject to the logic for plan caching. Ideally, they would be cached immediately; however, they are instead cached only after their second submission to the system. This can have a performance impact for those requests that are submitted only twice.
• If the specific plan and the generic plan are identical, and there are no potential benefits seen for the specific plan, then the generic plan is used for all subsequent requests. As a result, no performance benefit can be realized even if parameterized data values in subsequent requests are such that they would result in more optimal specific plans if they were exposed and inserted into the plan as literal data values. However, once the Request Cache is purged, the system reevaluates all affected specific and generic request plans.

• In some cases, the decision to cache a generic plan is based on the elapsed execution times of the specific and generic plans. The decision can be influenced wrongly because of other workloads running on the system that might block either of the requests being compared, and as a result skew their elapsed time values.

• In some cases, the algorithm that compares specific and generic costs uses estimates. A bad estimate for either plan can influence the caching decision.
Query Rewrite

Introduction

Query Rewrite is the stage of query processing that occurs just prior to query optimization. Query Rewrite involves applying the following processes to the parse tree wherever possible:

- Substituting underlying base tables and covering indexes for views and derived tables (a method called View Folding).
- Substituting hash and join indexes for underlying base tables, views, or join operations.
- Converting OUTER JOINs to INNER JOINs.
- Eliminating unnecessary joins altogether.
- Using logical satisfiability and transitive closure either to eliminate terms or to add terms that facilitate further rewrites.
- Simplifying predicates.
- Pushing projections and predicate conditions into spooled views.
- Eliminating set operation branches.
- Pushing joins into UNION ALL views.

The basic task of the Query Rewrite subsystem is to determine how query text can be rewritten to make it more efficient and more easily optimized without changing its semantics. In other words, Query Rewrite is the process of rewriting a query $Q$ in a new form, query $Q'$, such that both of the following statements are true:

- Both queries produce the identical result.
- $Q'$ runs faster than $Q$.

Query Rewrite techniques can be rule-based, such as Predicate Pushdown, or cost-based, such as rewriting a query to use a join index. Note that not all query rewrites are contained with the Query Rewrite subsystem. Some query rewrites, for example, can be applied in the query optimization phase of request parsing during join planning, or even after join planning is complete. For example, Partial GROUP BY rewrites are done during join planning, and Common Spool Usage is a rewrite that is applied after join planning is done. This topic describes only those rewrites that are applied by the Query Rewrite subsystem.

The Query Rewrite subsystem is isolated from other optimization subsystems to prevent undesirable interactions. The Query Rewrite subsystem contains a Rewrite Driver (see “Query Rewrite Driver” on page 86), which understands and takes advantage of the interactions among various query rewrites to transform requests into better performing variants of themselves.

The diagram on the following page shows the order in which the various Query Rewrite techniques are applied. Note that after the Pushing Joins Into UNION ALL Views stage, the Query Rewrite process loops back to the stage at which outer joins are converted to inner joins. This is because some of the later rewrites in the first round of the process often create new opportunities for additional rewrites that can be exploited to make $Q'$ run faster still.

6. Note that many query rewrites are done by the Optimizer, not by the Query Rewrite subsystem.
Chapter 1: Request Parsing
Query Rewrite

Block Diagram of Query Rewrite Activity

From Request Cache Peek

ResTree

Push Projections Into Spooled Views

Convert Outer Joins To Inner Joins

Fold Views

Satisfiability and Transitive Closure

Push Conditions Into Spooled Views

Eliminate Joins

Eliminate Set Operation Branches

Push Joins Into UNION ALL Views

ResTree'

To Optimizer
Query Rewrite Component Processes

1. Query Rewrite accepts the ResTree from the Resolver as input.
   The goal of Query Rewrite is to rewrite the SQL text in such a way that it produces a new
   ResTree that is semantically identical to the ResTree passed to it from the Resolver, but is
   revised in such a way that it will run faster that the original SQL text would have once it is
   optimized.

2. When possible, Query Rewrite pushes projections into spooled views.
   This means that any columns projected by the original query that do not contribute to its
   final outcome are dropped from the ResTree.
   If it cannot push the columns projected by the query into spooled views, or after all
   possible column projections have been pushed into spooled views, Query Rewrite passes
   the query to the outer join-to-inner join conversion stage of the process.

3. When possible, Query Rewrite rewrites outer joins as inner joins.
   Query Rewrite always converts outer joins to inner joins if the conversion does not
   produce spurious nonmatching rows.
   If it cannot convert outer joins to inner joins, or after all possible outer join-to-inner join
   rewrites have been done, Query Rewrite passes the query to the next stage of the process.

4. When possible, Query Rewrite folds views and derived tables.
   Folding means that any views or derived tables specified in the query are rewritten using
   their underlying base tables and query-covering indexes to make all their references both
   explicit and more optimal.
   There are two types of view folding:
   - Type 1
     Type 1 view folding is defined by exclusion to mean all view folding that is not Type 2
     view folding.
   - Type 2
     Type 2 view folding is used to rewrite the following set of special expressions that are
     not handled by Type 1 view folding:
     - Views and derived tables that have constants or null manipulating CASE
       expressions in their definitions
     and
     - Such views and derived tables are used as inner tables in OUTER JOIN operations
     and
     - The constants or null-manipulating CASE expression inside the view and derived
       table definitions are referenced in the outer query block.
   All other view and derived table folding is handled by Type 1 folding.
   If there are no views or derived tables to be folded, or after all possible view and derived
   table folding has been done, Query Rewrite passes the query to the next stage of the
   process.
When possible, Query Rewrite uses logical satisfiability and transitive closure (SAT-TC) to eliminate terms from the query.

For example, a predicate of `WHERE 1=0` cannot be satisfied, so it can be eliminated from the query.

Similarly, if transitive closure can be employed to eliminate redundant steps toward reaching satisfiability, then those intermediate steps can be eliminated.

Generally speaking, satisfiability is a by-product of transitive closure.

If it cannot eliminate terms using satisfiability and transitive closure, Query Rewrite passes the query to the next stage of the process.

Predicate Simplification is also a component of SAT-TC.

The following methods are the components of Predicate Simplification in Query Rewrite:

- Constant predicate evaluation
- Range-based simplification
- Constant movearound
- Duplicate predicate removal
- NULLEQ marking

Although Constant Predicate Evaluation and NULLEQ Marking are both components of Predicate Simplification conceptually, the system actually performs Constant Predicate Evaluation during the preprocessing phase of Query Rewrite, while it performs NULLEQ Marking during the postprocessing phase of Query Rewrite.

When possible, Query Rewrite pushes conditions into spooled views.

What this means is that predicates can often be moved into another query block, making it possible to eliminate the query block from which they originated and consolidating the number of blocks that must be processed.

Pushing predicate conditions into spooled views helps performance by reducing the size of spooled view files.

If no conditions can be pushed into spooled views, or after all possible conditions have been pushed into spooled views, Query Rewrite passes the query to the next stage of the process.

When possible, Query Rewrite eliminates join operations that do not contribute to the final result of the query.

The simplification made by eliminating joins within a view or the main query can provide more opportunities to fold views during the second iteration of the Query Rewrite process.
Any of the rewrites done in stages 3 through 8 might make it possible to do additional rewrites that had not been possible during the first pass Query Rewrite made on the query; therefore, those stages are repeated a second time to capitalize on any such possibilities that were created by the first pass through the process.

Query Rewrite passes ResTree’, which is semantically identical to the ResTree that entered the Query Rewrite subsystem from the Resolver, to the Optimizer for optimization of the access and join paths for the query.

End of process.

See “Query Rewrite” on page 74 for more details about the Query Rewrite process and subsystem.
Chapter 1: Request Parsing
Optimizer

Introduction

The 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 Rewrite and Optimization.”

- “Query Optimizers” on page 126
- “Optimizer Statistics” on page 140
- “Interval Histograms” on page 161
- “Sampled Statistics” on page 175
- “Random AMP Sampling” on page 178
- “When Should Statistics Be Collected Or Recollected?” on page 190
- “How the Optimizer Uses Statistical Profiles” on page 193
- “Derived Statistics” on page 206
- “Statistical Inheritance” on page 216
- “Stale Statistics” on page 251
- “Cost Optimization” on page 267
- “Environmental Cost Factors” on page 296

Also see Chapter 3: “Join Planning and Optimization” and Chapter 4: “Join Optimizations” for information about how Teradata Database optimizes join requests.

For information about the Teradata Index and Statistics Wizards, which are also a component of the Optimizer, see “Chapter 7 Database Foundations for the Teradata Index and Statistics Wizards” on page 633.
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.

- Should this request be optimized generically or specifically?
- 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.
- Are there interval histogram statistics for the column and index sets required to process the query?
- If there are existing interval histogram statistics, are they fresh enough to provide reasonable cardinality estimates, or are they stale?
- If there are existing interval histogram statistics, can they cover a range query over DATE values, or do they require extrapolation?
- Does the table have a primary index?
- Is the primary index for the table partitioned?
- If the primary index is partitioned, are there PARTITION statistics for it?
- 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 in the column set or index have one or more nulls for the columns on which statistics have been collected?
- How many rows in the column set or index are null for all the columns on which statistics have been collected?
- How many rows per column set value are expected to be returned?
- Can a base table be replaced by a covering secondary, 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 Get Answers for the Questions It Asks?

Ideally, these questions are answered largely based on statistical data that you have generated using the SQL COLLECT STATISTICS statement (see “COLLECT STATISTICS (Optimizer Form)” in SQL Data Definition Language). When database statistics are collected regularly, then you can expect the Optimizer to make the best decisions possible.

If statistics have been collected, but long enough ago that they no longer reflect the true demographics of the data, then the Optimizer might not be able to make the best-informed decisions about how to proceed (see “Time and Resource Consumption As Factors In Deciding How To Collect Statistics” on page 148, and “An Example of How Stale Statistics Can Produce a Poor Query Plan” on page 150), and “Stale Statistics” on page 251.

If no statistics have been collected on indexed columns in a request, 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 (see “Random AMP Sampling” on page 178). Any derived statistics that the Optimizer develops begin with the random AMP index sample as a basis for deriving additional statistics. Note that the Optimizer does not use random AMP samples of nonindexed columns to make cardinality estimates.

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 they generate 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 absolutely 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 ResTree\(^7\) 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 original SQL code is written.
  There are limits to the capability of query rewrite (see “Query Rewrite” on page 74) to increase the efficiency of a given user-written query.

7. Also known as the Red Tree.
Block Diagram of Optimizer Processes

A block diagram of Optimizer activity is shown in the following illustration and explained in the list of processing stages that follows in “Optimizer Component Processes” on page 63:
Optimizer Component Processes

1 The Optimizer examines an incoming Request parcel to determine if the request it is about to optimize is a DML statement or a DDL/DCL statement.

<table>
<thead>
<tr>
<th>IF the Request parcel contains this type of SQL statement ...</th>
<th>THEN the Optimizer ...</th>
</tr>
</thead>
<tbody>
<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 plans, join plans, and execution plans. The Optimizer then uses whatever statistical information it has, whether complete or sampled, to determine which access paths or plans are to be used. If there are no column or index statistics in the data dictionary, then the Optimizer uses dynamic random AMP sampling to estimate the population statistics of the data.</td>
</tr>
</tbody>
</table>

2 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.

3 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 available derived statistics and costing data it has to work with.

4 The Optimizer places, combines, and reorders locks to reduce the likelihood of deadlocks, then removes any duplicate locks it finds.

5 Finally, the Optimizer either passes its fully optimized parse tree, known as the White Tree, on to the Generator for further processing or, if has optimized an EXPLAIN request, produces a verbal explanation of the White Tree to which additional spool size information and costing data that would not otherwise be costed, is added both for human analysis and for the benefit of the Teradata Workload Manager software.

6 End of process.

Definition: Steps

A step is a 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.

You can find more information about plastic steps in “Definition: Plastic Steps” on page 67.
You can find more information about concrete steps in “Definition: Concrete Steps” on page 69.
**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.

```sql
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:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Process</th>
<th>Processing Mode</th>
<th>Step Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The system locks both tables (<code>employee_number</code> and <code>call_emp</code>).</td>
<td>Serial</td>
<td>Common</td>
</tr>
<tr>
<td>2</td>
<td>The system copies the rows from the <code>employee_number</code> table and redistributes them. &lt;br&gt;The system copies the rows from the <code>call_emp</code> table and redistributes them.</td>
<td>Parallel</td>
<td>Common</td>
</tr>
<tr>
<td>3</td>
<td>The system Merge Joins the results.</td>
<td>Serial</td>
<td>Individual</td>
</tr>
<tr>
<td>4</td>
<td>The system Exclusion Merge Joins the results.</td>
<td>Serial</td>
<td>Individual</td>
</tr>
<tr>
<td>5</td>
<td>The system releases the table-level locks on <code>employee_number</code> and <code>call_emp</code>.</td>
<td>Serial</td>
<td>Individual</td>
</tr>
<tr>
<td>6</td>
<td>End of process.</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
The Optimizer generates the parallel and common steps for the parcel as shown in the following illustration:

- **Parallel**
  - Copy and redistribute employee rows
  - Copy and redistribute call_emp rows
- **Common**
  - Lock both tables
- **Serial**
  - Merge join
  - Exclusion-merge join
  - Release locks

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 Database hashing algorithm.
Generator Component Processes

1. The Generator receives the optimized parse tree from the Optimizer and uses it to build plastic steps.

2. 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.

3. Depending on the nature of the request, one of the following outcomes occurs:

<table>
<thead>
<tr>
<th>IF the plastic steps were generated as the result of ...</th>
<th>THEN they are sent to ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a data-containing request parcel</td>
<td>1. the Request Cache.</td>
</tr>
<tr>
<td></td>
<td>2. OptApply 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>
</tbody>
</table>

4. End of process.

Definition: Plastic Steps

The Generator creates a series of optimized directives for the AMPs that contain column and row information, but no literal data values. These steps, called plastic steps, are later transformed into concrete steps by OptApply (see “Definition: Concrete Steps” on page 69). Plastic steps allow sets of values taken from the USING request modifier in parameterized-independent requests to be inserted into the optimized query plan during subsequent Parser operations.

The Generator writes the plastic steps into the Request Cache, then sends them to OptApply. 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 request modifiers are submitted repeatedly, albeit with different data, and will thus benefit from reusing the plastic steps.
OptApply

Introduction

OptApply produces concrete steps by placing data from a USING Data parcel set in a parameterized PK request (see “Terminology” on page 36), if present, into the plastic steps. Concrete steps are passed to the Dispatcher, which then distributes them to the AMPs via the BYNET.

For other parameterized requests, the Request Cache Peek subsystem places USING data into the parse tree prior to the Query Rewrite/Optimizer phases of request processing. See “Peeking at the Request Cache” on page 36 for details.

Block Diagram of OptApply Activity

The following diagram illustrates OptApply activity:
OptApply Component Processes

The OptApply component of the Parser engages in the following processing stages:

1. OptApply receives the plastic steps for a request from the Request Cache or from the Generator.
2. The plastic steps are compiled into executable machine code.
3. OptApply retrieves any parameterized PK request Data parcels associated with the Request parcel and applies them to the plastic steps, creating concrete steps.
   For any simple data-driven iterative insert request, the system optimizes the operation by processing multiple insert operations in a single AMP step.
   Note that the system performs this optimization only for simple iterated insert requests.
4. The concrete steps are sent to the Dispatcher, which sends them across the BYNET to the AMPs.
5. End of process.

Definition: Concrete Steps

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

Each concrete step is composed of four parts:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System message header</td>
<td>Contains the message class and kind, length, logical host identifier, session and request number, version number, and character set code.</td>
</tr>
<tr>
<td>Common step header</td>
<td>Contains account numbers and routing information. Specify 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).</td>
</tr>
<tr>
<td>Step components</td>
<td>A series of operation-dependent fields including step flags and the step mode.</td>
</tr>
<tr>
<td>Data</td>
<td>Contained in a varying length data area.</td>
</tr>
</tbody>
</table>
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 that submitted that request.
- To notify client applications and Teradata platform processors of aborted transactions and requests.

Execution Control

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

The Dispatcher routes the optimized query 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 available. This often means that one of a pair of deadlocked transactions must be aborted.

The Dispatcher process request and transaction aborts by notifying both the client application and all affected Teradata platform 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 request
- Internal Parser errors such as memory overflow
- Internal AMP errors such as primary index conflicts
- Transaction deadlocks

### Queue Table FIFO Cache Management

The Dispatcher maintains a cache that holds row information for a number of non-consumed rows for each queue table. The system creates this queue table cache, which resides in its own Dispatcher partition, during system startup. There can be a queue table cache task on each PE in your system.

The queue table FIFO cache row entries on a given PE are shared by all queue tables that hash to that PE. Each queue table row entry is a pair of QITS and rowID values for each row to be consumed from the queue, sorted in QITS value order. The entry for a given queue table exists only in one queue table cache on the system. The determination of which system PE is assigned to an active queue table is made within the Dispatcher partition by hashing its table ID using the following algorithm:

\[
\text{queue_table_cache_PE_number} = \text{tableID}[1] \mod \text{number_system_PEs}
\]

where:

<table>
<thead>
<tr>
<th>Syntax element ...</th>
<th>Specifies the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>queue_table_cache_PE_number</code></td>
<td>position number within the configuration map array of the PE containing the queue table cache to which the given queue table entry is assigned.</td>
</tr>
<tr>
<td><code>tableID[1]</code></td>
<td>numeric value of the second word of the double word tableID value (where the first word is notated as <code>tableID[0]</code>).</td>
</tr>
<tr>
<td><code>MOD</code></td>
<td>modulo function.</td>
</tr>
<tr>
<td><code>number_system_PEs</code></td>
<td>number of PEs in the system configuration.</td>
</tr>
</tbody>
</table>

For example, suppose a system has 6 online PEs and the value for `tableID[1]=34`.  

\[
\text{queue_table_cache_PE_number} = 34 \mod 6 = 4
\]
This value points to the fifth position in the system-maintained configuration map array of online PEs, which begins its numbering at position 0. As a result of the calculation, the system assigns entries for this queue table to the cache on the fifth online PE listed in the system configuration map. If that PE goes offline, then the system reassigns the queue table to the first online PE in the configuration map.

During startup, the system allocates 64KB to the queue table cache on each PE and increases its size dynamically in 64KB increments to a maximum of 1MB per PE as required. The system initializes all the fields in the cache during startup and allocates space for 100 table entries. As queue tables are activated, they populate these slots beginning with the lowest numbered slot and proceeding upward from there. When the maximum cache size is reached, the system flushes it, either partially or entirely, depending on the number of bytes that must be inserted into the cache.

See “CREATE TABLE (Queue Table Form)” in SQL Data Definition Language for more information about the queue table FIFO cache and its operation.
CHAPTER 2 Query Rewrite and Optimization

This chapter describes the fundamentals of query rewrite and 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 and cost optimization. Join planning is described in Chapter 3: “Join Planning and Optimization.” Joins typically consume significantly more optimization time than other query processes with the exception of Cartesian products.

This chapter does not describe the Teradata Index Wizard, which is also a component of the Optimizer.

- For information on the architecture of the Teradata Index Wizard, see Chapter 7: “Database Foundations for the Teradata Index and Statistics Wizards.”
- For information about using SQL statements as Teradata Index Wizard directives, see SQL Data Manipulation Language.
- For information about how to use the Teradata Index Wizard client utility, see Teradata Index Wizard User Guide.

---

1. A statistic is a single computed value 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 its properties. 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, so it is demographic, not statistical, information.
Query Rewrite

Introduction

Query Rewrite is a fundamental process undertaken by the Parser just prior to query optimization, and is the first piece of optimization performed in the preliminary stages of the query optimization process. 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 query optimizer can certainly take significant steps to close the gap between the two.

Stated semiformally, Query Rewrite is the process of rewriting query \( Q \) as query \( Q' \) such that:

- Query \( Q \) and Query \( Q' \) produce the identical answer set.
- Query \( Q' \) runs faster (is less costly) than Query \( Q \).

The various Query Rewrite techniques can either be rule-based (such as predicate move around) or cost-based (such as join index substitution).

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

This topic provides a survey of the stages of query rewrite undertaken by the Teradata Database Query Rewrite subsystem. The information is provided only to help you to understand what sorts of things Query Rewrite does and the relative order in which it does them.

2. Note that Query Rewrite is, in the strict sense, a self-contained process that precedes true query optimization. This is not just a peculiarity of the Teradata implementation, but is true universally. For the Teradata system, Query Rewrite steps are all performed in the Query Rewrite subsystem of the Parser (see "Resolver" on page 33).

3. Even trivial SQL queries can be written in an enormously large number of different ways. Consider the following verbally expressed query: “Get the names of suppliers who supply part P2.” Using the features available in the ANSI SQL-92 version of the language, it is possible to express this query in at least 52 different ways (assuming the system must access 2 tables in the process of returning the answer set). And this example only indicates the number of possible ways of writing the verbally expressed query as an SQL request. It does not even touch on the number of internal rewrites of those 52 individual SQL queries that various query optimization software packages might possibly undertake! (Date, 1998c).
Teradata Database Query Rewrite Processes

The following processes list the logical sequence of the processes undertaken by Query Rewrite and the Optimizer as they optimize a DML\(^4\) request.

Query Rewrite engages in the following process stages:

1. Query Rewrite receives the ResTree as its input from the Resolver (see “Resolver” on page 33) or from Request Cache Peek (see “Peeking at the Request Cache” on page 36).

2. The system performs Constant Predicate Evaluation (see “Constant Predicate Evaluation” on page 109) during the Query Rewrite preprocessing phase. This is done by searching for predicates that do not contain column references and generating the EVL code to evaluate the condition (at parse time). Running the EVL code returns TRUE, FALSE or UNKNOWN. The system replaces the predicate with the result when the result is TRUE or FALSE.

3. Wherever possible, Query Rewrite pushes projections into spooled views (see “Predicate Pushdown and Pullup” on page 115).

   This means that only those columns that are referenced by the main query are retained and pushed into spooled views, while other projections are dropped from the ResTree.

   Pushing projections into views can also enable views to be folded that would otherwise continue to be spool files. For example, views containing CASE expressions are not always merged; however, if the CASE expressions are removed from the view, view folding sometimes becomes possible.

   Pushing projections can also enable view elimination that would not otherwise occur.

   Some views are guaranteed to return only a single row (for example, when the select list specifies a single aggregate expression). If a query does not reference any columns of a single row view, Query Rewrite can reduce the view definition to `SELECT 0;` meaning that no table is referenced in the view.

4. Wherever possible, Query Rewrite converts outer joins to inner joins (see “OUTER JOIN-To-INNER JOIN Conversion” on page 94).

   Query Rewrite always converts outer joins to inner joins if the conversion does not produce nonmatching rows independently of the data.

   The rules for this are that Query Rewrite converts an outer join to an inner join if the following statements are true:

   - The WHERE clause contains one or more null filtering conditions\(^5\) on the inner table.
   - The outer join is involved with another join, and the other join has at least one null filtering condition on the inner relation of the outer join.

   In this case, the other join filters out the nonmatching rows of the outer join, making it possible to be converted to an inner join.

---

4. DDL and DCL requests are also optimized, but the primary goal of optimization is to ensure that DML requests run with as low a cost as is possible.

5. A null filtering condition, or NFC, is any predicate that evaluates to FALSE for nulls.
The following additional rule applies only for full outer joins:
If both of its relations have null filtering conditions, convert the outer join to an inner join.
Otherwise, do the following conversions:

<table>
<thead>
<tr>
<th>IF its ...</th>
<th>Has an NFC, then convert the full outer join to a ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>left relation</td>
<td>left outer join</td>
</tr>
<tr>
<td>right relation</td>
<td>right outer join</td>
</tr>
</tbody>
</table>

5 Whenever possible, Query Rewrite folds views (see “View Folding” on page 96).6 There are two categories of views when it comes to folding:

- Type 1 View Folding
  Any view that is not a Type 2 view.
  Type 1 spooling conditions (if any of the conditions is true):
  - Views with aggregates (views with GROUP BY, HAVING, WITH BY or aggregates in the select list) are spooled if any of the following conditions are satisfied:
    - View is not the only object in the query. Views can be folded in this case if the aggregation can be pulled up after joins.
      This optimization is referred to as lazy aggregation, and it is not yet implemented in Teradata. Future enhancements that include lazy aggregations will allow spooling aggregate views joined with other tables.
    - The query is an ABORT or ROLLBACK statement. The ABORT and ROLLBACK statements have limited syntax and do not allow clauses like GROUP BY and HAVING.
      Folding an aggregate view in this case could trigger an error for the abort statement and that is why aggregate views are spooled.
    - The main query has aggregations.
      Folding views in this case may cause nested aggregations, which Teradata does not yet support. It also may cause two levels of GROUP BY (both view and main query have a GROUP BY) which also is not supported currently.
    - The main query has window statistical functions.
      Folding views in this case might cause nested aggregations, which Teradata does not yet support. It also may cause two levels of GROUP BY (both view and main query have a group by) which also is not supported currently.
    - The view has extended grouping.
      As a side effect of folding a view is that conditions can be pushed into the tables referenced in the view. This could be a problem for views with extended grouping.

6. You should always read views to mean views and derived tables. Views and derived tables are semantically identical, and the Query Rewrite subsystem treats them so.
Both the main query and the view have OUTER JOINs and the view has a WHERE clause. It seems the restriction is based on the difficulty of merging the WHERE clause of the view into the main query. This case can be folded by moving the WHERE clause condition to the right place.

If the view is an outer table, the WHERE clause can be combined with the WHERE clause of the main query.

If the view is an inner table the WHERE clause can be combined with the ON clause that connects the view to the main query. This optimization will be done if time permits.

The view is the only object in the main query, but it references a constant in the definition and has no GROUP BY and HAVING clause. Folding such views caused a problem in the past if the main query references only the constant.

This case can be folded as well. A class of views called single-row views are identified as those aggregate views with no GROUP BY and HAVING clauses. Such views are guaranteed to produce a single row regardless of the data. If the main query only references constants from the view then we can either:

- Convert the view to a simple SELECT of the constants referenced in the query with no other clauses. If the main query references no columns of the view, the view can be simply rewritten as `SELECT 1;`
- Remove the view all together from the query and replace the view reference in the query with the constants.

Views with a DISTINCT operator. The presence of the DISTINCT operator in the view cannot be pulled up to the main query in general, which is why such views are spooled.

The view is defined on a single table with a WHERE clause and is involved in a FULL OUTER JOIN. The problem here is with the difficulty of where to place the view in the WHERE clause.

The view is an inner table of an Outer Join in the main query and has a constant, CASE expression, or ZEROIFNULL expression as well as one of the following conditions:

- The view is a derived table.
- The view is one of the relations specified in a FULL OUTER JOIN.
- The constant has casting.
- The constant, CASE, or ZEROIFNULL expression is referenced in an aggregate of the main query.

The system applies Type 2 view folding to cases that are not described by the preceding list.
• Views with set operations (UNION, MINUS or INTERSECT).
• Views with windowed statistical functions, a QUALIFY clause, or a SAMPLE clause.
• The main query is an UPDATE or DELETE and the view is defined with OUTER JOINs. This is an issue because UPDATE and DELETE statements do not allow OUTER JOINs.
• Type 2 View Folding is designed to handle problems that can otherwise occur under the following conditions:
  • Views and derived tables have constants or null manipulating CASE expressions in their definitions.
  and
  • Such views and derived tables are used as inner tables in OUTER JOIN operations.

The constants and null manipulating case expressions COALESCE and ZEROIFNULL are referred to as special expressions across the rest of this section to denote Type 2 Views.

The following example provides an idea of how Type 2 View Folding works:

Example:
Suppose you submit the following query:

```sql
SELECT ...
FROM T1 → T2 → v ON ... ON ...
```

where → indicates the join order and \( v(c,d,e) \) is defined as follows:

```sql
CREATE VIEW v(c,d,e) AS
  SELECT T3.C, T3.D, 10
  FROM T3 → T4 ON <..>;
```

The expected result of the query is the following:

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After the View Folding the query looks like the following SQL text:

```sql
SELECT t1.a, t2.b, t3.c, t3.d, 10 AS v.e
FROM t1 → t2 → t3 → t4 ON <..> ON <..>;
```
Chapter 2: Query Rewrite and Optimization

Query Rewrite

The expected result of the query is the following:

\[
\begin{array}{cccccc}
& t1.a & t2.b & v.c & v.d & v.e \\
\hline
1 & 1 & 1 & 1 & 10 \\
2 & 2 & 2 & ? & 10 \\
3 & 3 & ? & ? & ? \\
\end{array}
\]

Suppose the system joins \( t2 \rightarrow t3 \) first. Call the result \( res1 \). Because the join is done between a base relation \( t2 \) and a view relation \( t3 \), the system adds a derived column. As you can see, the derived column carries nulls for the nonmatching rows and nonnull values for the matching rows.

The first join undergoes the derived column setup phase.

\[
\begin{array}{cccc}
\text{res1} & t2.b & t3.c & dc(derived column) \\
\hline
& \bullet & \bullet & \bullet \\
1 & 1 & 1 \\
2 & 2 & 1 \\
3 & ? & ?
\end{array}
\]

Suppose the system joins \( t1 \rightarrow res1 \) which, when expanded, is \( t1 \rightarrow (t2 \rightarrow t3) \). In this case, the derived column comes from the inner table \( res1 \). Call this new result \( res2 \). In this join, the derived column undergoes the propagation phase.

\[
\begin{array}{cccc}
\text{res2} & t1.a & t2.b & t3.cd & c(derived column) \\
\hline
& \bullet & \bullet & \bullet & \bullet \\
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 1 \\
3 & 3 & ? & ? \\
4 & ? & ? & ?
\end{array}
\]

Suppose the system now makes the final join between \( res2 \) and \( t4 \) which, when expanded, is \( (t1 \rightarrow (t2 \rightarrow t3)) \rightarrow t4 \). In this join, the materialization and execution of the special expression occurs. As can be seen from the result, the expression is reported as 10 for all the rows where the derived column is 1, and reports null for rows where the derived column is null.
Query Rewrite can apply view folding at any time during the rewrite process, which allows
the subsystem to benefit from other rewrites. For example, conversion of a full outer join
to an inner join allows a view to be folded, but it would be spooled if it had not been
converted from a full outer join. This is distinct from the Resolver, which spools all views,
with the exception of updatable views, unconditionally (see “Resolver” on page 33).

1. Eliminate conditions through Satisfiability and Transitive Closure (see “Satisfiability and
   Transitive Closure” on page 102).
2. Push predicate conditions into spooled views (see “Predicate Marshaling” on page 123).
   Pushing predicate conditions into views helps performance by reducing the size of spooled
   view files.
3. Eliminate redundant joins (see “Eliminating Redundant Joins” on page 118).
4. Eliminate set operation branches (see “Eliminating Set Operation Branches” on page 117).
5. Push joins into UNION ALL branches (see “Pushing Joins Into UNION ALL Branches” on
   page 121).
6. Repeat stages 3 through 9 one additional time.
7. Query Rewrite produces the ResTree as its output, which is semantically equivalent to the
   ResTree input, and passes it to the Optimizer.

The system replaces the existing predicate with the predicate `<column1>*=<column2>`
where:

<table>
<thead>
<tr>
<th>Syntax element …</th>
<th>Specifies …</th>
</tr>
</thead>
<tbody>
<tr>
<td>*=</td>
<td>an equality marked with the NULLEQ flag.</td>
</tr>
</tbody>
</table>

8. End of process.

Note that Query Rewrite performs NULLEQ rewrites during its postprocessing phase. This
rewrite looks for predicates of the form `<column1>=<column2>` OR (<column1> IS
NULL AND <column2> IS NULL).
The following table summarizes the interactions among the different rewrites. Each row describes the impact of a single rewrite on the other rewrites. For example, the entry in the first row and third column is Yes, meaning that pushing projections into views can trigger view folding.

<table>
<thead>
<tr>
<th>PPV</th>
<th>O2I</th>
<th>VF</th>
<th>PCSV</th>
<th>SAT-TC</th>
<th>JE</th>
<th>SOBE</th>
<th>PJ-UA</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPV</td>
<td></td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O2I</td>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VF</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PCSV</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAT-TC</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JE</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOBE</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PJ-UA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some of the interactions between rewrites are fairly obvious, while others are not.
The following key to the preceding table, including some examples, explains the reasons behind the interactions:

<table>
<thead>
<tr>
<th>Row/Column Label</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPV</td>
<td>Pushing Projections into Views.</td>
</tr>
<tr>
<td></td>
<td>See “Projection Pushdown” on page 92.</td>
</tr>
<tr>
<td></td>
<td>• Type 2 View Folding is based on the presence of particular expressions in the view definition.</td>
</tr>
<tr>
<td></td>
<td>Pushing projections can eliminate such expressions and simplify Type 2 folding to Type 1 folding.</td>
</tr>
<tr>
<td></td>
<td>Another example is the case where a view whose definition contains window functions is joined to another table. Such a view cannot be folded; however, if the window functions are not referenced, they can be removed, which might enable view folding.</td>
</tr>
<tr>
<td></td>
<td>• One of the conditions of Join Elimination is that no Primary Key table columns are needed in the query block. Pushing projections can eliminate the Primary Key table columns and enable Join Elimination in the view block.</td>
</tr>
<tr>
<td></td>
<td>For example, consider a view with an FK-PK join that specifies columns from the dimension table in its SELECT list. If those columns are not referenced and are removed from the view, then it might become possible to eliminate the join to the dimension table.</td>
</tr>
</tbody>
</table>
Chapter 2: Query Rewrite and Optimization

Query Rewrite

O2I  Outer-to-Inner join conversion.
See “OUTER JOIN-To-INNER JOIN Conversion” on page 94.

- An outer-to-inner join conversion within a view definition or the query block referencing the view might make some views eligible for folding. This is true since there are a number of restrictions on folding views that either have OUTER JOINs or are somehow involved in OUTER JOINs.

- For OUTER JOINs, SAT-TC is applied to the ON clause conditions in isolation because it is not semantically correct to combine it with other ON clauses or with the WHERE clause. However, if an OUTER JOIN is converted to an INNER JOIN, the ON clause condition can be moved to the WHERE clause, thus providing more opportunities to apply SAT-TC.

- It is sometimes possible to apply Join Elimination to an INNER JOIN that has been converted from an OUTER JOIN. For example, the OUTER JOIN in the following query can be converted to an INNER JOIN based on the WHERE clause:

```sql
SELECT sales1.*
FROM product LEFT OUTER JOIN sales1
ON product_key=sales_product_key
WHERE quantity > 10;
```

The OUTER JOIN can be converted to an INNER JOIN because there is a null filtering condition on the inner table, `sales1`, making it possible to eliminate the join to `product`.

- Converting an OUTER JOIN to an INNER JOIN can also facilitate join pushing. For example, the LEFT OUTER JOIN in the following request can be converted to an INNER JOIN, making it possible for the join to the constrained dimension table `product` to be pushed into each branch of the UNION ALL view `sales`.

```sql
SELECT sales.*
FROM product LEFT OUTER JOIN sales
  ON product_key = sales_product_key
WHERE quantity > 10
  AND product_name LIKE 'French%';
```
Chapter 2: Query Rewrite and Optimization

Query Rewrite

### VF

**View Folding.**

See “View Folding” on page 96.

- Folding a view with an OUTER JOIN into a block that joins it to another table can facilitate the conversion of the OUTER JOIN into an INNER JOIN.
- Folding a view to combine its WHERE clause with the WHERE clause of the containing block provides additional opportunities to apply SAT-TC.
- View Folding can trigger Join Elimination.

For example, consider a view that contains only a dimension table. If the view is joined to a fact table, and the view is folded, it might then be possible to eliminate the join.

- Folding a view with an unsatisfiable condition into a block that is a branch of a set operation enables SOBE to remove the branch.

### PCSV

**Pushing query predicate Conditions into Spooled Views.**

See “Predicate Pushdown and Pullup” on page 115.

- Pushing predicates into a view can cause some OUTER JOINs within the view to become INNER JOINs.
- Pushing some predicates into a view enables more opportunities for SAT-TC to either find contradictions that can be eliminated or to derive new predicates.

### SAT-TC

**SATisfiability-Transitive Closure.**

See “Satisfiability and Transitive Closure” on page 102.

- TC might enable a null filtering condition to be introduced that could then be used to convert an OUTER JOIN to an INNER JOIN.
- It might be possible to push TC-derived predicates into views.
- The discovery by SAT of unsatisfiable conditions in a set operation branch would make it possible to eliminate that branch.

For example, SAT might eliminate the WHERE clause of an unsatisfiable query, which implies that some of the view columns that are referenced only in the WHERE clause can be removed from the view definition.

- SAT-TC also includes predicate simplification, which is a set of heuristic Query Rewrite methods the system uses to transform less performant predicates into forms that provide better performance.

The forms of predicate simplification supported for Teradata Database are the following:

- Constant predicate evaluation (done in QRW preprocessing phase)
- Range-based simplification
- Constant movearound
- Duplicate predicate removal
- NULLEQ marking (done in QRW postprocessing phase)
- Can trigger SAT-TC.
Query Rewrite

Join Elimination.

See “Eliminating Redundant Joins” on page 118.

- Eliminating an OUTER JOIN might enable an inner join to be released. To see this consider the join sequence (t1 INNER JOIN t2 ON t1.a1=a2) LEFT OUTER JOIN t3 ON b1=t3.a3. If a3 is a unique column, the LEFT OUTER JOIN can be eliminated. The remaining INNER JOIN can then be released by O2I conversion.

- Eliminating an OUTER JOIN between a view and a table might enable the view to be folded.

- Eliminating a join might enable a join to be pushed. For example, consider the following query:

  ```sql
  SELECT sales.*
  FROM sales
  INNER JOIN product
  ON sales_product_key=product_key
  INNER JOIN category
  ON product_category=category_key
  WHERE product_name LIKE 'French%';
  ```

  The join to `category` can be eliminated, enabling the join to `product` to be pushed into the `sales` view.

Set Operation Branch Elimination.

See “Eliminating Set Operation Branches” on page 117.

- Projection pushing is not allowed for set operation blocks that are either implicitly or explicitly DISTINCT; however, removing a DISTINCT branch can enable projection pushing.

- If removing branches of a set operation results in a single remaining SELECT request or UNION ALL set operation in a view, it might be possible to fold the view. If such a view cannot be folded (because, for example, it specifies a GROUP BY clause), then it might be possible for its join to be eliminated.

Pushing Joins Into UNION ALL Operations.

See “Pushing Joins Into UNION ALL Branches” on page 121.

- Pushing a join removes the constrained dimension table and the join predicate between it and the UNION ALL view from the block. Reapplying Projection Pushdown enables the Rewrite Driver to remove the foreign key column set from the UNION ALL branches if there are no remaining references to them.

- Pushing a join can trigger folding of a UNION ALL view if pushing results in the conditions outlined in “View Folding” on page 96 being met.

- Pushing a constrained join into a UNION ALL branch introduces new predicates into that branch that might be used by SAT-TC.

---

### Key

<table>
<thead>
<tr>
<th>Row/Column Label</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE</td>
<td>Join Elimination. See “Eliminating Redundant Joins” on page 118.</td>
</tr>
<tr>
<td>SOBE</td>
<td>Set Operation Branch Elimination. See “Eliminating Set Operation Branches” on page 117.</td>
</tr>
</tbody>
</table>

a. A null filtering condition, or NFC, is any predicate that evaluates to FALSE for nulls.
Query Rewrite Driver

The Rewrite Driver is the component of the Rewrite Subsystem that applies rewrite rules to each block of a request. The Query Rewrite Subsystem passes the rule set to the Rewrite Driver with each request to be rewritten. The Rewrite Driver maintains information about which rules are enabled for each block of the request. Initially, \textit{all} rules are enabled for each block.

The function of the Rewrite Driver is to apply the set of rewrite rules to each block of a request and, by exploiting interactions among the various rewrites, to reenable and reapply rewrites when possible.

The Rewrite Driver takes the next sequential rule to be applied from the rule set and applies it, either successfully or unsuccessfully, to each block in the order that is appropriate for the rule. When a rule is applied to a block, the Rewrite Driver takes two actions:

1. The rule is disabled for that block regardless of whether it was successfully applied or not.
2. Any triggers for the rule are applied.

A trigger simply reenables a rewrite rule for some block based on the interactions presented in the topic \textit{“Teradata Database Query Rewrite Processes” on page 75}. A trigger may re-enable a rule for the block to which the rule was applied, or for a containing (parent) or contained (for example, a subquery or view) block.

The Rewrite Driver makes two passes through the rule set, applying each rule to each block (if the rule is enabled for that block) on both passes. Two passes are made because this provides each rule with the opportunity to trigger every other rule.

To illustrate the functioning of the Rewrite Driver, consider the following simple example:

```sql
CREATE VIEW jan_feb_sales AS
SELECT sales_key, sales_date, sales_store_key,
    sales_product_key, quantity, amount
FROM sales1
UNION ALL
SELECT sales_key, sales_date, sales_store_key,
    sales_product_key, quantity, amount
FROM sales2;

SELECT SUM(amount*quantity)
FROM jan_feb_sales
WHERE EXTRACT(MONTH FROM sales_date) = 1;
```

The Rewrite Driver processes this request as indicated by the following process stages. Note that after each rule is applied to a block, it is disabled for that block. Also note that the stages of this process as outlined here indicate only some of the more interesting interactions that can be enabled by a rewrite rule.

b. The conversion of a join from INNER JOIN syntax to comma syntax is referred to as \textit{releasing} the join. A join can only be released if all joins in its query block can be released.

Releasing INNER JOINs presents a single canonical form of inner join to the Join Planner. This guarantees consistent plans for both syntaxes and eliminates the need for duplicate code to deal with both.
1. All rules are enabled for all blocks in the request.
2. Projection Pushdown is applied to all blocks.
   The Rewrite Driver removes `sales_key`, `sales_store_key`, and `sales_product_key` from the
   SELECT lists of each branch of the UNION ALL because they are not referenced by the
   containing block.
3. Outer-to-Inner Join Conversion is attempted, without success, on each block.
4. View folding is attempted, without success, on each block.
   The only block that is a candidate for folding is the `jan_feb_sales` view. This block cannot
   be folded because its containing block still has a WHERE clause with the predicate
   \( \text{EXTRACT(MONTH FROM sales_date) = 1} \), and the select list of the containing block
   does not match the select lists of each branch of the UNION ALL.
5. Predicate Pushdown is applied and the predicate \( \text{EXTRACT(MONTH FROM sales_date) = 1} \)
   is pushed into each branch of the UNION ALL.
6. SAT-TC is applied.
   This leads to an unsatisfiable condition in the UNION ALL branch containing `sales2`
   because `sales2` has the CHECK constraint \( \text{EXTRACT(MONTH FROM sales_date) = 2} \),
   and this contradicts the condition pushed into the block in Stage 5.
   This also includes an attempt at Predicate Simplification, without success, on each block.
7. Join Elimination is attempted, without success, on each block.
8. Set Operation Branch Elimination can be applied to the UNION ALL operation because
   one branch contains an unsatisfiable condition.
   The UNION ALL is replaced by the remaining branch. Applying the triggers for this rule
   reenables View Folding in the parent SELECT block.
9. Pushing Joins into UNION ALLs is attempted, without success, on each block.
10. End of first pass of the process.

At this point the Rewrite Driver has completed the first pass. The Rewrite Driver then makes a
second pass over the rule set. Some rule interactions reenable certain rules for some blocks.
For this example, the interesting interaction occurs in Stage 9 (refer to the boldface text). The
UNION ALL was replaced by the remaining SELECT request, and View Folding, which had
been disabled for each block in Stage 4, was reenabled for the `jan_feb_sales` view.

On this second pass, the view is successfully folded and the rewritten query is as follows:

```sql
SELECT SUM(amount * quantity)
FROM sales1
WHERE EXTRACT(MONTH FROM sales_date) = 1;
```

In this simple example, there was only one interesting rule interaction that enabled a rule that
had been unsuccessfully applied earlier, which occurred in Stage 9 during the first pass of the
Rewrite Driver over the request. For complex queries, however, many interesting rule
interactions might occur.
The individual rewrites interact with each other because applying a rewrite to a query block can enable other rewrites to be applied either to the same query block or to a parent block or sub-block of the block. To see this, consider the following example.

```sql
SELECT MAX(total)
FROM SELECT (product_key, product_name,
    SUM(quantity*amount) AS total
FROM sales, product
WHERE sales_product_key = product_key
    GROUP BY product_key, product_name) AS v
WHERE product_key IN (10, 20, 30);
```

Pushing the predicate from the outer block transforms the derived table $v$ into the following:

```sql
SELECT product_key, product_name, SUM(quantity*amount) AS total
FROM sales, product
WHERE sales_product_key = product_key
    AND product_key IN (10, 20, 30)
GROUP BY product_key, product_name;
```

Transitive Closure can now be applied to this block to derive the new predicate $sales_product_key$ IN (10, 20, 30). This newly added predicate can then be pushed into each branch of the sales view. This example shows that Predicate Pushing can enable Transitive Closure, which can, in turn, enable Predicate Pushing.

The example also illustrates an important property of the set of rewrite rules: the application of one rewrite can enable another rewrite that might have previously been applied to a block, either successfully or unsuccessfully.

For this example, consider what would happen if the Rewrite Driver applied Transitive Closure to each block first and then attempted to apply Predicate Pushing to each block, or vice versa, but then did not retry the first rewrite after applying the second. Neither of these simple sequences leads to the predicates getting pushed into each branch of sales.

Another property of each rewrite that is important to understand is the order in which it should be applied to each block of a query. Each rewrite can be applied either from the top down to each block before the blocks it contains or from the bottom up to each block contained in the block before the block itself.

For example, consider the following view definition and query against it:

```sql
CREATE VIEW sales_by_product AS
SELECT product_key, product_name, SUM(quantity*amount) AS total
FROM sales, product
WHERE sales_product_key = product_key
GROUP BY product_key, product_name;

SELECT SUM(total) AS total_sales
FROM sales_by_product;
```

For this query, Projection Pushdown can have a cascading effect by applying it from the top down. The only column of the view sales_by_product that is referenced is total, so the Projection Pushdown rewrite can remove both product_key and product_name from the select list of this view.

7. Recall that the semantics of derived tables are identical to those of views.
Furthermore, because `sales_by_product` only references `sales_product_key`, `quantity` and `amount` in the `sales` view select list, all of the other columns in the select list of `sales` (`sales_key`, `sales_date`, and `sales_store_key`) can be removed from each branch of the `sales` view. You can see that Projection Pushdown can have a cascading effect, because removing columns from the select list of a containing query block can cause columns in the select lists of nested views to become unreferenced, and thereby eligible for removal.

Likewise, Predicate Pushdown is most effectively applied from the top down because pushing a predicate into a data block might enable pushing the predicate again into a sub-block of its containing block.

On the other hand, View Folding is done from the bottom up because that ordering of the process guarantees that once folding is attempted on each view in a block, it is not necessary to try folding again for that block unless the rule is reenabled by another rewrite.

Obviously, the number of interactions among the individual rewrites is large. It is also true that no single order of applying each rewrite once is sufficient to guarantee that each block is rewritten completely because applying a rule may reenable another rule that was previously applied, whether successfully or not.
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. The vast majority of these semantically equivalent rewrites cannot be specified in the query itself, examples being the substitution of a join index for an explicitly specified join operation and the pushing of projections (see, for example, McMahan et al., 2004 and, for details, “Predicate Pushdown and Pullup” on page 115).

The following sequence lists many of the more important methods used by the Query Rewrite subsystem to rewrite SQL code:

- View folding (see “View Folding” on page 96)
- ANSI join syntax-to-comma join syntax (see “Converting ANSI Join Syntax To Comma Join Syntax” on page 91)
- Predicate marshaling\(^8\) (see “Predicate Marshaling” on page 123)
- Predicate push down and pullup\(^9\) (see “Predicate Pushdown and Pullup” on page 115)
- Outer join-to-inner join conversion (see “OUTER JOIN-to-INNER JOIN Conversion” on page 94)
- Satisfiability and Transitive Closure (see “Satisfiability and Transitive Closure” on page 102)
- Join elimination (see “Eliminating Redundant Joins” on page 118)
- Set operation branch elimination (see “Eliminating Set Operation Branches” on page 117)
- Pushing joins into UNION ALL branches (see “Pushing Joins Into UNION ALL Branches” on page 121)
- View materialization and other database object substitutions (see “View Materialization and Other Database Object Substitutions” on page 125)

---

8. Also referred to as predicate move around.

9. Also referred to as pushing conditions.
Converting ANSI Join Syntax To Comma Join Syntax

This rewrite, which is referred to as *releasing the join*, converts ANSI-style INNER JOIN syntax to comma-style syntax if the entire query is based on inner joins. For example, consider the following query:

```sql
SELECT *
FROM t1
INNER JOIN t2 ON a1=a2
INNER JOIN t3 ON a2=a3;
```

This query is converted to the following form by the rewrite:

```sql
SELECT *
FROM t1,t2,t3
WHERE a1=a2
AND a2=a3;
```
Projection Pushdown

Pushing projections can allow views to be folded that would remain as spool files otherwise. For example, views containing CASE expressions are not always merged. However, if these expressions can be removed, the system might be able to perform additional rewrites through the use of View Folding.

In the following example, the only column of the view `sales_by_product` that is referenced is `total`, so Projection Pushdown can remove both `product_key` and `product_name` from the select list of `sales_by_product`. Note that these column specifications must be retained in the GROUP BY clause for the view.

```
CREATE VIEW sales_by_product AS
  SELECT product_key, product_name, SUM(quantity*amount) AS total
  FROM sales, product
  WHERE sales_product_key = product_key
  GROUP BY product_key, product_name;

SELECT SUM(total) AS total_sales
FROM sales_by_product;
```

Furthermore, because `sales_by_product` only references `sales_product_key`, `quantity`, and `amount` in the `sales` view select list, all of the other columns (`sales_key`, `sales_date`, and `sales_store_key`) can be removed from each branch of the `sales` view. These columns can be removed safely only because the view is a UNION ALL view. Projection Pushdown cannot, however, be applied to UNION DISTINCT views because the elimination of duplicates is based on the select lists for the branches. This rewrite also cannot be applied to views defined with an explicit DISTINCT operator.

You can see from this example that Projection Pushdown can have a cascading effect, because removing columns from the select list of a containing query block can cause columns in the select lists of nested views to become unreferenced.

Another interesting case occurs when no columns of the view are referenced. This example again uses the view `sales_by_product`.

```
SELECT COUNT(*) AS cnt
FROM sales_by_product;
```

The request does not reference any columns from `sales_by_product`. As a result, the select list of `sales_by_product` can be rewritten as `SELECT 0`, because the rewrite only needs to ensure that the correct number of rows is returned.
Another special case, again based on the `sales_by_product` view, occurs when no columns of a view or derived table are referenced by the request, and the view is guaranteed to return only one row.

```
SELECT COUNT(*) AS cnt
FROM (SELECT SUM(total) AS total_sales
     FROM sales_by_product) AS dt ;
```

No rows of the derived table `dt` are referenced in this request, so the view contains only a single row because its select list specifies only an aggregate function. Such views are called **single-row views**. Projection Pushdown rewrites this request as follows:

```
SELECT COUNT(*) AS cnt
FROM (SELECT 0 AS dummy_col) AS dt ;
```

Spooling views with unreferenced columns is a common practice in query rewrite and optimization, and Projection Pushdown should help the performance of these cases by removing any unreferenced columns, thus reducing the size of spool files.
OUTER JOIN-To-INNER JOIN Conversion

In certain cases, an OUTER JOIN in a query block can be converted to an INNER JOIN.

Consider the following example:

```sql
SELECT DISTINCT product_name
FROM product LEFT OUTER JOIN sales ON product_key=sales_product_key
WHERE quantity > 10 ;
```

The OUTER JOIN in this request can be safely converted to an INNER JOIN because the subsequent WHERE predicate filters any nonmatching rows.

Similarly, the OUTER JOIN in the following request can be converted to an INNER JOIN because the ON clause predicate $b2 < 10$ is false if $b2$ is null, and therefore the condition removes all nonmatching rows of the OUTER JOIN:

```sql
SELECT *
FROM t1
LEFT OUTER JOIN t2 ON a1=a2
WHERE b2 < 10;
```

As a general rule, an OUTER JOIN can be converted to an INNER JOIN if there is a condition on the inner table that filters out nonmatching rows. In a LEFT OUTER JOIN, the right table is the inner table, while it is the left table in a RIGHT OUTER JOIN. In a FULL OUTER JOIN, both tables are inner tables. Conditions that are FALSE for nulls are referred to as null filtering conditions, and these are the conditions that enable the outer-to-inner join conversion to be made.

An OUTER JOIN can be converted to an INNER JOIN if at least one of the following conditions is true:

- The WHERE clause contains at least one null filtering condition on the inner table.
- The OUTER JOIN is involved in another join, and the other join condition has one or more null filtering conditions on the inner table. The other join in this case can be an INNER JOIN, LEFT OUTER JOIN, or RIGHT OUTER JOIN. It cannot be a FULL OUTER JOIN because there is no inner table in this case.

A null filtering condition on the right side of a FULL OUTER JOIN converts it to a LEFT OUTER JOIN, while a null filtering condition on the left side converts it to a RIGHT OUTER JOIN.

This rewrite also releases the join by converting the INNER JOIN syntax to comma syntax:

```sql
SELECT DISTINCT product_name
FROM product INNER JOIN sales ON product_key = sales_product_key
WHERE quantity > 10;
```
The Outer-to-Inner Join conversion rule rewrites this request as follows:

```
SELECT DISTINCT product_name
FROM product, sales
WHERE quantity > 10
AND   product_key = sales_product_key;
```

The conversion of a join from INNER JOIN syntax to comma syntax is referred to as *releasing the join*. A join can be released only if all joins in its block can be released.

Releasing inner joins presents one canonical form of inner joins to the Join Planner. This guarantees consistent plans for both inner join syntaxes and eliminates the need for duplicate code to handle them both.
View Folding

Introduction

View Folding is an important query rewrite technique in which a request that references a view is rewritten without making an explicit reference to that view. This eliminates the need to create a spool file for the view result and makes it possible for the Join Planner to consider additional join orders (see “Optimizer Join Plans” on page 318).

However, many views cannot be folded. For example, a view with aggregation that is joined to another table must be spooled.

Query Rewrite supports two types of View Folding:

- **Type 1**
  
  Type 1 View Folding is defined by exclusion to mean all View Folding that is not Type 2 View Folding.

- **Type 2**
  
  Type 2 View Folding is used to rewrite the following set of special expressions that are not handled by Type 1 View Folding:
  
  - Views and derived tables that have constants or null manipulating expressions in their select lists.

  Null-manipulating expressions include all of the following types:
  
  - CASE (see SQL Functions, Operators, Expressions, and Predicates)
  - COALESCE (see SQL Functions, Operators, Expressions, and Predicates)
  - ZEROIFNULL (see SQL Functions, Operators, Expressions, and Predicates)
  - UDFs (see SQL Data Definition Language and SQL External Routine Programming)
  - UDMs (see SQL Data Definition Language and SQL External Routine Programming)

  Constants and null-manipulating expressions are collectively referred to as null-sensitive expressions.

  - Views and derived tables specified as inner tables in OUTER JOIN operations

All other view and derived table folding is handled by Type 1 View Folding.

---

10. The semantics of views and derived tables are identical; therefore, any statement in this section that applies to views applies equally to derived tables, and any mention of views can be substituted for by a reference to derived tables without any change in the truth of the statement.
Examples of View Folding

Consider the following view definition and query against it:

```
CREATE VIEW sales_by_product AS
SELECT product_key, product_name, SUM(quantity*amount) AS total
FROM sales, product
WHERE sales_product_key = product_key
GROUP BY product_key, product_name;

SELECT product_name
FROM sales_by_product
WHERE total > 50000;
```

There is no need to evaluate the view result separately from the containing query block, so View Folding can be applied to yield the following rewritten query:

```
SELECT product.product_name
FROM sales, product
WHERE sales_product_key = product.product_key
GROUP BY product.product_key, product.product_name
HAVING (SUM(quantity * amount)) > 50000;
```

Spooling a view, on the other hand, means that the view definition is materialized and then treated as a single relation in the main query.

Query Rewrite attempts to fold views whenever it can because folding a view provides the Optimizer with more options for optimizing the query, while spooling the view does not permit its tables to be joined directly with other tables in the main query.

An important example of View Folding is folding UNION ALL views, as illustrated by the following example:

```
SELECT *
FROM sales;
```

where `sales` is a view involving 11 UNION ALL operations among the 12 sales months for the year.

This query is rewritten as follows:

```
SELECT *
FROM sales1
UNION ALL
SELECT *
FROM sales2
UNION ALL
...
UNION ALL
SELECT *
FROM sales12;
```
The requirement for this rewrite to take place is that the containing block be of the following form:

```
SELECT <column_list> FROM <UNION ALL view>
```

where there can only be a SELECT and a FROM clause in the view definition, and column_list is a set of columns that exactly matches the columns in the select lists of the UNION ALL view. If the set of columns in the containing query is a subset of the columns in the UNION ALL view, the rewrite still occurs because Query Rewrite applies Projection Pushdown before View Folding, so the unreferenced columns are removed from the view, leaving behind an exact match between the sets of columns.

The following example illustrates the need for Type 2 View Folding. The rewrite uses Type 2 View Folding because the view contains the null-sensitive expression ZEROIFNULL(quantity), it refers to an expression in the SELECT list of the containing block (qty), and it is on the inner side of an OUTER JOIN.

Consider the following view definition and query against it:

```
CREATE VIEW jan_sales AS
    SELECT sales_product_key, ZEROIFNULL(quantity) AS qty
    FROM sales1;

SELECT product_name, SUM(qty)
    FROM product LEFT OUTER JOIN jan_sales
    ON product_key=sales_product_key
    GROUP BY product_key, product_name;
```

A Type 1 rewrite of this request would be incorrect because the value of quantity can be NULL, as it was in sales1 (in which case the value of the ZEROIFNULL expression would be 0), or because there was no match in sales1 for a row in product, in which case the value of the ZEROIFNULL expression would be NULL. Type 2 View Folding solves this problem by tracking whether or not quantity was NULL in sales1 originally (meaning before the OUTER JOIN was made) and produces the correct value based on this tracking. This is illustrated by the following correct rewrite for the query against the jan_sales view using Type 2 View Folding:

```
SELECT product_name, SUM(CASE
    WHEN sales1.ROWID IS NULL THEN NULL ELSE quantity
END)
    FROM product LEFT OUTER JOIN sales1
    ON product_key=sales_product_key
    GROUP BY product_key, product_name;
```
**Rules for Folding Views**

To fold a view or derived table, Query Rewrite first determines whether it can be folded. If it can, then Query Rewrite operates on the view or derived table in the following stages:

1. Removes the table reference of the folded view or derived table from the containing query block.
2. Removes the view reference of the folded view or derived table from the view list of the containing query block.
3. Merges the FROM clauses of the containing block and the view or derived table.
4. Splits the WHERE clause of the containing block into the following groups of conditions:
   a. Those that can remain in the WHERE clause.
   b. Those that must be moved to the HAVING clause of the containing query block, for example, when the condition references an aggregate in the view or derived table.
5. Merges the WHERE and HAVING clauses of the containing block and the view or derived table.
6. Sets the DISTINCT flag of the containing query block to TRUE if the view or derived table is a SELECT DISTINCT. This is done for simple view merging only.
7. Changes all references to the view or derived table SELECT list to references to the expressions these references map to in the assignment list of the view or derived table.
8. Moves any SAMPLE, QUALIFY, and GROUP BY clauses in the view or derived table to the containing query block.
9. If there is a joined table tree for the containing query block, modify it to point to one of the following structures:

<table>
<thead>
<tr>
<th>IF the view or derived table ...</th>
<th>THEN modify it to point to this structure ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>is a single-table view or derived table</td>
<td>the table specified in the view definition.</td>
</tr>
<tr>
<td>specifies multiple tables but does not specify an outer join</td>
<td>a joined table tree of inner joins.</td>
</tr>
<tr>
<td>specifies an outer join</td>
<td>the joined table tree of the view or derived table.</td>
</tr>
</tbody>
</table>

The algorithm for folding views is driven by conditions would otherwise call for spooling a view. The exception is the class of queries called *simple queries*, which are defined as queries with a direct retrieval from a single view.
This case can be formally defined as follows:

- The view is the only object in the query.
- The view is not a derived table.
  The system spools derived tables unconditionally.
- The query does not have any clause other than a SELECT and a FROM.

This means that the main query cannot have any of the following operators or clauses:

- DISTINCT
- SAMPLE
- TOP \(n\)
- WHERE
- HAVING
- QUALIFY
- ORDER BY
- GROUP BY

- The main query can have only SELECT * or SELECT list of columns from the view

The logic of View Folding checks for a list of conditions to spool a view.

<table>
<thead>
<tr>
<th>IF the searched conditions are ...</th>
<th>THEN Query Rewrite ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>found</td>
<td>spools the view.</td>
</tr>
<tr>
<td>not found</td>
<td>folds the view.</td>
</tr>
</tbody>
</table>

The list of spooling conditions is summarized in the following list. Note that the list is checked only if the query is not a simple query.
The conditions that invoke spooling if any exists in the query are as follows:

- Views with aggregates, meaning those having a GROUP BY clause, HAVING clause, WITH ... BY clause, or aggregates in the select list, are spooled if any of the following conditions are satisfied:
  - The query is an ABORT or ROLLBACK statement.
  - The ABORT and ROLLBACK statements have a simple syntax that does not include clauses like GROUP BY and HAVING. Folding an aggregate view in this case could trigger an error for the ABORT or ROLLBACK statement, which is why they are spooled in this case.
  - The main query has aggregations.
  - Folding views in this case could cause nested aggregations, which the system does not support. It could also cause two levels of GROUP BY, in which both the view and the main query have a GROUP BY clause, which the system does not support.
  - The main query has windowing statistical functions. The reasoning is identical to that for main query aggregations.
  - The view has extended grouping sets.
  - A side effect of folding a view is that conditions can be pushed into the tables referenced in the view. This could be a problem for views with extended grouping sets.
  - Both the main query and the view have outer joins, and the view has a WHERE clause.
  - The view is the only object in the main query, but it references a constant in the definition and has no GROUP BY or HAVING clause.
  - Views with a DISTINCT operator.
  - In general, a DISTINCT operator in the view cannot pulled up to the main query, so such views must be spooled.
  - The view is defined on a single table with a WHERE clause and is part of a full outer join.
  - The view is an inner table of an outer join in the main query, and has a constant, CASE, or ZEROIFNULL plus one of the following conditions:
    - The view is actually a derived table.
    - The view is part of a full outer join.
    - The constant is casting to a different data type.
    - The constant, CASE, or ZEROIFNULL expression is referenced in an aggregate of the main query.
  - The view has any set operations such as the following:
    - UNION
    - MINUS/CEPT
    - INTERSECT
  - The view has windowing statistical functions, a QUALIFY clause, or a SAMPLE clause.
  - Either the main query or the view has the TOP N operator.
  - The main query is an UPDATE or DELETE and the view definition contains outer joins.
    This is excluded because neither UPDATE nor DELETE statements allow OUTER JOINs.
Satisfiability and Transitive Closure

Introduction

Satisfiability is a term from formal logic that describes checks Query Rewrite and the Optimizer perform to determine if a set of constraints is contradictory. The SAT-TC (Satisfiability-Transitive Closure) rewrite analyzes a set of predicates to determine if either of the following can be exploited to rewrite a request:

- A contradiction.
  A simple example is a condition like \( a=1 \text{ AND } a=0 \).
- Inferring new predicates from the existing predicate set.
  For example, suppose you submit a request that specifies the predicate \( a=1 \text{ AND } a=b \). This implies that \( b=1 \), which might be useful information for rewriting or otherwise optimizing the request.

SAT-TC also includes the Predicate Simplification query rewrite (see “Predicate Simplification” on page 109).

Algorithms for SAT and TC

The problems of SAT and TC are inherently related. For example, we could figure out that \( (a=1) \text{ AND } (a=2) \) is contradictory from \( \text{TC}\{ (a=1) \text{ AND } (a = 2) \} = \{ (1 = 2) \} \).

Another example, TC \( \{ (a >= 2 \text{ AND } a <= b \text{ AND } b <= 1) \} = \{ b >= 2 \text{ AND } a <= 1 \text{ AND } 2 <= 1 \} \), has the contradiction of \( 2 <= 1 \). The two previous examples suggest that SAT is actually a by-product of TC.

Many algorithms have been developed to solve TC and SAT. One of the better known algorithms was developed by Rosenkrantz and Hunt (1962), and is based on the shortest path algorithm of Floyd (1962). The Rosenkrantz-Hunt algorithm considers only conjunctive conditions, where each condition is of the form \( (X \text{ op } Y + C) \text{ OR } (X \text{ op } C) \). Both \( X \) and \( Y \) are INTEGER variables, \( C \) is an INTEGER constant, and \( \text{op} \in \{ <, =, >, \geq, \leq \} \).

The IN clause is commonly specified in SQL requests, but it can also be specified in the definition of a sparse join index (see “CREATE JOIN INDEX” in SQL Data Definition Language for more information about sparse join indexes). The conditions for a join index definition are assumed to be conjunctive, and each comparison is one of the following forms:

- \( (X=Y) \)
- \( X \text{ IN } (value_1, value_2, \ldots, value_n) \)
  where \( value_1, value_2, \ldots, value_n \) are all constants.
- \( X <\text{operator}> \text{ constant} \)
  where \( <\text{operator}> \in \{ <, =, >, \geq, \leq \} \)

11. More generally, “Propositional Satisfiability is the problem of determining, for a formula of the propositional calculus, if there is an assignment of truth values to its variables for which that formula evaluates to true. By SAT we mean the problem of propositional satisfiability for formulas in conjunctive normal form...” (Cook and Mitchell, 1997, page 1).
Note that both algorithms work strictly for *variables* and not for operators on variables. The exception is the EXTRACT function for dates, which is treated like a variable in both algorithms. Also, if a condition has a constraint $C_1$ on a DATE column, and another condition has a constraint on an EXTRACT of that same column, then the system adds an EXTRACT constraint using $C_1$.

For example, for the set of conditions $\{o_{\text{orderdate}} = '1999-05-01' \text{ AND } \text{EXTRACT(MONTH FROM } o_{\text{orderdate}}) > 2\}$, the system adds $\text{EXTRACT(MONTH FROM } o_{\text{orderdate}}) = 5$ based on $o_{\text{orderdate}} = '1999-05-01'$.

Another set of operators that is handled by both algorithms is $\text{IS NULL}$ and $\text{IS NOT NULL}$. Neither operator is applicable to TC, but both are applicable to SAT. Both algorithms return FALSE for SAT if they encounter $x \text{ IS NULL AND } x \text{ IS NOT NULL}$. Note that both algorithms are independent of whether the variables are nullable or not.

The algorithms can also be applied to cross-query Transitive Closure operations, meaning transitive closure between outer and inner query blocks. This allows conditions to be pushed into and out of subqueries. The basic approach is to combine the query block conditions before computing the transitive closure. The $\text{IN}$ and $\text{NOT IN}$ operators are treated as $=$ and $\neq$ operators, respectively. Derived conditions are added, as appropriate, to each query block.

### Determining If a Set of Constraints Is Satisfiable

The problem of checking for contradictions in predicates, such as $\text{WHERE } a=1 \text{ AND } a=2$ is called the satisfiability problem, or SAT. Any solution to SAT produces a set of constraints that is either declared to be FALSE if it is contradictory, or TRUE if it is satisfiable for some specific data.

There are two more applications to SAT in the use and maintenance of join indexes. The first problem is to determine if a join index requires maintenance for a maintenance operation performed on one of its base tables. This problem can be solved by calling SAT for the conjunction of the join index conditions and the condition applied in the base table maintenance. For example, the following operation implies that there is a requirement to do maintenance for $j_2$, but not for $j_1$:

```sql
DELETE lineitem WHERE \text{EXTRACT(MONTH FROM } l_{\text{shipdate}}) = 12;
```

This decision can be made because SAT returns TRUE for $\text{EXTRACT(MONTH FROM } l_{\text{shipdate}}) = 12 \text{ AND EXTRACT(MONTH FROM } l_{\text{shipdate}}) \geq 7$, and FALSE for $\text{EXTRACT(MONTH FROM } l_{\text{shipdate}}) = 12 \text{ AND EXTRACT(MONTH FROM } l_{\text{shipdate}}) \leq 6$.

The problem of determining whether a join index completely or partially covers a query can be solved in general as a number of SAT problems. Note that the use of SAT in this problem is more important once more complex conditions are specified in the join index definition.

Satisfiability is done automatically by Query Rewrite and the Optimizer, and does not require user intervention. When SAT is applied to a request, certain queries or maintenance operations might run faster.
**Transitive Closure of a Set of Constraints**

The Transitive Closure or TC of a set of constraints \( s_1 \), denoted by TC\((s_1)\), is the set of all possible derivable constraints from \( s_1 \). For example, if \( s_1 \) is \((a=b \text{ and } a=1)\), TC\((s_1)\) is \((b=1)\).

In many decision support and CRM applications, TC is needed to optimize date ranges and IN clause. The following example request illustrates one of these cases:

```sql
SELECT l_shipmode, SUM (CASE
    WHEN o_orderpriority = '1URGENT'
    OR o_orderpriority = '2-HIGH'
    THEN 1
    ELSE 0
END)
FROM lineitem
WHERE l_commitdate < l_receiptdate
AND l_shipdate < l_commitdate
AND l_receiptdate >= '1994-01-01'
AND l_receiptdate < ('1994-06-06')
GROUP BY l_shipmode;
```

From the example, it is possible to find the sequence of \( \leq \) as follows:

\[
s_1 = (l_{\text{shipdate}} \leq l_{\text{commitdate}} - 1 \\
      \text{AND } l_{\text{commitdate}} \leq l_{\text{receiptdate}} - 1 \\
      \text{AND } l_{\text{receiptdate}} \leq '1994-06-05')
\]

The new set of constraints that can be derived from \( s_1 \) or TC\((s_1)\) is the following:

\[
(l_{\text{commitdate}} \leq '1994-06-04' \\
  \text{AND } l_{\text{shipdate}} \leq '1994-06-03').
\]

If `lineitem` or one of its join or covering indexes is either value-ordered or partitioned on `l_shipdate`, the new constraint `l_shipdate \leq '1994-06-03'` enables the system to access only a portion of the table instead of doing a full-table scan.

Similar to the SAT test, TC is computed and applied to queries automatically by the Optimizer. You might notice performance improvements for some queries because of the extra predicates TC adds. The extra predicates can also be seen in the EXPLAIN reports for requests that use them.

**SAT-TC and Query Rewrite**

One important aspect of using Transitive Closure for query rewrite is its application across ON and WHERE clauses. For example, suppose you have the following request:

```sql
SELECT product_name, sum(amount*quantity) AS qty
FROM product LEFT OUTER JOIN sales1
ON product_key=sales_product_key
WHERE product_key=10
GROUP BY product_key, product_name ;
```

For this request, Transitive Closure adds the inferred predicate `sales_product_key=10` to the ON clause. This is particularly effective when the predicate added to the ON clause is a constraint on the primary index of the inner table in a join.
Another important property of Transitive Closure is its ability to infer new predicates across the ON clauses of consecutive inner joins. For example, consider the following request:

```sql
SELECT product_key, product_name, SUM(s1.amount * s1.quantity+s2.amount * s2.quantity) AS total
FROM product
LEFT OUTER JOIN
((sales1 AS s1 INNER JOIN store
  ON s1.sales_store_key = store_key)
INNER JOIN sales2 AS s2
  ON s2.sales_store_key = store_key
AND s2.sales_store_key = 10)
ON product_key = s1.sales_product_key
AND product_key = s2.sales_product_key
GROUP BY product_key, product_name;
```

To see this application of Transitive Closure, consider the consecutive inner joins between `s1`, `s2`, and `store`. The predicates in consecutive inner joins can be treated collectively by Transitive Closure as if they were specified in a single WHERE clause. In this example Transitive Closure processes these predicates as if they appeared as the following compound predicate:

```sql
WHERE s1.sales_store_key = store_key
AND s2.sales_store_key = store_key
AND s2.sales_store_key = 10
```

By grouping the predicates logically like this, Transitive Closure can derive the new predicates `store_key=10` and `s1.sales_store_key=10` and then place them in the ON clause of the uppermost INNER JOIN in the set of consecutive inner joins. In this example, that is the ON clause joining to `s2`.

If a condition is false mathematically, it is said to be **contradictory** or **unsatisfiable**. In this context, the opposite of contradictory is **satisfiable**, regardless of the data. An example might be specifying the conditions `a=1` and `a=2` in the same request. If Query Rewrite discovers such an unsatisfiable condition, it simplifies and optimizes the condition in such a way that all joins and retrievals can be done on a single AMP basis.

One way to fix a contradictory condition is to add CHECK constraints to the request, enabling Query Rewrite to eliminate unnecessary conditions, thus permitting the Optimizer to later construct better execution plans (see “Transitive Closure of a Set of Constraints” on page 104).

For example, assume that you want to list all orders made in the first three months of the fiscal year. Assuming that you have access only to the `ordertbl` view, and the query looks like this:

```sql
SELECT *
FROM ordertbl
WHERE EXTRACT(MONTH FROM o_orderdate) <= 3;
```

Without being able to add such constraints to a query during query rewrite, the system must access all of the tables `orders1`, `orders2`, … `orders12` using the constraint `EXTRACT(MONTH FROM o_orderdate) <= 3`, even though it only needs to access `orders1`, `orders2`, and `orders3` to satisfy the request. The only way Query Rewrite knows to filter out the other nine tables is to add CHECK constraints for every table to the query, and then to determine the contradiction between the CHECK constraint and the query constraint.
For example, if the CHECK constraint on `order4` is added to the corresponding step, Query Rewrite sees the following compound predicate, which is a contradiction.

```
EXTRACT(MONTH FROM o_orderdate) <= 3
AND
EXTRACT(MONTH FROM o_orderdate) = 4
```

For this particular case, Query Rewrite can simply eliminate this step. In general, Query Rewrite needs to know if a set of constraints is satisfiable.

Of course, you would rarely submit a contradictory query that does not return results regardless of the data like `a=1` and `a=2`. The example in the previous paragraph indicates the need for such checks. As previously stated, this issue is referred to as the satisfiability problem. Any solution to satisfiability generates a set of constraints and either declares them to be FALSE to denote that they are contradictory, or TRUE, which means that for some specific data, the set of constraints is satisfiable.

You should be aware of the cost of enforcing CHECK constraints if you decide to use them for horizontal table partitioning. Note, however, that the use of CHECK constraints in query optimization is done automatically by the Optimizer and does not require user intervention.

Another way to partition tables horizontally is to define them as partitioned primary index tables. This should be your first choice for horizontal partitioning, and you should consider using CHECK constraints only for those cases where a PPI is not possible.

**SAT-TC and Query Optimization**

Query optimization introduces two more applications of satisfiability in the usage and maintenance of join indexes:

- Determining whether a join index must be updated to keep in synchrony with an update operation\(^{12}\) on one or more of its base tables.
- Determining whether a join index partly or fully covers a query.

The join index update problem can be solved by using satisfiability for the conjunction of the join index conditions and the condition applied in the base table maintenance.

Assume the following sparse join index definitions:

```
CREATE JOIN INDEX j1 AS
SELECT *
FROM lineitem
WHERE EXTRACT (MONTH FROM l_shipdate) <= 6;
```

```
CREATE JOIN INDEX j2 AS
SELECT *
FROM lineitem
WHERE EXTRACT (MONTH FROM l_shipdate) >= 7;
```

Now consider the following delete operation on the `lineitem` table for example:

```
DELETE lineitem
WHERE EXTRACT (MONTH FROM l_shipdate) = 12;
```

\(^{12}\) The term update operation here signifies an insert, update, or delete operation against the base table in question.
This implies that there is a need to update \( j_2 \), but not \( j_1 \). The system can make this decision because the Satisfiability check returns TRUE for the following predicate:

\[
\begin{align*}
\text{EXTRACT(MONTH FROM l_shipdate)} &= 12 \\
\text{AND} \\
\text{EXTRACT(MONTH FROM l_shipdate)} &\geq 7
\end{align*}
\]

but FALSE for this predicate:

\[
\begin{align*}
\text{EXTRACT(MONTH FROM l_shipdate)} &= 12 \\
\text{AND} \\
\text{EXTRACT(MONTH FROM l_shipdate)} &\leq 6.
\end{align*}
\]

The problem of determining whether a join index completely or partially covers a query is solved as a set of satisfiability problems. Note that the use of satisfiability in this problem becomes more important when more complex conditions, like constants in WHERE clause predicates, are specified in the join index definition.\(^\text{13}\)

Transitive Closure is a relational algebraic method Query Rewrite and the Optimizer use to solve questions of the sort "can I get to point \( b \) from point \( a \)?\(^\text{14}\) The Transitive Closure of a set of constraints \( S_1 \), denoted by TC(\( S_1 \)), is the set of all possible derivable constraints from \( S_1 \).

For example if \( S_1 \) is \( (a=b \text{ AND } a=1) \) then TC(\( S_1 \)) is \( (b=1) \).

Query Rewrite finds Transitive Closure, but limited to simple cases like the previous example. Query Rewrite only finds TC(\( S_1 \)) if \( S_1 \) is a conjunction of constraints, and it only derives new ones for a sequence of equality constraints.

In many decision support and Customer Relationship Management applications, Transitive Closure is needed to optimize date range predicates and IN clauses.

The following example illustrates one of these cases:

\[
\begin{align*}
\text{SELECT L_SHIPMODE, SUM(CASE WHEN o_orderpriority = '1-URGENT' OR o_orderpriority = '2-HIGH' THEN 1 ELSE 0 END)} \\
\text{FROM lineitem} \\
\text{WHERE l_commitdate < l_receiptdate} \\
\text{AND l_shipdate < l_commitdate} \\
\text{AND l_receiptdate} &\geq '1994-01-01' \\
\text{AND l_receiptdate} &< ('1994-06-06')
\end{align*}
\]

GROUP BY l_shipmode;

From this example, you can find the sequence of \( \leq \) as follows:

\[
S_1, = (l_shipdate \leq l_commitdate-1 \\
\text{AND } l_commitdate \leq l_receiptdate-1 \\
\text{AND } l_receiptdate \leq '1994-06-05')
\]

---

\(^\text{13}\) This happens, for example, when you create a sparse join index. See the chapter on hash and join indexes in Database Design and the documentation for CREATE JOIN INDEX in SQL Data Definition Language for more information about sparse join indexes.

\(^\text{14}\) This problem is often referred to as Reachability and the determination of its answer is called the Reachability problem.
The new constraints that can be derived from $S_1$ or $TC(S_1)$ are the following set:

$\begin{align*}
1\text{_commitdate} & \leq '1994-06-04' \\
\text{AND } 1\text{_shipdate} & \leq '1994-06-03'
\end{align*}$

If `lineitem` or one of its join or covering indexes is value-ordered or distributed to the AMPs on `l\_shipdate`, then the new constraint `l\_shipdate <= '1994-06-03'` allows the system to access only a portion of the table instead of performing a full table scan.

The problems of Satisfiability and Transitive Closure are inherently related. For example, it is fairly simple to determine that $(a=1)\text{AND}(a=2)$ is contradictory from the following:

$\text{TC}\{(a=1) \text{ AND } (a=2)\} = \{(1=2)\}$

Consider another example, which has the contradiction of $(2\leq1)$:

$\text{TC}\{(a=2 \text{ AND } a\leq b \text{ AND } b\leq1)\} = \{(b=2 \text{ AND } a\leq1 \text{ AND } 2\leq1\}$

These examples suggest (correctly) that satisfiability is a by-product of Transitive Closure.
Predicate Simplification

Introduction

Predicate Simplification is a set of heuristic\(^{15}\) Query Rewrite methods the system uses to transform less performant predicates into forms that provide better performance.

The complete set of Predicate Simplification techniques is as follows:

- Constant Predicate Evaluation
- Range-Based Simplification
- Constant Movearound
- Duplicate Predicate Removal
- NULLEQ Marking

The triggers for performing Predicate Simplification are the same as those for SAT-TC. Predicate Simplification and SAT-TC (see “Satisfiability and Transitive Closure” on page 102) can trigger one another.

Note that while constant predicate evaluation is conceptually a component of Predicate Simplification, it is performed in the preprocessing phase of Query Rewrite.

Similarly, NULLEQ marking is performed in the postprocessing phase of Query Rewrite.

Constant Predicate Evaluation

Constant Predicate Evaluation evaluates predicates that contain only constants (those having no column references) at the time a request is parsed. For example, consider the following complex predicate:

\[ t1.pi = t2.pi \]

OR 'a' IN ('b', 'c')

The system generates EVL code that returns only the values TRUE, FALSE, or UNKNOWN to evaluate such predicates, and then replaces the predicate by the result if it is either TRUE or FALSE.

The IN predicate in this example can be evaluated to FALSE when Teradata Database parses the request, so Query Rewrite transforms it as follows:

\[ t1.pi = t2.pi \]

Because Constant Predicate Evaluation might otherwise interact with other rewrites, Teradata Database executes it in the preprocessing phase of Query Rewrite rather than in the Query Rewrite system itself.

\(^{15}\) Specifically meaning that these techniques are not cost-based.
Range-Based Simplification

Range-Based Simplification evaluates single table predicates and rewrites them to eliminate overlapping, or redundant, conditions. It also consolidates consecutive ranges into a single range. Range-Based Simplification can be applied to any data type for which ranges of values can be specified in predicates. The basic idea of Range-Based Simplification is to eliminate redundancy in a set of predicates by finding overlapping ranges. For example, consider the following set of predicates:

\[
\begin{align*}
& a_1 \ \text{NOT IN} \ (1,3) \\
& \text{AND} \ a_1 < 4 \\
& \text{AND} \ a_1 \geq 2 \\
\end{align*}
\]

If \(a_1\) is an INTEGER type, these predicates can be simplified as follows:

\[
a_1 = 2
\]

Similarly, consider the following predicate:

\[
\begin{align*}
& \text{int\_col \ IN} \ (1, 3, 5, 7, 9) \\
& \text{AND} \ \text{int\_col} > 4
\end{align*}
\]

where \(\text{int\_col}\) is a column typed as INTEGER.

The Range-Based Simplification rewrite transform this predicate as follows:

\[
\begin{align*}
& \text{int\_col} = 5 \\
& \text{OR} \ \text{int\_col} = 7 \\
& \text{OR} \ \text{int\_col} = 9
\end{align*}
\]

This rewrite enables the Optimizer to calculate more accurate selectivity estimates while at the same time eliminating the execution of redundant predicates.

To accomplish Range-Based Simplification, each predicate of the form \(<\text{column}>\) \(<\text{comparison\_operator}>\) \(<\text{constant}>\) is represented by a value range, where:

<table>
<thead>
<tr>
<th>Syntax element ...</th>
<th>Specifies ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{column}&gt;)</td>
<td>a column reference.</td>
</tr>
<tr>
<td>(&lt;\text{comparison_operator}&gt;)</td>
<td>one of the following Boolean operators:</td>
</tr>
<tr>
<td></td>
<td>(&lt;) \quad \geq)</td>
</tr>
<tr>
<td></td>
<td>(&gt;) \quad =)</td>
</tr>
<tr>
<td></td>
<td>(\leq) \quad \(&lt;)</td>
</tr>
<tr>
<td>(&lt;\text{constant}&gt;)</td>
<td>a constant of any type for which a range is valid, such as any of the following types:</td>
</tr>
<tr>
<td></td>
<td>(\text{CHARACTER}) \quad \text{REAL})</td>
</tr>
<tr>
<td></td>
<td>(\text{DATE}) \quad \text{TIME})</td>
</tr>
<tr>
<td></td>
<td>(\text{DECIMAL}) \quad \text{TIMESTAMP})</td>
</tr>
<tr>
<td></td>
<td>(\text{FLOAT}) \quad \text{VARCHAR})</td>
</tr>
<tr>
<td></td>
<td>(\text{INTEGER (all types)})</td>
</tr>
</tbody>
</table>
Each individual conjunctive or disjunctive operator can be represented as a range set, and an AND/OR can be represented by the intersection/union of the range sets for each conjunct/disjunct.

In the earlier example of the predicate \( a_1 \text{ NOT IN (1,3) AND } a_1<4 \text{ AND } a_1>=2 \), the set of ranges\(^{16}\) is as follows:

1. \( a_1 \text{ NOT IN (1,3)} \) : \((-\infty, 0], [2, \infty), (-\infty, 2], [4, \infty)\)
2. \( a_1 < 4 \) : \((-\infty, 3]\)
3. \( a_1 >= 2 \) : \([3, \infty)\)

Each range set represents a union of individual ranges, where:

<table>
<thead>
<tr>
<th>Syntax element ...</th>
<th>Specifies ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inf</td>
<td>negative infinity.</td>
</tr>
<tr>
<td>inf</td>
<td>positive infinity.</td>
</tr>
<tr>
<td>[</td>
<td>an inclusive lower bound.</td>
</tr>
<tr>
<td>]</td>
<td>an inclusive upper bound.</td>
</tr>
<tr>
<td>(</td>
<td>an exclusive lower bound.</td>
</tr>
<tr>
<td>)</td>
<td>an exclusive upper bound.</td>
</tr>
</tbody>
</table>

Each range is either discrete or continuous. These terms are defined by the following table:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
</table>
| Discrete range      | A range for which the next or previous value for any value \( x \) can be found by executing either of the following operations, respectively:  
|                     | \( x + 1 \) for the next value in the range  
|                     | \( x - 1 \) for the previous value in the range  
|                     | A range of integer numbers is an example of a discrete range. |
| Continuous range    | By default, any range that is not discrete is continuous.  
|                     | A range of floating point numbers is an example of a continuous range. |

A range set can be simplified by unioning the ranges within it. For example, by unioning its ranges, range set 1 can be simplified to the following:

\[ \{(-\infty, 0], [2, 2], [4, \infty)\} \]

\(^{16}\) The ranges are numbered for reference only. No specific ordering is implied.
Similarly, it is possible to union of range sets to simplify ORs, and to do an intersect of range sets to simplify ANDs. Intersecting range sets 1 and 2, which is a valid operation because the predicates they represent are ANDed in the original predicate, yields the following range set:

\[ \{(-\infty, 0], [2, 2]\} \]

Intersecting range sets 4 and 3 yields the final range set:

\[ \{[2, 2]\} \]

This range set translates to the predicate \( a_1 = 2 \), which can replace the original predicate in the query.

Range-Based Simplification can lead to a range set containing a single empty set, symbolized by \( \{\emptyset\} \), which indicates that the predicate is unsatisfiable, or the range \((-\infty, +\infty)\), which indicates that the predicate does not constrain the set of possible values.\(^{17}\)

An example of the first case is the following predicate:

\[
a_1 \text{ NOT IN } (1, 2, 3) \\
\text{AND } a_1 \geq 1 \\
\text{AND } a_1 \leq 3
\]

The predicate can be rewritten as follows if \( a_1 \) is an INTEGER column:

\[ 0 = 1 \]

This, of course, evaluates to FALSE.

An example of the second case is the following predicate:

\[
a_1 > 1 \\
\text{OR } a_1 < 2
\]

The predicate can be rewritten as follows:

\[ a_1 \text{ IS NOT NULL.} \]

The Range-Based Simplification rewrite method can detect, and then union, contiguous discrete ranges.

For example, consider the following predicate:

\[
(a_1 \geq 1 \text{ AND } a_1 \leq 3) \\
\text{AND } (a_1 \geq 4 \text{ AND } a_1 \leq 10)
\]

This predicate can be rewritten by incrementing the inclusive upper bound of one range and checking to see if the incremented value equals the inclusive lower bound of another set. If so, the ranges can be unioned as long as they are not points.\(^{18}\)

The rewrite is as follows:

\[
a_1 \geq 1 \\
\text{AND } a_1 \leq 10
\]

\(^{17}\) Note that such a predicate does filter nulls, which are, by definition, not values.

\(^{18}\) In this case, a point is defined as a range whose lower bound equals its upper bound.
**Constant Movearound**

The Constant Movearound rewrite method can enable better selectivity estimates and wider use of indexes by the Optimizer. Constant Movearound attempts to move constants from one side of a Boolean comparison operator to the other in order to rewrite predicates from the following form:

\[ \text{<column> } <\pm> \text{<constant_1> } \langle\text{comparison_operator}\rangle \text{<constant_2>} \]

to this form:

\[ \text{<column> } \langle\text{comparison_operator}\rangle \text{<constant>} \]

The method does this by removing the plus or minus operation from the column side of the predicate and then adding its negation to the constant-only side of the predicate. The constant expression is then folded. Any errors that occur during folding, such as overflow or underflow, cause the rewrite to be rejected in favor of using the original expression.

This transformation is only done for the plus and minus arithmetic operators. The constant expression that is moved can be an INTERVAL expression as in, for example, DATE operations.

For example, consider the following WHERE predicate:

\[ \text{date_col + INTERVAL '3' MONTH } \leq \text{ '2007-03-31' } \]

This predicate can be rewritten as follows using Constant Movearound:

\[ \text{date_col } \leq \text{ '2006-12-31' } \]

**Duplicate Predicate Removal**

Duplicate Predicate Removal eliminates identical conjuncts, disjuncts, or both from AND and OR predicates. The maximum number of duplicate conjuncts and disjuncts compared per predicate is 100.

**NULLEQ Marking**

NULLEQ Marking transforms binary join predicates for those cases where nulls should be considered equal to one another. More abstractly, NULLEQ marking evaluates predicates of the form:

\[ \text{<column_1> = <column_2> OR (<column_1> IS NULL AND <column_2> IS NULL)} \]

and rewrites them in this form:

\[ \text{<column_1> *= <column_2> } \]
where:

<table>
<thead>
<tr>
<th>Syntax element</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>*=</td>
<td>an equality operator that compares nulls as equals and all combinations of values the same as the standard = operator.</td>
</tr>
</tbody>
</table>

For example, consider the following predicate:

\[
t1.a1 = t2.a2 \\
\text{OR} \ (t1.a1 \text{ IS NULL AND } t2.a2 \text{ IS NULL})
\]

By using NULLEQ Marking, this predicate can be rewritten as follows:

\[
t1.a1* = t2.a2
\]

Because this method might otherwise interact with SAT-TC, Teradata Database executes it in the postprocessing phase of Query Rewrite rather than in the Query Rewrite system itself.
Predicate Pushdown and Pullup

Introduction

The justification for Predicate Pushdown is to move operations as close to the beginning of query processing as possible in order to eliminate as many rows (using the relational algebra SELECT operator) and columns (using the relational algebra PROJECT operator) as possible to reduce the cost of the query to a minimal value. The act of pushing projections is sometimes referred to as a semijoin. 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 (The concept of parse trees is described in “Translation to Internal Representation” on page 134, as is Predicate Pushdown).

Pushing predicates enhances performance by reducing the cardinality of spool files for views and derived tables that cannot be folded.

Predicate Pullup is less frequently used than pushdown in query rewrite and 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 after other query rewrites have been done. Because query processing begins at the bottom of the parse tree, these predicates are said to be pulled up the tree.

This rewrite method pushes predicates from containing query blocks into views or derived tables whenever possible.

Examples

Suppose you have the following request:

```sql
SELECT MAX(total)
FROM (SELECT product_key, product_name,
    SUM(quantity*amount) AS total
FROM Sales, Product
WHERE sales_product_key=product_key
GROUP BY product_key, product_name) AS v
WHERE product_key IN (10, 20, 30);
```

Predicate Pushdown can move the outer WHERE clause predicate for this request into the derived table \( v \), and then evaluate it as part of the WHERE clause for \( v \).

Pushing predicates helps performance by reducing the number of rows in spool files for views and derived tables that cannot be folded.

A significant component of Predicate Pushdown is devoted to pushing conditions into spooled views or spooled derived tables. This rewrite analyzes conditions that reference a view, maps those conditions to the base tables of the view definition, and then appends them to the view definition. Such a rewrite has the potential to improve the materialization of that view.

19. See “Logical Operations on Relations” in Database Design for more information about these relational algebra operators.
For example, consider the derived table $dt(x,y,z)$ in the following query:

```sql
SELECT *
FROM (SELECT *
      FROM t1) AS dt(x,y,z)
WHERE x=1;
```

If the derived table in the query is spooled, then the condition $a1=1$ can be pushed into it. The rewrite produces the following query:

```sql
SELECT *
FROM (SELECT *
      FROM t1
      WHERE a1=1) AS dt(x,y,z)
WHERE x=1;
```

Note that the original query would have required a full-table scan of $t1$, while the rewritten query requires only a single-AMP scan of $t1$.

Consider another view definition and a specific query against it:

```sql
CREATE VIEW v (a, b, c) AS
  SELECT a1, a2, SUM(a3)
  FROM   t1, t2, t3
  WHERE  b1=b2
  AND    c2=c3
  GROUP BY a1, a2;

SELECT v.a, v.b
FROM v, t4
WHERE v.a=a4 ;
```

Because view column $v.c$ is not referenced by the containing query block, the $\text{SUM}(a3)$ term can be removed from the rewritten select list of the view definition for this request. This action reduces the size the spool file for the view (assuming the view is spooled) as well as eliminating the unnecessary computation of the aggregate term $\text{SUM}(a3)$.

Pushing projections can enable other rewrites as well. For example, consider the following table definitions:

```sql
CREATE TABLE t1 (a1 INTEGER NOT NULL,
                 b1 INTEGER,
                 PRIMARY KEY (a1));

CREATE TABLE t2 (a2 INTEGER,
                 b2 INTEGER,
                 FOREIGN KEY (a2) REFERENCES t1);
```

Join Elimination can be applied to the following request if the references to $t1$ are removed from the select list of the derived table:

```sql
SELECT 1
FROM (SELECT *
      FROM t1,t2
      WHERE a1=a2) AS dt;
```
Eliminating Set Operation Branches

This rewrite looks for branches of set operations that contain unsatisfiable conditions, which are usually derived by SAT-TC, and then removes those branches if it can.

Consider the following example:

```sql
SELECT * 
FROM sales1 
WHERE EXTRACT(MONTH FROM sales_date) = 1 
UNION ALL 
SELECT * 
FROM sales2 
WHERE EXTRACT(MONTH FROM sales_date) = 1 ;
```

The second branch of the UNION ALL in this example is unsatisfiable because `sales2` only contains rows where the value for month equals 2 (a CHECK constraint). Therefore, the query can be rewritten as follows:

```sql
SELECT * 
FROM sales1 
WHERE EXTRACT(MONTH FROM sales_date) = 1 ;
```

When branches of UNION operations, as opposed to UNION ALL operations, are eliminated, the rewrite adds a DISTINCT to the remaining branch if necessary to guarantee correct results. Unsatisfiable branches of INTERSECT and MINUS operations are also eliminated by this rewrite technique.
Eliminating Redundant Joins

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. Join Elimination removes unnecessary tables from queries. Join Elimination is most commonly used in the following situations:

- Inner joins based on any form of referential integrity between two tables.
- Left and right outer joins can be eliminated if the join is based on unique columns from the right table in the join specification.

In both cases, a join and a table are removed from the query, assuming that no projections are needed from the eliminated table to keep the query whole.

Because of its many optimized join algorithms, Teradata Database 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 Optimizer recognizes and eliminates unnecessary or redundant joins is provided in "Outer Join Case Study" and its subordinate topics in SQL Data Manipulation Language. The topics "First Attempt" and "Third Attempt" are particularly good examples of how the Optimizer can streamline poorly written joins.

Primary key-foreign key relationships between tables are very common in normalized data models and their corresponding physical databases. Define the parent tables in such a relationship as PK-tables and the child tables as FK-tables. In Teradata Database, you can define these relationships explicitly by specifying referential integrity constraints when you create or alter tables (see “CREATE TABLE” and “ALTER TABLE” in SQL Data Definition Language. Also see the chapter “Designing for Database Integrity” in Database Design).

It also common to query PK-tables and FK-tables with joins based on their primary key and foreign key column sets, which are referred to as PK-FK joins. Sometimes the PK-FK joins are redundant, as, for example, when a query does not reference columns from the PK-table other than the PK-column set itself. Recognizing and eliminating such redundant joins can significantly reduce query execution times.

Another application where you might want to implement referential integrity constraints between two tables is the vertical partitioning of a single wide table. Vertical partitioning means that a table with a very large number of columns, many of which are seldom queried, is split into two tables where one table retains the frequently accessed columns and the other table has the rest. To ensure consistency between the tables, thereby maintaining a robustly consistent virtual table that contains correct data in all of the columns, a referential integrity constraint is needed between the tables. The choice of the PK-table and the FK-table in this case is arbitrary, but the best practice is to define the table with the frequently accessed columns as the FK-table.

20. The table eliminated is the parent table in inner joins and the inner table in outer joins.
To hide this artificial partitioning, you can define a view that selects all of the columns from both tables using a PK-FK join. When a request is submitted against the view that only references columns from the FK-table, its join is obviously redundant and can be eliminated. Note that without the intervention of Query Rewrite, you have no way of removing the join.

The following request illustrates redundant joins.

```
SELECT s_suppkey, s_address, n_nationkey
FROM supplier, nation
WHERE s_nationkey=n_nationkey
ORDER BY n_nationkey;
```

The join is redundant in this request because every row in `supplier` has exactly one match in `nation` based on the referential integrity definition). Also, only `n_nationkey` is referenced in the request, and it can be replaced by `s_nationkey`. The modified query after Join Elimination looks like this:

```
SELECT s_suppkey, s_address, s_nationkey
FROM supplier
ORDER BY s_nationkey;
```

The following conditions are sufficient to eliminate a PK-FK join and, of course, the PK-table, and all are checked by this rewrite method:

- There is a referential integrity constraint defined between the two tables.
- The query conditions are conjunctive.
- No columns from the PK-table other than the PK-columns are referenced in the request. This includes the SELECT, WHERE, GROUP BY, HAVING, ORDER BY, and other clauses.
- The PK-columns in the WHERE clause can only appear in PK=FK (equality) joins.

If these four conditions are met, then the PK-table and PK-FK join can be removed from the request and all references to the PK columns in the query are mapped to the corresponding FK columns. Also, Query Rewrite adds a NOT NULL condition on the FK columns if they are nullable. This optimization is applied as a rule, so no costing is required. Also, Query Rewrite does these optimizations automatically. To trigger this optimization, you must define a referential integrity constraint between the child and parent tables.

Be aware that the cost of Referential Integrity maintenance can exceed the benefit of Join Elimination. To mitigate this possibility, you can substitute either a batch Referential Integrity constraint or a Referential Constraint (see “CREATE TABLE” in SQL Data Definition Language and the chapter “Designing for Database Integrity” in Database Design for details of these Referential Integrity types).
The following table outlines the differences between these two referential integrity mechanisms.

<table>
<thead>
<tr>
<th>Referential Integrity Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch Constraint</td>
<td>Referential integrity is enforced between the tables in the relationship. The cost of maintaining a Batch RI constraint is normally significantly less than that of maintaining the same relationship with a standard Referential Integrity constraint.</td>
</tr>
<tr>
<td>Referential Constraint</td>
<td>Referential integrity is not enforced between the tables in the relationship. The constraint is used mainly as a hint for Join Elimination. Generally speaking, you should only specify a Referential Constraint when you are confident that the parent-child relationship is consistent (meaning that there is little or no possibility for referential integrity violations to occur).</td>
</tr>
</tbody>
</table>

Also consider the following example:

SELECT sales1.*
FROM sales1, product
WHERE sales_product_key=product_key;

The table product can safely be removed from the query because it is a dimension table of sales1 and the only reference to it is by means of a PK-FK join with sales1. This logic is also applied to remove the inner table of an OUTER JOIN if the join condition is an equality condition on a unique column of the inner table.

The following example illustrates OUTER JOIN elimination:

SELECT sales1.*
FROM sales1 LEFT OUTER JOIN product
ON sales_product_key=product_key;
Pushing Joins Into UNION ALL Branches

Representing a fact table as a UNION ALL view of horizontal partitions of fact table rows is a commonly used technique. These views are then often joined to constrained dimension tables as shown in the following example view and query against that view:

```sql
CREATE VIEW jan_feb_sales AS
SELECT *
FROM sales1
UNION ALL
SELECT *
FROM sales2;

SELECT SUM(quantity*amount) AS total
FROM jan_feb_sales, product
WHERE sales_product_key=product_key
AND product_name LIKE 'French%';
```

In this case, the join with the constrained dimension table can be pushed into each branch of the UNION ALL. The rewrite of this example is as follows:

```sql
SELECT sum(quantity*amount) AS total
FROM (SELECT quantity, amount
      FROM sales1, product
      WHERE sales_product_key=product_key
      AND product_name LIKE 'French%'
      UNION ALL
      SELECT quantity, amount
      FROM sales2, product
      WHERE sales_product_key=product_key
      AND product_name LIKE 'French%') AS jan_feb_sales;
```

The rewritten query can reduce the size of the spool file for the view by using the constrained join to filter rows from `sales1` and `sales2` before writing the spool file. This rewrite is cost-based (see “Cost Optimization” on page 267), so the Join Planner is called by the Query Rewrite Subsystem to determine whether the original or rewritten version of a request can be executed more efficiently.
Other Query Rewrites

Not all query rewrites are performed by the Query Rewrite subsystem. Some, such as the Partial GROUP BY optimization (see “Partial GROUP BY Block Optimization” on page 353), are performed as late in the optimization process as join planning. The following topics briefly describe some of the rewrites and the parse tree transformations that are done by the Optimizer after the Query Rewrite subsystem has completed its tasks.
Predicate Marshaling

The first step in the process of query rewrites undertaken by the Optimizer is to marshal the predicates, both ordinary and connecting, for the DML operations presented to the Optimizer by the rewritten ResTree. This means that first the query predicates are, whenever possible, isolated from the parse tree, converted to conjunctive normal form, or CNF (see “Path Selection” on page 268 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,

\[ A + CD \rightarrow (A+C)(A+D) \]
\[ AB + CD \rightarrow (A+C)(A+D)(B+C)(B+D) \]

where the symbol \( \rightarrow \) indicates transformation.

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.

\[ (\text{NUSI}_1 = 1 \text{ AND } \text{NUSI}_2 = 2) \text{ OR } (\text{NUSI}_1 = 3 \text{ AND } \text{NUSI}_2 = 4) \]

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. For example,

\[ A = B \text{ AND } A = C \rightarrow B = C \]
\[ A = 5 \text{ AND } A = B \rightarrow B = 5 \]
\[ A = 5 \text{ AND } B = 5 \rightarrow A = B \]

21. Note that this rewrite is not undertaken by the Query Rewrite subsystem.

22. Transitive closure is a relational algebraic method used to solve questions of the sort "can I get to point b from point a?" This is sometimes referred to as Reachability.

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 request implies that \( x < 1 \).

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

Similarly, the following SELECT request 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.

See "Satisfiability and Transitive Closure" on page 102.
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.

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

\[
\text{(table}_1.x, \text{table}_1.y) \text{ IN (SELECT table}_2.a, \text{table}_2.b \\
\text{ FROM table}_2) \\
\rightarrow \text{(table}_1.x \text{ IN spool}_1.a) \text{ AND (table}_1.y \text{ IN spool}_1.b)
\]

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

\[
\text{(table}_1.x, \text{constant}) \text{ IN (SELECT table}_2.a, \text{table}_2.b \\
\text{ FROM table}_2) \\
\rightarrow \text{(table}_1.x \text{ IN spool}_1.a)
\]

The term \((\text{table}_2.b = \text{constant})\) is pushed to \(\text{spool}_1\).

The following transformation is more complex.

\[
\text{(table}_1.x, \text{table}_1.y) \text{ IN (SELECT table}_2.a, \text{constant} \\
\text{ FROM table}_2) \\
\rightarrow \text{(table}_1.x \text{ IN spool}_1.a) \text{ AND (table}_1.y = \text{spool}_1.\text{constant})
\]

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 134 and “Predicate Pushdown and Pullup” on page 115 for additional information on predicate push down and pullup.
View Materialization and Other Database Object Substitutions

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

For example, all views referenced by a query must be resolved into their underlying base tables before the query can be performed. These rewrite methods also replace 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 replaced profitably, Partial GROUP BY rewrites, and Common Spool Usage rewrites.

For various reasons, these query rewrites are performed by the Optimizer subsystem after the Query Rewrite subsystem has completed its various rewrite tasks.

23. The term materialized view is sometimes used to describe database objects like snapshot summary tables, and hash or join indexes (see Gupta and Mumick, 1999). For purposes of the current discussion, the term refers to materializing the base table components of a view definition in a spool relation.
Chapter 2: Query Rewrite and Optimization

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.

The fundamental task of the Optimizer is to produce the most efficient access and execution plan for retrieving the data that satisfies an SQL query. This query plan determines the steps, the order, the parallelism, and the data access methods that can most efficiently deliver the result for a given SQL request.

Determining the best query plan depends on numerous factors, including all of the following:

- Physical implementation of the database
- Current interval histogram statistics
- System configuration and costing formulas
- Column and index set cardinalities
- Algorithms to generate interesting plan alternatives
- Heuristics to limit the size of the search space

Some of these factors are predetermined algorithms and heuristics inside the Optimizer and beyond your control, but you can control and manipulate other factors by modifying the physical database design and by fine tuning your queries. Among the user-controllable factors are table layout, the choice of whether a table has a primary index or not, the primary index column set and whether it is partitioned, the selection and type of secondary, join, and hash indexes, and the availability and accuracy of column and index set statistics.

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 the structure of a database and how to access its records, respectively. 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 several transformations on SQL queries. The following graphic is a very high-level representation of an SQL query optimizer:

![Diagram of a query optimizer]

The input to the Optimizer, labelled as *Input SQL Query* in the graphic, is actually the ResTree\(^{24}\) output from Query Rewrite, so by this point the original SQL query text has already been reformulated as an annotated parse tree.

The *Plan Exploration Space* is a workspace where various query and join plans and subplans are generated and costed (see “Bushy Search Trees” on page 327 and “Possible Join Orders as a Function of the Number of Relations To Be Joined” on page 330).

The *Costing Engine* compares the costs of the various plans and subplans generated in the Plan Exploration Space and returns the least costly plan from a given evaluation set (see “Cost Optimization” on page 267).

The Costing Engine primarily uses cardinality estimates based on the collected statistical information on the set of relations being evaluated to produce its choice of a given “best” plan. The statistical data used to make these evaluations is contained within a set of synoptic data structures called *histograms* that are maintained in the dictionary (see “Optimizer Statistics” on page 140).

The *Heuristics Engine* is a set of rules and guidelines used to evaluate plans and subplans in those situations where cost alone cannot determine the best plan.

24. Sometimes called *Red Tree*.'
Given this framework from which to work, Query Rewrite and the Optimizer performs the following actions in, roughly, the following order:

1. Translates an SQL query into some internal representation.
2. Rewrites the translation into a canonical form (the conversion to canonical form is often referred to as Query Rewrite\(^{25}\)).
3. Assesses and evaluates candidate procedures for accessing and joining database tables. Join plans have three additional components:
   - Selecting a join method.
     There are often several possible methods that can be used to make the same join. For example, it is usually, but not always, less expensive to use a Merge Join rather than a Product Join. The choice of a method often has a major effect on the overall cost of processing a query.
   - Determining an optimal join geography.
     Different methods of relocating rows to be joined can have very different costs. For example, depending on the size of the tables in a join operation, it might be less costly to duplicate one of the tables rather than redistributing it.
   - Determining an optimal join order.
     Only two tables can be joined at a time. The sequence in which table pairs\(^{26}\) are joined can have a powerful impact on join cost.
4. Generates several candidate query plans\(^{27}\) and selects the least costly plan from the generated set for execution.
5. End of process.

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

---

25. More accurately, Query Rewrite is a processing step that precedes the query processing steps that constitute the canonical process of query optimization. See “Query Rewrite” on page 74 for more information about Query Rewrite.

26. In this context, a table could be joined with a spool file rather than another table. The term table is used in the most generic sense of the word. Both tables and spool files are frequently categorized as relations when discussing query optimization.

27. 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” on page 271 and “Statistics And Cost Estimation” on page 293 for a description of query plan costing).
Parallels With Compiler Optimization

The parallels between SQL query optimization and compiler code 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. 28

The following table indicates how these criteria generalize to the optimization of SQL requests:

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

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 platform 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.

28. Or, as the old folk aphorism phrases it, “the juice should be worth the squeeze.” If the probability of gaining a less costly plan from exploring a larger plan space is small, then an intensive exploration of that plan space rarely pays off in a plan that is significantly better than one achieved in less time by exploring a more restricted, but also more likely to produce an optimal plan, plan space. Over the long haul, even with an occasional big payoff, this adds up to a great deal of wasted optimization time that could be put to better use.
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 Database query optimizer is cost-based.29

---

29. Cost-based query optimizers also use rules, or heuristics, in certain situations where cost alone cannot determine the best plan from the set being evaluated, but the primary method of determining optimal query and join plans is costing. Heuristics are also used to prune certain categories or particular branches of join order search trees from the join order evaluation space because experience indicates that they rarely yield optimal join orders, so they are not considered to be worth the effort it would take to evaluate them.
What are the specific costs that a cost-based optimizer seeks to minimize? The 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. An operation that takes the least time is, by definition, an operation that has the least cost to perform.

The plans generated by the 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 the following:

- To find a superior strategy from a set of candidate strategies.
- To avoid strategies that are obviously poor.
Teradata Database Optimizer Processes

Introduction

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

Query Optimization Processes

The following processes list the logical sequence of the processes undertaken by the Optimizer as it optimizes a DML request. The processes do not include the influence of Request Cache peeking to determine whether the Optimizer should generate a specific plan or a generic plan for a given request (see “Peeking at the Request Cache” on page 36).

The input to the Optimizer is the Query Rewrite ResTree’ (see “Query Rewrite” on page 74). The Optimizer then produces the optimized white tree, which it passes to the Generator (see “Generator” on page 66).

The Optimizer engages in the following process stages:

1. Receive the Query Rewrite ResTree’ as input.
2. Process correlated subqueries by converting them to nonnested SELECTs or simple joins.
3. Search for a relevant join or hash index.
4. Materialize subqueries to spool files.
5. Analyze the materialized subqueries for optimization possibilities.
   a. Separate conditions from one another (see “Predicate Marshaling” on page 123).
   b. Push down predicates (see “Predicate Pushdown and Pullup” on page 115).
   c. Generate connection information.
   d. Locate any complex joins.
   e. End of subprocess.
6. Generate size and content estimates of spool files required for further processing (see “How the Optimizer Uses Statistical Profiles” on page 193).
7. Generate an optimal single table access path.
8. Simplify and optimize any complex joins identified in stage 5d.
9. 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.
10 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 term: 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

11 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.

12 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 349).

13 If the join plan identified in step 12 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 352).

14 Select the better plan of the two plans generated in steps 12 and 13.

15 Generate a star join plan (see “Star and Snowflake Join Optimization” on page 442).

16 Select the better plan of the selection in step 14 and the star join plan generated in stage 15.

17 Pass the optimized white tree to the Generator.

18 End of process.

The Generator (see “Generator” on page 66) then generates plastic steps for the plan chosen in step 16.
Translation to Internal Representation

Parse Tree Representations of an SQL Request

Like most cost-based optimizers, the Optimizer represents SQL requests 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 requests, and they form the nodes and leaves 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:

30. See any introductory text on graph theory for a review of trees and cyclicity, for example Chartrand (1984), Bollobás (1998), Even (1979), or Trudeau (1994).
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:

<table>
<thead>
<tr>
<th>Parts</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PartNum</td>
<td>Description</td>
</tr>
<tr>
<td>PK</td>
<td></td>
</tr>
<tr>
<td>PI</td>
<td></td>
</tr>
<tr>
<td>ND, NN</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustNum</td>
<td>CustName</td>
</tr>
<tr>
<td>PK</td>
<td></td>
</tr>
<tr>
<td>PI</td>
<td></td>
</tr>
<tr>
<td>ND, NN</td>
<td></td>
</tr>
</tbody>
</table>

Example SQL Request

Consider the following example SQL request:

```
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 request into a parse tree is performed by the Syntaxer after it finishes checking the query text for syntax errors. Query Rewrite receives a processed parse tree as input from the resolver, then produces a rewritten, but semantically identical, parse tree as output to the Optimizer. This request tree is just a parse tree representation of the original query text.

The Optimizer further transforms the tree by 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 request tree has been discarded and replaced by an entirely new parse tree that contains instructions for performing the DML request. The parse tree is now an operation tree. It is a textual form of this tree, also referred to as a white tree, that the Optimizer passes to you as EXPLAIN text when you explain a request.\(^\text{31}\)

---

31. Note that a separate subsystem adds additional costing information about operations the Optimizer does not cost to the white tree before any EXPLAIN text is produced for output. See "EXPLAIN and Teradata Workload Manager" on page 485 for further information.
Assume the Optimizer is passed the following simplified parse tree by Query Rewrite (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).

```
PROJECT PartNum

SELECT OrderDate < 01-01-2001

PROJECT PartNum, MfgPartNum, CustAddress, CustName, CustCity, CustNum, OrderDate

```

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 process enumeration. 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.
The first set of processes is performed by Query Rewrite:

1. Split the compound ANDed condition into separate predicates. The result is the following pair of SELECT operations.

   \[
   \begin{align*}
   &\text{SELECT Customer.CustNum = Order.CustNum} \\
   &\text{SELECT Parts.MfgPartNum = Order.MfgPartNum}
   \end{align*}
   \]

2. By commutativity, the SELECTION \(\text{Order.OrderDate < 2001-01-01}\) 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.

   a. Begin with the following predicate.

   \[
   \text{SELECT OrderDate < 2001-01-01((Order X Customer) X Parts)}
   \]

   b. Transform it to the following form.

   \[
   \text{SELECT (OrderDate < 2001-01-01(Order X Customer)) X Parts}
   \]

   c. Transform it further to the following form.

   \[
   \text{SELECT ((OrderDate < 2001-01-01(Order)) X Customer) X Parts}
   \]

   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.

   d. End of subprocess.

3. The Optimizer examines the following SELECT operation to see if it can be pushed further down the parse tree.

   \[
   \text{SELECT Parts.MfgPartNum = Order.MfgPartNum}
   \]

   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.

4. The Optimizer examines the following SELECT operation to determine if it can be pushed any further down the parse tree.

   \[
   \text{SELECT Customer.CustNum = Order.CustNum}
   \]

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

   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.

5. 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:

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.

\[
\text{PROJECT Parts.PartNum} \\
\text{SELECT Parts.MfgPartNum = Order.MfgPartNum} \\
\text{PROJECT Parts.MfgPartNum, PartNum} \\
\text{PROJECT Order.MfgPartNum, Order.CustNum} \\
\text{SELECT Order.OrderDate < 01-01-2001} \\
\text{PROJECT CustNum} \\
\text{Customer Table} \\
\text{Order Table}
\]

7 Using the rules of commutation of a PROJECTION with a Cartesian product, replace the last PROJECTION in Stage 6 with the following PROJECTION.

\[
\text{PROJECT PartNum, Parts.MfgPartNum}
\]

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.

\[
\text{PROJECT Order.MfgPartNum} \\
\text{SELECT Customer.CustNum = Order.CustNum} \\
\text{PROJECT Order.MfgPartNum, Customer.CustNum, Order.CustNum}
\]

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.

![Parse Tree](https://via.placeholder.com/150)

11 End of process.

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 corresponds very roughly to 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.
Optimizer Statistics

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 table cardinality, which does not collect all the statistics a COLLECT STATISTICS request does.

Besides estimated cardinalities, random AMP samples also collect a few other statistics, but far fewer than are collected by a COLLECT STATISTICS request.

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?

Because query optimizers cannot dynamically reoptimize a suboptimal query plan once it has been generated, it is extremely important to have reasonably accurate statistical and demographic data about your tables at all times.

For example, if the statistics for a table are significantly at variance with its current state, the Optimizer might estimate the cardinality after one processing step as 10,000 rows, when the actual query returns only 15 rows at that step. The remaining steps in the query plan are all thrown off by the misestimation that led to the result of step 1, and the performance of the request is suboptimal. See “An Example of How Stale Statistics Can Produce a Poor Query Plan” on page 150 for a simple example of the extent to which a query plan can be affected negatively by bad statistical data. Ioannidis and Christodoulakis (1991) have demonstrated that bad statistics propagate estimation errors in join planning at an exponential rate.
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.

- For PPI tables, statistics collected on the PARTITION system-derived column permit the Optimizer to better estimate costs.

Teradata Database also uses PARTITION statistics for estimates when predicates are based on the PARTITION column, for example: `WHERE PARTITION IN (3, 4)`.

- The Teradata Statistics Wizard uses existing statistics, which it assumes are current by default, as a basis for recommending other statistics to be collected for a query or query workload (see “Special Considerations for Sampled and Stale Statistics” on page 668).

This is yet another reason you should keep your statistics as current as you possibly can.

In many cases, random all-AMPs (see “Sampled Statistics” on page 175) or random single-AMP sampled statistics (see “Random AMP Sampling” on page 178) 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. In most cases, the creation and propagation of derived statistics by the Optimizer can balance all but the most stale statistics (see “Derived Statistics” on page 206).

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.\(^\text{32}\).

\(^{32}\) 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.
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 estimates fewer statistics.

Unfortunately, this decision is not so easily made in a production environment. Other factors must be accounted for, including the length of 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:

<table>
<thead>
<tr>
<th>Level</th>
<th>Type of Optimization</th>
<th>Questions Asked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom</td>
<td>Query</td>
<td>If collecting full-table statistics makes queries run faster, what reasons could there be for collecting less accurate statistical samples?</td>
</tr>
<tr>
<td>Middle</td>
<td>Workload</td>
<td>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?</td>
</tr>
<tr>
<td>Top</td>
<td>Mix</td>
<td>What mix of query and workload optimization is best for overall system performance?</td>
</tr>
</tbody>
</table>
The following table compares the various characteristics of the three methods of collecting statistics and documents their respective most productive uses.

<table>
<thead>
<tr>
<th>Method</th>
<th>Characteristics</th>
<th>Best Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full statistics</td>
<td>• Collects all statistics for the data.</td>
<td>• Best choice for columns or indexes with highly skewed data values.</td>
</tr>
<tr>
<td></td>
<td>• Time consuming.</td>
<td>• Recommended for small tables; meaning, tables with fewer than 1,000 rows per AMP.</td>
</tr>
<tr>
<td></td>
<td>• Most accurate of the three methods of collecting statistics.</td>
<td>• Recommended for selection columns having a moderate to low number of distinct values.</td>
</tr>
<tr>
<td></td>
<td>• Stored in interval histograms in the data dictionary.</td>
<td>• Recommended for most NUSIs and other selection columns.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Collection time on NUSIs is very fast.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Recommended for all column sets or index column sets where full statistics add value, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where sampling does not provide satisfactory statistical estimates.</td>
</tr>
<tr>
<td>Sampled statistics</td>
<td>• Collects all statistics for the data, but not by accessing all rows in the table.</td>
<td>• Acceptable for columns or indexes that are highly singular; meaning that their number of</td>
</tr>
<tr>
<td></td>
<td>• Significantly faster collection time than full statistics.</td>
<td>distinct values approaches the cardinality of the table</td>
</tr>
<tr>
<td></td>
<td>• Stored in interval histograms in the data dictionary.</td>
<td>• Recommended for unique columns, unique indexes, and for columns or indexes that are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>highly singular.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Experience suggests that sampled statistics are useful for very large tables; meaning tables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>with tens of billions of rows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Not recommended for small tables; meaning tables whose cardinality is less than 20 times</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the number of AMPs in the system.</td>
</tr>
<tr>
<td>Random AMP Sample</td>
<td>• Estimates fewer statistics than COLLECT STATISTICS does.</td>
<td>• Good for cardinality estimates when there is little or no skew and the table has significantly more rows than the number of AMPs in the system.</td>
</tr>
<tr>
<td></td>
<td>Statistics estimated include the following for all columns:</td>
<td>• Collects reliable statistics for NUSI columns when there is limited skew and the table has</td>
</tr>
<tr>
<td></td>
<td>• Cardinalities</td>
<td>significantly more rows than the number of AMPs in the system.</td>
</tr>
<tr>
<td></td>
<td>• Average rows per value</td>
<td>• Collects reliable statistics for NUSI columns when there is limited skew and the table has</td>
</tr>
<tr>
<td></td>
<td>For indexes only, the following additional statistics are estimated:</td>
<td>significantly more rows than the number of AMPs in the system.</td>
</tr>
<tr>
<td></td>
<td>• Average rows per index</td>
<td>• Collects reliable statistics for NUSI columns when there is limited skew and the table has</td>
</tr>
<tr>
<td></td>
<td>• Average size of the index per AMP</td>
<td>significantly more rows than the number of AMPs in the system.</td>
</tr>
<tr>
<td></td>
<td>• Number of distinct values</td>
<td>• Collects reliable statistics for NUSI columns when there is limited skew and the table has</td>
</tr>
<tr>
<td></td>
<td>• Extremely fast collection time, so is not detectable.</td>
<td>• Collects reliable statistics for NUSI columns when there is limited skew and the table has</td>
</tr>
<tr>
<td></td>
<td>• Stored in the file system data block descriptor for the table, not in interval histograms in the data dictionary.</td>
<td>significantly more rows than the number of AMPs in the system.</td>
</tr>
</tbody>
</table>
How the Optimizer Determines Whether To Use Interval Histogram Statistics or a Random AMP Sample for Base Table Cardinality Estimates

Because the statistics contained in the interval histograms for a table (see “Interval Histograms” on page 161) can be so stale that a random AMP sample would produce more accurate cardinality estimates, the Optimizer bases its selection of which statistics to use based on the following set of rules. It uses a random AMP sample over the existing interval histogram statistics only when the following criteria are true:

- Either the sample is taken from all AMPs rather than a single AMP or the primary index is not skewed. Random AMP sampling for NoPI tables is always done across all AMPs.

and

- The table is not small. A table is defined to be not small if its interval histogram cardinality value is greater than the number of AMPs on the system.

and

- The sample cardinality is larger than the interval histogram cardinality.

and

- The deviation between the two cardinalities exceeds a user-specified percentage that you define in your Cost Profile.

The implication of this is that if the primary index for a large table has a fairly uniform distribution, you need not collect statistics on it as frequently as you otherwise would, because even if the statistics on the primary index are stale, you should still be able to obtain a reasonably good query plan.

33. Defined for purposes of this implication as a table having more rows than the system has AMPs.
Note that this does not relieve you of the responsibility of collecting or refreshing statistics for the following:

- Small tables
- Tables whose primary index distribution is skewed
- Whenever you delete large numbers of rows from a table
- Workloads that contain frequently submitted queries with single-table predicates on the primary index.

### Where the Optimizer Looks for Statistics When It Is Building a Query Plan

The following flowchart diagrams the process by which the Optimizer looks for statistics when it is creating a query plan for a request.
The general process stages the Optimizer follows while performing this task are as follows:

1. On receiving a parsed and rewritten request, the Optimizer looks for the data block descriptor (DBD) in the dictionary cache.

<table>
<thead>
<tr>
<th>IF the table DBD is ...</th>
<th>THEN the Optimizer ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>in the dictionary cache</td>
<td>extracts the random AMP samples it needs.</td>
</tr>
<tr>
<td>not in the dictionary cache</td>
<td>reads the DBD from disk, then extracts the random AMP samples it needs.</td>
</tr>
</tbody>
</table>

Each time a table header is read from disk, the system collects a fresh random AMP sample of its statistics and places it in the appropriate DBD, which is then cached along with the table header. Remember that the statistics derived from a random AMP sample are not as complete as those gathered by a COLLECT STATISTICS request and stored in interval histograms in the dictionary.

2. The Optimizer calls a routine that looks for collected statistics on each index or column set, or both, it needs whenever it requires statistics to build a query plan.

   a. The system searches the dictionary cache for the relevant interval histogram.

   b. If the desired interval histogram is not cached, the system issues an express request that retrieves it from the appropriate dictionary tables on disk.

The system stores the histogram data in the `DBC.TVFields` dictionary table.

3. If no statistics have been collected for the desired index or unindexed column set, proceed as outlined in the following table.

<table>
<thead>
<tr>
<th>IF the desired histogram is ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>located</td>
<td>place it in the dictionary cache.</td>
</tr>
<tr>
<td>not located</td>
<td>determine if the column set represents an index.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF the column set is an ...</th>
<th>THEN use ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexed column set</td>
<td>the random AMP sampled statistics from the DBD to estimate cardinalities.</td>
</tr>
<tr>
<td>unindexed column set</td>
<td>an appropriate heuristic to estimate cardinalities.</td>
</tr>
</tbody>
</table>
Purge the statistics from the dictionary cache when any of the following occur:

- The statistics have been in the cache for 4 hours.
- The demographics of the table have changed by more than 10 percent as the result of set-based row insertions or deletions.
  
  Note that cardinality changes incurred by individual INSERT or DELETE requests, no matter how extensive, are ignored in making this determination.
- A more recent process needs the cache space currently occupied by the DBD.
- The dictionary cache is spoiled for the table.

End of process.

How the Optimizer Determines Whether To Use Interval Histogram Statistics or a Random AMP Sample for NUSI Subtable Statistics

Because the statistics contained in the interval histograms for a NUSI subtable (see “Interval Histograms” on page 161) can be so stale that a random AMP sample would produce more accurate cardinality estimates, the Optimizer bases its selection of which statistics to use based on the following set of rules:

- The number of NUSI subtable index rows in an AMP is the number of distinct values in the index columns.
  
  The Optimizer assumes that all of the distinct values it samples from a single AMP exist on all of the AMPs in the system.
- If a NUSI has an interval histogram, and if that histogram has fewer distinct values than a random AMP sample gathers, then the system increases the number of index values to the number of values obtained from the random AMP sample, with the exception of the primary index.
  
  This counters the problem of stale NUSI statistics.
- The system sets the number of distinct values for unique indexes to the table cardinality by overriding the values from either the interval histogram or from a random AMP sample.

The implication of this is that you need not refresh the statistics for a NUSI each time new data is loaded into its base table as long as the distribution of the primary index is fairly uniform.

Note that this does not relieve you of the responsibility of collecting or refreshing NUSI statistics for the following:

- Tables having a high correlation between the primary index and NUSI column sets.
- Workloads that contain frequently submitted queries with single-table predicates.
  
  The Optimizer needs interval histograms to make accurate single-table cardinality estimates.
**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.  

---

34. You cannot gather sampled single-column PARTITION statistics. The system allows you to submit such a request, but it does not honor it. Instead, the sampling percentage is set to 100 percent. Sampled collection is permitted for multicolumn PARTITION statistics.
The elapsed time required to collect statistics varies as a function of the following factors:

- Base table cardinality
- Number of distinct index or nonindexed 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 and other applications. 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 never refreshing your statistics. You can do this by specifying the USING SAMPLE option of the COLLECT STATISTICS (Optimizer Form) statement (see SQL Data Definition Language). 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.

35. This is less critical than it once was because the system now has the capability to evaluate the relative usefulness of stale interval histogram statistics and random AMP sample statistics dynamically. See “How the Optimizer Determines Whether To Use Interval Histogram Statistics or a Random AMP Sample for Base Table Cardinality Estimates” on page 144 and “How the Optimizer Determines Whether To Use Interval Histogram Statistics or a Random AMP Sample for NUSI Subtable Statistics” on page 147 for details.
An Example of How Stale 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>
<thead>
<tr>
<th>Table Name</th>
<th>Statistics Collected?</th>
<th>Cardinality When Statistics Were Collected</th>
<th>Current Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Yes</td>
<td>1,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>B</td>
<td>No</td>
<td>unknown</td>
<td>75,000</td>
</tr>
</tbody>
</table>

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 that is three orders of magnitude larger than its cardinality at the time its statistics were collected: 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 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 request:

```
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-specified criteria from the Recommendations tab of its Options dialog box. 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 sets of user-specified workloads, and provides several other utilities you can use to perform various analyses of system statistics.

See Teradata Statistics Wizard User Guide and Chapter 7: “Database Foundations for the Teradata Index and Statistics Wizards” for additional information about the uses and capabilities of this utility.
How the AMP Software Collect Statistics

Introduction

When you collect statistics, the AMP software creates statistics rows that summarize the table columns specified by the COLLECT STATISTICS statement. These rows come in four basic types:

- Regular statistics rows (one or more per AMP)
- Null rows (zero or one per AMP)
- All-null rows (zero or one per AMP)
- Average rows per value (one per AMP)

The information taken from these rows is used to populate the statistical interval histograms used by the Optimizer to make its initial cardinality estimates (see “Interval Histograms” on page 161).

The following statistics are global per histogram, so are reported only once:

- Number of null rows
- Number of all-null rows
- Overall average of the average number of rows per value per AMP

Regular Row Statistics

So-called “regular” row statistics contain the cardinalities for each value in the column or index column set on which statistics are collected.

Each regular statistics row has three columns with the following values.

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ynet key</td>
<td>Cardinality for the value in column 3</td>
<td>Value</td>
</tr>
</tbody>
</table>

Each AMP has one or more regular statistics rows at the time AMP-local aggregation occurs.

Null and All-Null Statistics

Null and all-null statistics provide the following cardinalities:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumNulls</td>
<td>Number of rows in the column or index column set for which one or more of the column values are null.</td>
</tr>
<tr>
<td>NumAllNulls</td>
<td>Number of rows in a composite column set for which all of the column are set null.</td>
</tr>
</tbody>
</table>
Consider the following example of what is meant by null and all-null rows. Suppose you create a 3-column USI for table \( t_{ex} \) on columns \( b, c, \) and \( d \). When you collect statistics on the USI, one row is found in which all 3 columns making up the USI are null. This would be reported as an all-nulls instance in the statistics.\(^{36}\)

The columns for this row would look something like the following, where nulls are represented by the QUESTION MARK character:

\[
\begin{array}{cccccc}
  & a & b & c & d & e & f \\
\hline
  \text{PI} & \text{USI} & \text{PI} & \text{USI} & \text{PI} & \text{USI} & \text{PI} \text{USI} \\
  355 & ? & ? & ? & 25000.00 & 375000.00 \\
\end{array}
\]

This contrasts with the case where statistics have been collected on a composite column set, and one or more, but not necessarily all, of the columns on which those statistics have been collected for a given row are null.

Assume the same table and index definitions as before. When you collect statistics on the USI, any row found where one or two, but not all, of the three columns making up the USI are null would be reported as an occurrence of a null column in the statistics.

All of the following rows in \( t_{ex} \) would be reported as null fields, because all have at least one null in one of the USI columns, while only the first row would be reported for the all-null fields statistic:

\[
\begin{array}{cccccc}
  & a & b & c & d & e & f \\
\hline
  \text{PI} & \text{USI} & \text{PI} & \text{USI} & \text{PI} & \text{USI} \\
  355 & ? & ? & ? & 25000.00 & 375000.00 \\
  685 & ? & 155 & ? & 45000.00 & 495000.00 \\
  900 & 325 & ? & ? & 32000.00 & 400000.00 \\
  275 & ? & ? & 760 & 55000.00 & 575000.00 \\
  597 & ? & 254 & 893 & 10000.00 & 150000.00 \\
  322 & 679 & ? & 204 & 75000.00 & 650000.00 \\
  781 & 891 & 357 & ? & 32000.00 & 400000.00 \\
\end{array}
\]

All-nulls describes the case where statistics have been collected on a composite column set, and all the columns on which those statistics have been collected for a given row are null. With this information, the Optimizer can more accurately estimate the true number of unique values in a set, thus enabling more accurate join costing.

\(^{36}\) This example is degenerate because there could never be more than one row having a USI with all null columns in a table, but it is useful as an example, even though it does not generalize.
For example, suppose you have the table \( t1 \), with columns \( x1 \) and \( y1 \) as follows:

\[
\begin{array}{cc}
\hline
\text{Column 1} & \text{Column 2} & \text{Column 3} \\
0 & \text{Cardinality for the number of rows having one or more columns null.} & \text{Null} \\
\hline
\end{array}
\]

If you could collect only the NumNulls statistic, then when you collected statistics on composite columns, and one or more of the columns was null, the row would be counted as a null row. This is not an issue for single-table cardinality estimation because a comparison against a null is evaluated as FALSE and is treated as an unmatched row.

For example, if you could collect only the NumNulls statistic, the histogram on \((x1, y1)\) would indicate that the number of nulls is 4, the number of unique values is 2, and the total number of rows is 5. If columns \( x \) and \( y \) are used as join columns, the Optimizer would then evaluate the join costing by incorrectly assuming there are only 2 unique values in the data when there are actually 4. This can cause problems in scenarios such as redistribution costs, skew detection, and the like.

However, because both the NumNulls and NumAllNulls statistics are collected when you submit a COLLECT STATISTICS request, the Optimizer is able to correctly identify the true number of unique values in a value set, which makes it possible to do more accurate costing estimates.

For this particular case, the true number of unique values is calculated as follows:

\[
\text{NumUniqueValues} = \text{NumValues} + \text{NumNulls} - \text{NumAllNulls} = 2 + 4 - 2 = 4
\]

A NumNulls row has three columns with the following values.
A NumAllNulls row has three columns with the following values.

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Negative cardinality for the number of rows having all columns null.</td>
<td>Null</td>
</tr>
</tbody>
</table>

The following table shows how each individual AMP aggregates its regular, null, and all-null statistics rows. There are one or more regular rows, zero or one null row, and zero or one all-null row per AMP.

<table>
<thead>
<tr>
<th>Row Type</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-Null</td>
<td>0</td>
<td>Negative cardinality for rows with all columns null</td>
<td>Null</td>
</tr>
<tr>
<td>Null</td>
<td>0</td>
<td>Cardinality for rows with one or more columns null</td>
<td>Null</td>
</tr>
<tr>
<td>Regular</td>
<td>Ynet key</td>
<td>Cardinality</td>
<td>Value</td>
</tr>
<tr>
<td>...</td>
<td>Ynet key</td>
<td>Cardinality</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Ynet key</td>
<td>Cardinality</td>
<td>Value</td>
</tr>
</tbody>
</table>

The AMP-locally aggregated statistics rows are sorted and merged based on their Column 1 values to produce the global aggregation. Aggregation rows are then further processed by to combine rows from different AMPs, finally to yield the summarized counts in the interval histograms in the Dictionary.
Consider the following example as an illustration:

```
SELECT *
FROM demo_table;
```

```
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>8</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
SELECT HASHAMP(HASHBUCKET(HASHROW(x))) AS ampno, x, y, z
FROM demo_table
GROUP BY 1,2,3,4
ORDER BY 1;
```

```
<table>
<thead>
<tr>
<th>ampno</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0555</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0611</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02?2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03??</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0655</td>
<td>18??</td>
<td>17??</td>
<td>144?</td>
</tr>
</tbody>
</table>
```

Now submit a COLLECT STATISTICS request on the composite column set (y, z):

```
COLLECT STATISTICS demo_table COLUMN (y, z);
```

The hash function indicates what the initial aggregation looks like after the COLLECT STATISTICS request.

At the level of local aggregation, the following statistics rows exist on a two-AMP system: Column 1 is the Ynet key, which is produced from the corresponding value in Column 3, Column 2 represents the cardinalities, and Column 3 represents values from the base table. Notice that null, including all-null, rows are not merged, despite having identical Ynet key values of zero. The regular rows are merged if their hash values are identical on the same AMP.

To distinguish the all-nulls row count from the nulls row count, the system negates the count in Column 2 for the all-null rows.
After redistribution, the configuration changes to the following:

**AMP 0**

<table>
<thead>
<tr>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ynet(5,5)</td>
<td>+2</td>
<td>(5,5)</td>
</tr>
</tbody>
</table>

**AMP 1**

<table>
<thead>
<tr>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-2</td>
<td>all-null</td>
</tr>
<tr>
<td>0</td>
<td>+3</td>
<td>null</td>
</tr>
</tbody>
</table>

The system then sums all the null and all-null rows to create the results, as you can see in the HELP STATISTICS report:

- NumNulls : 5
- NumAllNulls : 3

## Average AMP Rows Per Value Statistic

The Average AMP Rows per Value statistic is a system-wide exact average of the average rows per value for each individual AMP over the number of rows per value for a NUSI column set on which statistics have been collected. This statistic is computed only for NUSI columns and is used for nested join costing.
The following table shows how each individual AMP aggregates its regular, null, all-null, and average AMP rows per value statistics rows. There are one or more regular rows, zero or one null rows, zero or one all-null rows, and one average rows per value row per AMP.

<table>
<thead>
<tr>
<th>Row Type</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average AMP RPV</td>
<td>0</td>
<td>Local Average RPV</td>
<td>0</td>
</tr>
<tr>
<td>All-Null</td>
<td>0</td>
<td>Negative cardinality</td>
<td>Null</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for rows with all</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>columns null</td>
<td></td>
</tr>
<tr>
<td>Null</td>
<td>0</td>
<td>Cardinality for rows</td>
<td>Null</td>
</tr>
<tr>
<td></td>
<td></td>
<td>with one or more</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>columns null</td>
<td></td>
</tr>
<tr>
<td>Regular</td>
<td>Ynet key</td>
<td>Cardinality</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>Ynet key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ynet key</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that when the column set is not a NUSI, the value for Average AMP RPV is always reported as 0. The following example illustrates the Average AMP RPV statistic:

First create the table on which statistics are to be collected.

```sql
CREATE TABLE demo_table (
  x INTEGER,
  y INTEGER,
  z INTEGER
) PRIMARY INDEX (x)
INDEX (y, z);
```

Notice the NUSI on columns y and z.

Now populate the table with values. There are no nulls in any of the resulting columns.

```sql
INSERT INTO demo_table VALUES (1,1,1);
INSERT INTO demo_table VALUES (2,2,1);
INSERT INTO demo_table VALUES (3,2,1);
INSERT INTO demo_table VALUES (4,3,8);
INSERT INTO demo_table VALUES (5,3,8);
INSERT INTO demo_table VALUES (6,3,8);
INSERT INTO demo_table VALUES (7,6,7);
INSERT INTO demo_table VALUES (8,6,7);
INSERT INTO demo_table VALUES (9,6,7);
INSERT INTO demo_table VALUES (10,6,7);
```
The following query reports the distribution of these rows across the 2 AMPs in the system, AMP0 and AMP1.

```sql
SELECT HASHAMP(HASHBUCKET(HASHROW(x))) AS ampno, x, y, z
FROM demo_table
GROUP BY 1,2,3,4
ORDER BY 1;
```

<table>
<thead>
<tr>
<th>ampno</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Now count the rows according to their value distribution in AMP0 and AMP1:

<table>
<thead>
<tr>
<th>AMP Number</th>
<th>Value of (y,z)</th>
<th>Number of Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(2,1)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(6,7)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(3,8)</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>(6,7)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(3,8)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(1,1)</td>
<td>1</td>
</tr>
</tbody>
</table>

From the distribution, it is possible to calculate the average rows per value for each AMP.

The values for each AMP are:

\[
\text{Avg RPV}_{\text{AMP0}} = \frac{(2+1+2)}{3} = 1.67
\]

\[
\text{Avg RPV}_{\text{AMP1}} = \frac{(3+1+1)}{3} = 1.67
\]
At the local aggregation stage of statistics collection, the following statistics rows exist on a two-AMP system according to the hash function:

<table>
<thead>
<tr>
<th>AMP Number</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1.67</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Ynet(2,1)</td>
<td>2</td>
<td>(2,1)</td>
</tr>
<tr>
<td></td>
<td>Ynet(6,7)</td>
<td>1</td>
<td>(6,7)</td>
</tr>
<tr>
<td></td>
<td>Ynet(3,8)</td>
<td>2</td>
<td>(3,8)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1.67</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Ynet(6,7)</td>
<td>3</td>
<td>(6,7)</td>
</tr>
<tr>
<td></td>
<td>Ynet(3,8)</td>
<td>1</td>
<td>(3,8)</td>
</tr>
<tr>
<td></td>
<td>Ynet(1,1)</td>
<td>1</td>
<td>(1,1)</td>
</tr>
</tbody>
</table>

In abbreviated tabular form, the average rows per value per AMP are as follows:

<table>
<thead>
<tr>
<th>AMP Number</th>
<th>Average Rows Per Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.67</td>
</tr>
<tr>
<td>1</td>
<td>1.67</td>
</tr>
</tbody>
</table>

The system-wide average AMP-local RPV is therefore the average of the two:

$$\text{Average AMP RPV} = \frac{\text{AvgRPV}_{\text{AMP0}} + \text{AvgRPV}_{\text{AMP1}}}{\text{Number of AMPS}} = \frac{1.67 + 1.67}{2} = 1.67$$
Interval Histograms

Introduction

A synopsis data structure is a data structure that is substantially smaller than the base data it represents. Interval histograms (Kooi, 1980) are a one dimensional form of synopsis data structure that provide a statistical and demographic profile of attribute values characterizing the properties of the raw data.

Generically defined, a histogram is a count of the number of occurrences, or cardinality, of a particular category of data that fall into defined disjunct categories. These categories are typically referred to as bins or buckets.

Formally, a histogram is defined as follows:

\[ N = \sum_{k=1}^{n} h_k \]

where:

<table>
<thead>
<tr>
<th>Equation element</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>the total number of occurrences of the measured data.</td>
</tr>
<tr>
<td>( n )</td>
<td>the total number of disjunct bins into which the data is segregated.</td>
</tr>
<tr>
<td>( h )</td>
<td>histogram.</td>
</tr>
<tr>
<td>( k )</td>
<td>an index over the histogram buckets.</td>
</tr>
</tbody>
</table>

Teradata Database 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. The greater the number of intervals in a histogram, the more accurately it can describe the distribution of data by characterizing a smaller percentage of its composition. Each interval histogram in the system is composed of a maximum of 200 intervals, which in a 200 interval histogram permits each interval to characterize roughly 0.5 percent of the data.\(^{37}\)

The statistical and demographic information maintained in the histograms is used to estimate various attributes of a query, most importantly the cardinalities of various aspects of the relations that are specified in the request.

Conceptually, the cardinality of each interval can be determined from the area under the curve of each interval in the histogram.

\(^{37}\) The number of intervals used to store statistics is a function of the number of distinct values in the column set represented. For example, if there are only 10 unique values in a column or index set, the system does not store the statistics for that column across 201 intervals, but across 11 (interval 0 plus the 200 - or 10 - additional intervals containing summary data for the distinct values in the set). The system employs the maximum number of intervals for a synopsis data structure only when the number of distinct values in the column or index set for which statistics are being captured equals or exceeds the maximum number of intervals.
The use of interval histograms to make these estimates is a nonparametric statistical method. Nonparametric statistical analyses are used whenever the population parameters of a variable, such as its distribution, are not known.

See Ioannidis (2003) for a brief history of histograms and their use in query optimization for relational database management systems.

Note that the system also uses derived statistics to estimate cardinalities and selectivities. These statistics are based on the initial values stored in the interval histograms, but are then adjusted for accuracy at each stage of query optimization by incorporating additional information such as CHECK and referential integrity constraints, query predicates, and hash and join indexes. See “Derived Statistics” on page 206 for details).

Terminology

The terms used to describe the intervals and histograms used by Teradata Database are defined in the following table (Also see “Glossary” on page 851).

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bin</td>
<td>See “Bucket.”</td>
</tr>
<tr>
<td>Bucket</td>
<td>A synonym for interval in an interval histogram.</td>
</tr>
<tr>
<td>Cardinality</td>
<td>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 Initial Cardinality Estimates Are Made for Simple Queries Using Interval Histograms” on page 197 for a fuller explanation of what cardinality means in this context.</td>
</tr>
<tr>
<td>Compressed interval histogram</td>
<td>A family of histograms that combines high-biased intervals and equal-height intervals with interval 0. More formally, a compressed interval histogram is an array of equal-height intervals or high-biased intervals or both. When both interval types are present, the high-biased intervals always precede the equal-height intervals. In the mixed intervals case, the high-biased intervals are stored in buckets 1 through $n$, where $n$ represents the highest numbered bucket containing high-biased interval information. Equal-height intervals begin at bucket $n + 1$ in a compressed interval histogram. A compressed interval histogram can have a maximum of 199 high-biased intervals.</td>
</tr>
<tr>
<td>Equal-height interval</td>
<td>An interval containing column statistics normalized across the distribution in such a way that the graph of the distribution of the number of rows as a function of interval number is flat. This is achieved by varying the width of each interval so it contains approximately the same number of rows (but with different attribute value ranges) as its neighbors. Also known as equal-depth intervals.</td>
</tr>
</tbody>
</table>
Equal-height interval histogram

A family of histograms characterized by approximately equal cardinalities and a variable attribute value range per bucket plus interval 0.

Also known as equal-depth interval histograms.

More formally, an equal-height histogram is an array of ordered equal-height intervals.

The definitions for some of the byte fields common to equal-height and high-biased interval histograms differ as indicated by the following table (see “Content and Storage of Histograms” on page 168 for details).

<table>
<thead>
<tr>
<th>This byte field</th>
<th>Is defined for equal-height interval histograms as the</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>total number of nonmodal values in the interval.</td>
</tr>
<tr>
<td></td>
<td>This value is always ≥ 0 for an equal-height histogram.</td>
</tr>
<tr>
<td>Mode</td>
<td>most frequent value in the interval.</td>
</tr>
<tr>
<td>Modal frequency</td>
<td>number of rows having the modal value.</td>
</tr>
<tr>
<td>Maximum value</td>
<td>maximum value covered by the interval.</td>
</tr>
<tr>
<td>Rows</td>
<td>total number of rows for the nonmodal values in the</td>
</tr>
<tr>
<td></td>
<td>interval.</td>
</tr>
</tbody>
</table>

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 165) is summarized by a high-biased interval.

Each high-biased interval contains statistics for at most two attribute row values.
Interval Histograms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-biased interval histogram</td>
<td>A family of histograms characterized by all loner buckets as seen, for example, with a multimodal or otherwise-skewed attribute value distribution. The definitions for some of the byte fields common to equal-height and high-biased interval histograms differ as indicated by the following table (see “Content and Storage of Histograms” on page 168 for details).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>This byte field ...</th>
<th>Is defined for high-biased interval histograms as the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>representation of the number of loners in the interval.</td>
</tr>
<tr>
<td></td>
<td><strong>WHEN an interval stores this many loners</strong> ...</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Mode</td>
<td>smaller (or only) loner value in the interval.</td>
</tr>
<tr>
<td>Modal frequency</td>
<td>number of rows having the smaller loner value in the interval.</td>
</tr>
<tr>
<td>Maximum value</td>
<td>larger loner value in a high-biased interval that stores two loners.</td>
</tr>
<tr>
<td></td>
<td><strong>The field is defined as the</strong> ...</td>
</tr>
<tr>
<td></td>
<td>larger loner</td>
</tr>
<tr>
<td></td>
<td>modal value</td>
</tr>
<tr>
<td>Rows</td>
<td>number of rows having the larger loner value in the interval.</td>
</tr>
<tr>
<td>Histogram</td>
<td>A graphic means for representing distributions as a function of the number of elements per an arbitrarily determined interval width.</td>
</tr>
<tr>
<td></td>
<td>Histograms are often called bar charts. Each bar in a histogram represents the number of rows 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.</td>
</tr>
<tr>
<td></td>
<td>In relational query optimization theory, the term is used to describe the rows in a dictionary table or system catalog that store the buckets defined by the particular intervals used to characterize the frequency distribution of the attribute values for a column set.</td>
</tr>
<tr>
<td></td>
<td>All histograms described in this topic are frequency histograms. Each bucket in a frequency histogram contains some number of tokens representing the number of rows that have the attribute values belonging to its range.</td>
</tr>
<tr>
<td>Interval</td>
<td>A bounded, nonoverlapping set of attribute value cardinalities.</td>
</tr>
</tbody>
</table>
Chapter 2: Query Rewrite and Optimization

Interval Histograms

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.

A loner is a distinct value that satisfies the following condition:

\[ f \geq \frac{T}{400} \]

where:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<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.</td>
</tr>
<tr>
<td>Skew</td>
<td>A measure of the asymmetry of the distribution of a set of attribute values or their cardinalities.</td>
</tr>
<tr>
<td>Synoptic data structure</td>
<td>A data structure that contains summary, or synopsis, metadata.</td>
</tr>
</tbody>
</table>

Skew 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 have only two values such as TRUE or FALSE.
- **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.

Skew is somewhat more likely to be seen with NoPI tables because of the way the system distributes their rows.

Furthermore, the rows inserted by a given request for a FastLoad or Teradata Parallel Data Pump ARRAY insert operation are always sent to the same AMP. As more rows are inserted by different requests, the distribution of rows eventually balances among the AMPs, but for small NoPI tables, where only a few requests are required to load all the rows, the distribution can be skewed.
Types of Interval Histograms Used By Teradata Database

Depending on the distribution of values (the degree of skew) in a column or index set, any one of three 38 varieties of histogram types is used to represent its statistics:

- **Equal-height interval histogram**
  The statistics for a column set are expressed as an equal-height interval histogram if none of the frequencies of its values are skewed.
  In an equal-height interval histogram (also referred to as an equal-depth interval histogram) each interval represents approximately the same number of rows having that range of values, making their ordinate, cardinality, an approximately constant value or height. If a row has a value for a column that is already represented in an interval, then it is counted in that interval, so some intervals may represent more rows than others.
  For example, suppose there are about one million rows in a table. Then there would be approximately 5,000 rows per interval, assuming a 200 interval histogram. Suppose that for a given interval, the system processes the values for 9,900 rows and the next value is present in 300 rows. Those rows would be counted in this interval, and the cardinality for the interval would 1,2000. Alternatively, the rows could be counted in the next interval, so this interval would still only represent 9,900 rows.
  A COLLECT STATISTICS request divides the rows in ranges of values such that each range has approximately the same number of rows, but it never splits rows with the same value across intervals. To achieve constant interval cardinalities, the interval widths, or value ranges, must vary.
  When a histogram contains 200 equal-height intervals, each interval effectively represents a half percentile score for the population of attribute values it represents.
  The following graphic illustrates the concept of an equal-height interval histogram. Note that the number of rows per equal-height interval is only approximately equal in practice, so the graphic is slightly misleading with respect to the precise equality of heights for all equal-height intervals in the histogram.

38. Actually, only two interval histogram types are ever used, equal-height and compressed. The system never uses pure high-biased interval histograms.
High-biased interval histogram

High-biased intervals are used to represent a column or index set only when there is significant skew in the frequency distribution of its values. The Optimizer does not maintain pure high-biased interval histograms. Instead, it mixes high-biased intervals with equal-height intervals in a compressed histogram whenever some values for a column or index set are skewed significantly.

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

<table>
<thead>
<tr>
<th>IF a high-biased interval stores this many loners</th>
<th>THEN the Values variable in that high-biased interval is represented by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
</tr>
</tbody>
</table>

Note that the number of rows represented by each loner is approximately half the number represented by an equal-height interval in most cases because a loner represents greater than $\frac{1}{400}$, or 0.25 percent of the rows, while an equal-height histogram represents approximately $\frac{1}{200}$, or 0.50 percent of the rows.

Compressed histogram

Compressed histograms contain a mix of up to 200 equal-height and high-biased intervals plus interval 0, with a maximum of 199 high-biased intervals plus one equal-height interval and interval 0. Compressed histograms are an improvement over pure equal-height histograms because they provide exact statistics for the largest values in the data. This is useful for join estimation, for example, which compares the largest values in relations to be joined.

39. The number of high-biased intervals in a compressed histogram is limited to a maximum of 199.
Beginning at interval 1 and ranging to \textit{number_of_intervals} - 1, high-biased intervals always precede equal-height intervals in a compressed histogram.

The following graphic illustrates the concept of a compressed histogram. Note that the number of rows per equal-height interval is only \textit{approximately} equal in practice, so the graphic is slightly misleading with respect to the precise equality of heights for all equal-height intervals in the compressed histogram.

In this example compressed interval histogram, the first three intervals, which define five loner values, have the values -2, -2, and -1 respectively.\footnote{High-biased intervals with 2 values are stored first, followed by the high-biased interval with only 1 value, if any, followed by the equal-height intervals for the histogram.}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{compressed_histogram.png}
\caption{Compressed Interval Histogram}
\end{figure}

\section*{Content and Storage of Histograms}

Index and column histograms are stored in the \textit{FieldStatistics} column of the \textit{DBC.TVFields} table in the data dictionary. The \textit{MaxStatsInterval} flag in your Cost Profile or in the DBS Control record\footnote{The \textit{MaxStatsInterval} flag can be set in either place. If there is a conflict between the settings, the system uses the value in your Cost Profile and ignores the setting of the flag in DBS Control.} determines the maximum number of interval histograms for your system. This is an internal flag, so if you need to change it from its Type 2 default value of 200,\footnote{The Type 1 Cost Profile default for \textit{MaxStatsInterval} is 100.} you must consult your Teradata support representative to make the change for you. The maximum size of the statistics structure is 14,114 bytes.
The following table lists details of the data stored in this column:

<table>
<thead>
<tr>
<th>Type of Statistic/Interval</th>
<th>Field Length (Bytes)</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global to Histogram</td>
<td>2</td>
<td>Length of this field.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Time the statistics were collected in the following format:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Time Element</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Year</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Month</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Day</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hour</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Centisecond</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Statistics version.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Sampled.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Sampling percentage.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Number of partly null and all null rows in the column or index set.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Number of all null rows in the column or index set.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>All-AMPs average of the average number of rows per NUSI value per individual AMP.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Number of intervals in the frequency distribution for the column or index set.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Number of AMPs on the system.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Flag indicating whether the stored statistics are numeric.</td>
</tr>
</tbody>
</table>
Interval 0 contains global statistics for the column or index set. The following information is stored in interval 0. Note that the number of bytes used to store the values for fields 1 and 2 depends on whether the column data is numeric or nonnumeric.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>Maximum numeric value for the column set.</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>Maximum nonnumeric value for the column set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For interval 0 only, this is actually the minimum value for the entire column or index set represented by the histogram.</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>Modal numeric value for the column set.</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>Modal nonnumeric value for the column set.</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>Frequency of occurrence of the modal value for the column or index set.</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>Number of distinct values for the column or index set.</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>Cardinality of the column or index set.</td>
</tr>
</tbody>
</table>

**Specific Interval Types**

Intervals 1 - 200 contain information specific to their respective column or index set histogram. The data stored in the columns is different for equal-height and high-biased intervals. Although they are mixed within a compressed histogram, equal-height and high-biased intervals are not interspersed.

- The high-biased intervals in a compressed histogram, if any, begin with interval 1 and continue up to interval \( n \), where \( n \) represents the highest numbered high-biased interval. The maximum number of high-biased intervals per 200 interval histogram is 199.
- The equal-height intervals in a compressed histogram begin at interval \( n + 1 \). If the histogram contains no high-biased intervals, then the equal-height intervals begin at interval 1. All compressed histograms have at least one equal-height interval.
High-Biased Intervals
High-biased intervals contain cardinality statistics for skewed data values.

<table>
<thead>
<tr>
<th>Type of Statistic/Interval</th>
<th>Field Length (Bytes)</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-biased interval</td>
<td>40 per numeric data interval. 56 per nonnumeric data interval.</td>
<td>For a high-biased interval, the fields are defined as follows:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
<th>Statistic</th>
</tr>
</thead>
</table>
| 1     | 8     | Number of loners in the interval. 

<table>
<thead>
<tr>
<th>WHEN the interval has this many loners ...</th>
<th>THEN the field is set to this value ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>Smaller (or only) loner numeric value for the interval.</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>Smaller (or only) loner nonnumeric value for the interval.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>Cardinality of the smaller (or only) loner value in the interval.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>Larger loner numeric value for the interval if it contains two loners.</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>Larger loner nonnumeric value for the interval if it contains two loners.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WHEN the interval has this many loners ...</th>
<th>THEN the field is defined as ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>the same as Field 2.</td>
</tr>
<tr>
<td>2</td>
<td>the value of the larger loner in the interval.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>Cardinality of the larger loner value in the interval if it contains two loners.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WHEN the interval has this many loners ...</th>
<th>THEN the field is defined as ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>the same as Field 3.</td>
</tr>
<tr>
<td>2</td>
<td>the cardinality of the larger loner value in the interval.</td>
</tr>
</tbody>
</table>
Rows for each histogram are separated into three categories:

- Nulls
- Loners
- Nonloners

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

- 199 high-biased (loner) intervals and one equal-height interval
- All equal-height (nonloner) intervals, referred to as an equal-height interval histogram.
- A mix of high-biased and equal-height intervals, referred to as a compressed histogram.

Only the first 16 bytes of nonnumeric values are stored,43 so loners whose first 16 bytes match one another are stored contiguously. To address this storage with respect to estimating cardinalities, the statistic for calculating cardinality must be adjusted.

---

43. And numeric values are stored as 8-byte FLOAT values. See “Possible Issues With Respect To Statistics On Large DECIMAL and CHARACTER Values” on page 174.
The adjusted statistic for estimating the cardinality for equivalent loner values is given by the following equation:

\[
C = \frac{\sum f^2}{\sum f}
\]

where:

<table>
<thead>
<tr>
<th>This variable</th>
<th>Represents the</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C )</td>
<td>estimated cardinality of the relation.</td>
</tr>
<tr>
<td>( f )</td>
<td>frequency of the loner value in question.</td>
</tr>
</tbody>
</table>

Consider the following two very simple examples:

<table>
<thead>
<tr>
<th>For this value of loner ( f ) ...</th>
<th>The cardinality estimate is ...</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( \cdot 1 ) and ( \cdot 3 )</td>
<td>2.5</td>
<td></td>
</tr>
</tbody>
</table>

\[
C = \frac{\sum f^2}{\sum f} = \frac{1^2}{1} = 1
\]

\[
C = \frac{\sum f^2}{\sum f} = \frac{1^2 + 3^2}{1 + 3} = \frac{10}{4} = 2.5
\]

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 \( f = 1 \).
**Possible Issues With Respect To Statistics On Large DECIMAL and CHARACTER Values**

The system stores statistics on numbers using the FLOAT data type. Converting a 16-byte DECIMAL value to an 8-byte FLOAT value rounds the number down to 53 bits of precision, and a 16-byte DECIMAL number potentially requires 127 bits of precision to be represented exactly.

This may or may not impact the usefulness of the statistics collected. For example, if a DECIMAL(30,0) column has values containing all 30 digits where the leftmost 15 digits are all identical, but the rightmost 15 digits are not, then the statistics will show all of the values to be equal. This is very similar to the case for CHARACTER data, where only the first sixteen bytes are stored.

See *SQL Data Types and Literals* and *Database Design* for more information about large DECIMAL numbers.

See the documentation for the MaxDecimal flag of the DBS Control record in *Utilities* and *SQL Data Types and Literals* for more information about how to control the precision and storage size of DECIMAL values.
Sampled Statistics

Introduction

There are occasions when it becomes difficult to expend the effort required to collect full-table statistics. For example, suppose you have a multibillion row table that requires several hours collection time, and a collection window that is too narrow to permit the operation. If you encounter situations like this when it is not possible to collect full-table statistics, you can opt to collect statistics on a sample of table rows instead.44

Note that sampled statistics are different from random AMP samples in that you specify the percentage of rows you want to sample explicitly in a COLLECT STATISTICS (Optimizer Form) request to collect sampled statistics, while the number of AMPs from which random AMP samples are collected and the time when those samples are collected is determined by Teradata Database, not by user choice. Furthermore, sampled statistics produce a full set of collected statistics, while random AMP samples collect only a subset of the statistics that are stored in interval histograms (see “Random AMP Sampling” on page 178 for details).

Statistical sampling is known to be a reliable method of gathering approximate statistical estimates when the appropriate preconditions are met (Acharya et al., 1999; Babcock et al., 2003; Chaudhuri et al., 1999, 2001, 2004; Ganguly et al., 1996; Gibbons et al., 2002; Haas and König, 2004; Haas et al., 1994; Jermaine, 2003; Jermaine et al., 2004; Lipton et al., 1990).

Caution: The quality of the statistics collected with full-table sampling is not guaranteed to be as good as the quality of statistics collected on an entire table without sampling. Do not think of sampled statistics as an alternative to collecting full-table statistics, but as an alternative to never, or rarely, collecting statistics.

When you use sampled statistics rather than full-table statistics, you are trading time in exchange for what are likely to be less accurate statistics. The underlying premise for using sampled statistics is usually that sampled statistics are better than no statistics.

44. You cannot collect sampled statistics using the COLLECT STATISTICS USING SAMPLE … COLUMN column_name syntax if a column specified by column_name is a component of the partitioning expression of a PPI table.

You can collect statistics on such a column using the COLLECT STATISTICS USING SAMPLE … INDEX column_name | index_name syntax if the specified column is both a member of the partitioning expression column set and a member of an index column set.
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 taken across all AMPs 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:

<table>
<thead>
<tr>
<th>Statistical Sampling</th>
<th>Random AMP Sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collects statistics on a small sample of rows from all AMPs.(^a)</td>
<td>Collects statistics on a small sample of rows from a single AMP.(^b)</td>
</tr>
<tr>
<td>Collects full statistics (see “Content and Storage of Histograms” on page 168) and stores them in histograms in (DBC.TVFields).</td>
<td>Collects estimates for cardinality, number of distinct index values, and a few other statistics only and stores them in the data block descriptor.</td>
</tr>
<tr>
<td>Expands sample size dynamically to compensate for skew.</td>
<td>Sensitive to skew.</td>
</tr>
<tr>
<td>Provides fairly accurate estimates of all statistical parameters.</td>
<td>Provides fairly accurate estimates of base table and NUSI cardinality if the following conditions are met:</td>
</tr>
<tr>
<td></td>
<td>• The table is large</td>
</tr>
<tr>
<td></td>
<td>• The distribution of values is not skewed</td>
</tr>
<tr>
<td></td>
<td>• The data is not taken from an atypical sample</td>
</tr>
<tr>
<td></td>
<td>Other standard statistical parameters are less likely to be as accurate.</td>
</tr>
</tbody>
</table>

\(^a\) If the columns are not indexed, then the rows are organized randomly on each AMP, so the system just scans the first \(n\) percent of rows it finds, where the value of \(n\) is determined by the relative presence or absence of skew in the data. Conceivably, the entire sample could be taken from the first data block on each AMP, depending on the system configuration and cardinality of the table being sampled.

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

\(^b\) This is the system default. You can change the number of AMPs from which a random AMP sample is taken by altering the value of an internal DBS Control flag.

The full range of possibilities is 1 AMP, 2 AMPs, 5 AMPs, all AMPs on a node, and all AMPs on a system. Consult your Teradata support representative for details.
You cannot sample single-column PARTITION statistics at a level lower than 100 percent. You can submit a COLLECT STATISTICS request at a lower percentage without the request aborting and without receiving an error message, but Teradata Database does not honor the specified percentage, and the system automatically changes the sampling percentage to 100 internally (see the documentation for “COLLECT STATISTICS (Optimizer Form)” in SQL Data Definition Language for details).

A Note on Recollecting Sampled Statistics

When you recollect statistics on a column or index, whether for the Optimizer or for a QCD analysis, Teradata Database uses the same method of collection, either full-table scan or sampling, to recollect the statistics as was used in the initial collection.

The only exception to this is the case where you had collected sampled statistics on a column that is part of the column set defined in the partitioning expression for a partitioned primary index in a release before that became a restricted action. In this case, the recollection is not done using sampling, but is performed on all rows in the table. Teradata Database does not return a message to the requestor when this occurs.

If you had collected statistics on the column as part of an index on a PPI table, then the system follows the general rule and recollects sampled statistics on the index column.

If you would like to change the collection method, you must submit a new COLLECT STATISTICS statement with the new collection method fully specified.45

See the documentation for “COLLECT STATISTICS (Optimizer Form)” in SQL Data Definition Language for additional information.

45. This option does not apply for sampled COLUMN statistics on a component of the partitioning expression for a PPI table, which are not valid, and which you cannot specify in a COLLECT STATISTICS statement. The system always collects (and recollects) statistics on all the rows in the table for this particular case.
Random AMP Sampling

Introduction

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

The cardinality estimates collected by a random AMP sample are stored in the NumRows column of the file system data block descriptor (DBD) for the table. The system collects a new random AMP sample whenever a DBD is fetched, but if interval histogram statistics exist for the columns or indexes of interest, they override the statistics collected by this random AMP sample by default.

Note that the statistics collected by a random AMP sample apply to indexed columns only. If you do not collect statistics on nonindexed columns, then the Optimizer uses various situation-specific heuristics to provide arbitrary estimates of cardinalities.

Do not think of random AMP samples as a substitute for collecting statistics. For example, while it is true that a random AMP sample generally produces reasonable estimates of the cardinality and number of unique values for a column or index set, it does not generally produce a good estimate of the selectivity of a given predicate in a query.

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 AMP in the system accurately characterizes the demographics for the entire table.
- The rows of the sampled table are distributed evenly, without skew.

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

46. Hashing on table ID makes the statistics more consistent from sample to sample; however, because the hashing is based on the table ID, the sample is not truly random. A more accurate term to describe random AMP sampling would be dynamic AMP sampling.

47. An important implication of this assumption is that random AMP samples can be poor estimators of the true population statistics of a table whenever the cardinality of the table is less than the number of AMPs on the system. Because of this, you should always collect statistics on small tables.

48. Sampled statistics do not work well with the partitioning columns of PPI tables, however, and should not be used to collect statistics for them.
Aside

It is possible to randomly sample table rows from more than a single AMP. The number of randomly selected AMPs from which the system samples statistics is controlled by the RandomAMPSampling internal flag in the DBS Control record. The default is one AMP, but it is possible to change this default specification to sample statistics from 2 AMPs, 5 AMPs, all AMPs in a node, or all AMPs on your system.

A related internal flag in DBS Control, RowsSampling, controls the percentage of rows to read in order to make an estimate of the RowsPerValue statistic. Because this is an internal DBS Control flag, you cannot change its value. Consult your Teradata support representative for details if you think your site would find it useful to use multiple AMP random sampling instead of single AMP sampling.

Multiple AMP random sampling improves the row count, row size, and rows per value statistics estimates for a given table. Multiple AMP sampling produces better join plans, better query execution times, and shorter elapsed times than single AMP random samples for several reasons.

The maximum benefit is realized for tables that have heavily skewed row distributions. When the distribution of rows is severely skewed, a single AMP sample can produce incorrect estimates of row count and row size information, which in turn can cause the Optimizer to produce a bad plan. Poor estimates can occur in such a case because the single sampled AMP in a skewed distribution might have significantly fewer rows than are found in the total population, or even no rows. Also see “How Cardinality Is Estimated From a Random Multiple-AMPS Sample” on page 182.

The following table compares the cardinality estimates for several tables collected from one possible set of random AMP samples, taken from one or all AMPs on a 20 AMP system, with the true table cardinality. With the exception of the 2 small tables, nation and region, each AMP contained roughly the same number of rows per table. The data is only intended to suggest a general range of results, not to imply that identical results would be seen for any system of any size. The Percent Difference columns express the magnitude of the difference between a single-AMP cardinality estimate and the true cardinality of the table and the difference between an all-AMP cardinality estimate and the true cardinality of the table, respectively. They are not a measure of the percent difference between the two sampling methods, which is expressed by the last column in the table.
### Chapter 2: Query Rewrite and Optimization

#### Random AMP Sampling

See [http://www.tpc.org/tpch/spec/tpch2.6.0.pdf](http://www.tpc.org/tpch/spec/tpch2.6.0.pdf) for the definitions of these tables. The following table provides the primary and secondary indexes on the tables studied because the TPC benchmark tables are not defined with indexes.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Actual Table Cardinality</th>
<th>Cardinality Estimate from Sampling 1 AMP</th>
<th>Percent Difference</th>
<th>Cardinality Estimate from Sampling All AMPs</th>
<th>Percent Difference</th>
<th>Percent Difference Between the Two Estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td>lineitem</td>
<td>300,005,811</td>
<td>299,968,260</td>
<td>0.01</td>
<td>299,966,000</td>
<td>0.01</td>
<td>7.53</td>
</tr>
<tr>
<td>ordertbl</td>
<td>75,000,000</td>
<td>74,958,940</td>
<td>0.06</td>
<td>74,978,450</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>customer</td>
<td>7,500,000</td>
<td>7,497,900</td>
<td>0.03</td>
<td>7,501,810</td>
<td>0.02</td>
<td>0.05</td>
</tr>
<tr>
<td>partsupp</td>
<td>40,000,000</td>
<td>39,915,260</td>
<td>0.21</td>
<td>39,979,133</td>
<td>0.05</td>
<td>0.16</td>
</tr>
<tr>
<td>parttbl</td>
<td>10,000,000</td>
<td>10,003,640</td>
<td>0.04</td>
<td>9,998,712</td>
<td>0.01</td>
<td>0.05</td>
</tr>
<tr>
<td>supplier</td>
<td>500,000</td>
<td>497,760</td>
<td>0.45</td>
<td>499,269</td>
<td>0.15</td>
<td>0.30</td>
</tr>
<tr>
<td>nation</td>
<td>25</td>
<td>20</td>
<td>22.22</td>
<td>13</td>
<td>63.64</td>
<td>42.42</td>
</tr>
<tr>
<td>region</td>
<td>5</td>
<td>20</td>
<td>120.00</td>
<td>2</td>
<td>163.64</td>
<td>163.64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Primary Index Column</th>
<th>Unique/Nonunique</th>
<th>Secondary Index Column</th>
<th>Unique/Nonunique</th>
</tr>
</thead>
<tbody>
<tr>
<td>lineitem</td>
<td>l_orderkey</td>
<td>NUPI</td>
<td>l_partkey</td>
<td>NUSI</td>
</tr>
<tr>
<td>ordertbl</td>
<td>o_orderkey</td>
<td>UPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>customer</td>
<td>c_custkey</td>
<td>UPI</td>
<td>c_nationkey</td>
<td>NUSI</td>
</tr>
<tr>
<td>partsupp</td>
<td>ps_partkey</td>
<td>UPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parttbl</td>
<td>p_partkey</td>
<td>UPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>supplier</td>
<td>s_suppkey</td>
<td>UPI</td>
<td>s_nationkey</td>
<td>NUSI</td>
</tr>
<tr>
<td>nation</td>
<td>n_nationkey</td>
<td>NUPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>region</td>
<td>r_regionkey</td>
<td>UPI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The results suggest that random AMP sampling, even when only one AMP is sampled, collects acceptably accurate cardinality estimates when both of the following assertions are true:

- The columns are primary or secondary indexes for tables that are not skewed.
- The cardinalities of those tables are substantially larger than the number of AMPs in the system.

See http://www.tpc.org/tpch/spec/tpch2.6.0.pdf for the definitions of these tables.

**End of Aside**

**How Cardinality Is Estimated From a Random Single-AMP Sample**

The process used to gather statistics on an indexed column using a random AMP sample is as follows:

1. Select an AMP to be used to collect a cardinality estimate by hashing on the value of the table ID to generate an AMP number.
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{Estimated cardinality of table} = \frac{\text{Average number of rows in sampled cylinder}}{\text{Datablock}} \times \frac{\sum \text{Number of data blocks in sampled cylinder}}{\sum \text{Number of cylinders with data for the table on this AMP}} \times \frac{\sum \text{Number of AMPs in the system}}{\sum \text{Number of distinct values, the average cardinality for each index, and the average size of each index on each AMP are the only statistics estimated by a random AMP sample.}^{49}

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 193 for a list of those statistics).

49. 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.

49 In the case of a PPI table where the partitioning columns are not part of the primary index, rows per value sampling is not done.

50. The term skewed here means having outlier values that skew the value distribution in the statistical sense. It does not refer to an unbalanced distribution of table rows among the AMPs of a system.
Because the Optimizer understands that the information it collects using a single random AMP sample is less reliable, it assumes Low confidence\(^{51}\) in the outcome and is less aggressive in its pursuit of optimal query plans, particularly join strategies. This means 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 by collecting full statistics as necessary.

However, if random multiple AMP sampling is used, and the number of AMPs sampled is 5 or more, the Optimizer upgrades the confidence level to High if it determines that the sampled data is not skewed. The system performs a skew analysis based on the distribution of rows from each AMP in the sample, and computes the expected number of rows per AMP. If the number of rows on a AMP is less than 5 percent of the expected number of rows per AMP, the AMP is considered to be skewed with respect to the table in question.

If the total number of AMPs in the skewed list is less than or equal to 5 percent of the total number of AMPs sampled, then the confidence level is set to Low; otherwise, it is set to High.

**How Cardinality Is Estimated From a Random Multiple-AMPs Sample**

The only procedural difference between random single-AMP and random multiple-AMP samples involves the selection of which AMPs to sample in addition to the initial AMP identified by hashing the table ID.

Once the first AMP has been identified by hashing on the table ID value, the next \(n\) AMPs are selected in increasing order of their AMP ID.\(^{52}\) If the first AMP selected is near the end of the AMP ID series, the system wraps the selection to the beginning of the series.

For example, consider a 10 AMP system where the default number of AMPs to sample is 5. The first AMP selected by the table ID is 6. The AMPs selected to complete the sample of five would be those with AMP IDs of 7, 8, 9, and 0.

Similarly, given the same system but a different table ID, suppose the AMP ID selected was 9. The AMPs selected to complete the sample of five would be those with AMP IDs of 0, 1, 2, and 3.

Note that the system uses a Last Done Channel technique to reduce messaging overhead for random AMP samples taken from all AMPs. This method is used when all-AMP sampling is enabled by setting the internal DBS Control flag RowsSampling or your cost profile to 0, which is the default. The following graphic illustrates the Last Done Channel all-AMP sampling method:

---

\(^{51}\) As reported by an EXPLAIN of a query where random AMP sampling would be used. Actually, the EXPLAIN would report No confidence because there are no statistics on the condition. When the request is actually performed, the system would perform a random AMP sample, and the resulting cardinality estimate would then be expressed with Low confidence. See “Optimizer Confidence Levels” on page 487 for more details about Optimizer confidence levels.

\(^{52}\) Sequential AMP IDs are selected to obtain a more efficient load distribution.
It is important to realize that even multiple-AMP sampling is not a replacement for collecting complete statistics.

It is equally important to understand that sampling from an AMP subset does not guarantee that the subset is representative of the data across all AMPs. It is even possible, though not likely, that a subset sample can produce statistics that are less representative of the population than statistics produced by sampling a single AMP.

**Random AMP Sampling of USIs**

Random AMP sampling assumes that the number of distinct values in a USI equals its cardinality, so it does not read the index subtable for USI equality conditions. The number of distinct values in the USI is assumed to identical to the table cardinality taken from the random AMP sample on the primary index. Because equality conditions on a unique index return only one row by definition, the Optimizer always chooses the direct USI path without costing it or using statistics. However, if a USI will frequently be specified in nonequality predicates, such as range constraints, then you should collect statistics on it.

**Random AMP Sampling of NUSIs**

Random AMP sampling for NUSIs is very efficient. The system reads the cylinder index that supports the index subtable rows on the sampled AMP and determines the number of rows on that cylinder. Except for situations where the NUSI is very nonunique, there is one subtable row for each distinct value in the NUSI. Following through on that knowledge, the sampling process assumes that each subtable row it finds translates to one index value.

The sampling process assumes the following assertions about the data are true:

- The same values occur on all AMPs
- The number of distinct values found on the sampled AMP represents the total number of distinct values for the NUSI
Because of these assumptions, random AMP samples are less accurate for fairly singular NUSIs than they are for fairly nonunique NUSIs. If more than one AMP is sampled, the system calculates an average number of distinct values across all sampled AMPs.

NUSI estimates are always made after the total cardinality of the table has been estimated. The sampling process divides the estimated total cardinality by the NUSI cardinality to estimate the approximate number of rows per value in the index. If a query passes a single value in an equality condition to match to the NUSI, and skew, if present, is not extensive, then query plans are generally quite good.

For the definitions of these tables see http://www.tpc.org/tpch/spec/tpch2.6.0.pdf.

The following table summarizes some cases illustrating random-AMP sampled NUSI estimates where there is no significant skew in the data. Note that in the cases where there are very few rows per value in the index (c_phone and c_acctbal), the system makes less accurate cardinality estimates.
<table>
<thead>
<tr>
<th>Table Name</th>
<th>NUSI Column</th>
<th>Query Text</th>
<th>Actual Result Cardinality</th>
<th>Estimated Result Cardinality from Random AMP Sample</th>
<th>Percent Difference</th>
<th>Estimated Cardinality from Collecting Full Statistics</th>
<th>Percent Difference</th>
<th>Percent Difference Between the Two Estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td>parttbl</td>
<td>p_type</td>
<td>SELECT * FROM parttbl WHERE p_type = 'economy brushed copper';</td>
<td>67,103</td>
<td>67,123</td>
<td>0.03</td>
<td>66,667</td>
<td>0.65</td>
<td>0.68</td>
</tr>
<tr>
<td>partsupp</td>
<td>ps_suppkey</td>
<td>SELECT * FROM partsupp WHERE ps_suppkey = 555;</td>
<td>80</td>
<td>82</td>
<td>2.47</td>
<td>81</td>
<td>1.24</td>
<td>0.01</td>
</tr>
<tr>
<td>customer</td>
<td>c_acctbal</td>
<td>SELECT * FROM customer WHERE c_acctbal = 7944.22</td>
<td>10</td>
<td>24</td>
<td>82.35</td>
<td>7</td>
<td>35.29</td>
<td>109.68</td>
</tr>
<tr>
<td>customer</td>
<td>c_phone</td>
<td>SELECT * FROM customer WHERE c_phone = '25-548-367-9974';</td>
<td>1</td>
<td>21</td>
<td>181.82</td>
<td>2</td>
<td>165.22</td>
<td>158.33</td>
</tr>
</tbody>
</table>
Sampling Efficiency for Nonindexed Predicate Columns

While random AMP sampling provides reasonable cardinality estimates for UPIs, USIs, and distinct NUSI values, it fails to provide useful cardinality estimates for unindexed predicate columns in a query. When the WHERE clause of a request specifies a predicate of \texttt{zip\_code} = 90230 and there are no statistics on the column \texttt{zip\_code}, the system uses various heuristics to apply default selectivities to estimate default cardinalities for those columns. For this example, the heuristic is to estimate the cardinality to be 10 percent of the table rows.

You should collect statistics on frequently selected columns, particularly if their values are skewed, rather than relying on the default cardinality estimates.

The table on the next page shows three queries with simple selection criteria. The actual response cardinalities from the queries are presented, along with the cardinality estimates derived from the heuristic defaults, the cardinality estimates derived from the collection of full statistics, and the percentage difference between the two estimates. There is considerable divergence between the true cardinalities and the heuristic estimates in all cases, while the estimates made by collecting full statistics are very close to reality in all cases.

If you compare the actual cardinalities to the default estimates, you can see that with a single selection column in an equality condition, the Optimizer assumes that about 10 percent of the rows in the table will be returned. With two selection criteria, as seen in the third request in the table, the Optimizer assumes that about 7.5 percent of the rows will be returned. In every case, the default cardinality estimates significantly overestimate the number of rows returned, potentially leading to poor join planning if the example selection criteria were part of a more complex query where additional tables were involved in join operations with them.

If NUSIs had been defined on those columns, the Optimizer would use a random AMP sample estimate of their cardinalities even if the NUSI column had not been used to develop the query plan.

For selection criteria on unindexed columns, the identical poor cardinality estimates are made whether random AMP sampling is defined for one, few, or all AMPs in the system. The cardinality estimates for unindexed columns do not improve when all AMPs are sampled because the system always samples 10 percent of the table rows irrespective of the number of AMPs on which the sampling is done.

The significant conclusion to be drawn from these results is that it is important to collect statistics on nonindexed columns that are frequently specified in selection predicates. Random AMP samples never provide adequate cardinality estimates for unindexed columns, even if those columns are not skewed.
<table>
<thead>
<tr>
<th>Query Text</th>
<th>Actual Table Cardinality</th>
<th>Actual Response Set Cardinality</th>
<th>Heuristic Estimate of Response Set Cardinality</th>
<th>Percentage Difference</th>
<th>Full Statistics Estimate of Response Set Cardinality</th>
<th>Percentage Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM lineitem WHERE l_commitdate = '1998-01-06';</td>
<td>300,005,811</td>
<td>123,959</td>
<td>30,017,594</td>
<td>198.36</td>
<td>124,591</td>
<td>0.25</td>
</tr>
<tr>
<td>SELECT * FROM partsupp WHERE ps_availqty = 7874;</td>
<td>40,000,000</td>
<td>3,936</td>
<td>3,991,526</td>
<td>199.61</td>
<td>4,000</td>
<td>1.62</td>
</tr>
<tr>
<td>SELECT * FROM parttbl WHERE p_size = 22 AND p_brand = 'Brand#23';</td>
<td>10,000,000</td>
<td>8,061</td>
<td>750,398</td>
<td>195.75</td>
<td>7,997</td>
<td>0.80</td>
</tr>
</tbody>
</table>
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. 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 Accuracies 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.
   Random AMP samples are also recollected each time the DBD is retrieved from disk, so they are nearly always more current than residual statistics.

2. Random all-AMPs samples are better than random AMP samples in most cases.

3. Full-table population statistics are usually better than any form of 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.

53. 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.
### Comparing the Relative Accuracies of Various Methods of Collecting Statistics

<table>
<thead>
<tr>
<th>Collection Method</th>
<th>Relative Elapsed Time to Collect</th>
<th>Accuracy Rank (Higher Number = Higher Accuracy)</th>
<th>Comments</th>
</tr>
</thead>
</table>
| None. 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.  
At best, can produce as good a query plan as freshly collected statistics.  
• Because statistics exist, Optimizer does not collect fresh statistics using a random AMP sample.  
• Optimizer does use derived statistics, but their starting point is the existing statistics. The Derived Statistics subsystem has ways to compensate for stale statistics, but it is still better to begin with the freshest set of statistics that can be made available (see “Derived Statistics and Stale Statistics” on page 251) |
| Random AMP sample          | Almost none.                     | 2                                             | • Data is collected from a subset of the rows on a single AMP\(^{b}\) and might not be representative of full table demographics.  
• Collects statistics for table cardinality, average cardinality per value, average cardinality per index, average size of each index on each AMP, 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. |
| 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. |
| 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. |

---

\(^{a}\) When the data is skewed, this percentage is larger, depending on how much the system dynamically increases its sample size.  
\(^{b}\) This is the default. Depending on an internal DBS Control variable, the default number of AMPs sampled ranges over 1, 2, 5, all AMPs on a node, or all AMPs on a system. Consult your Teradata support representative for details.
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 Sampled Statistics for Static Columns” on page 254 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.

Note that with the exception of statistics obtained by a random AMP sample, statistics are collected globally, so are not affected by reconfiguring your system. In other words, all things being equal, there is no need to recollect statistics after you reconfigure your system.

Teradata Statistics Wizard

The Teradata Statistics Wizard 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, and provides a means for verifying the enhanced efficiency of queries run against the specified workloads.

See *Teradata Statistics Wizard User Guide* and Chapter 7: “Database Foundations for the Teradata Index and Statistics Wizards” 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.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Required or Recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recollect all statistics when you upgrade to a new Teradata Database release.</td>
<td>Strongly recommended.</td>
</tr>
<tr>
<td>By the Ten Percent Rule, you should recollect statistics whenever table or partition demographics change by 10 percent or more.</td>
<td>Strongly recommended.</td>
</tr>
<tr>
<td><strong>FOR this type of table ...</strong></td>
<td><strong>Recollect statistics whenever the demographics of this entity changes by 10 percent or more ...</strong></td>
</tr>
<tr>
<td>NPPI</td>
<td>table</td>
</tr>
<tr>
<td>PPI</td>
<td>partition</td>
</tr>
<tr>
<td>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.</td>
<td>-</td>
</tr>
<tr>
<td>Collect statistics on newly created, empty tables to create the synoptic data structures for subsequent collection of statistics.</td>
<td>Recommended.</td>
</tr>
<tr>
<td>Recollect statistics whenever the number of rows per distinct value is less than 100.</td>
<td>Recommended.</td>
</tr>
</tbody>
</table>

The following table provides more specific recommendations for collecting statistics. You should consider this set of recommendations to be both of the following:

- Minimal
- Essential

<table>
<thead>
<tr>
<th><strong>FOR this category of table ...</strong></th>
<th><strong>You should collect statistics on ...</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>all columns used in join conditions.</td>
</tr>
<tr>
<td>large</td>
<td>all NUPIs.</td>
</tr>
<tr>
<td>small\textsuperscript{a}</td>
<td>the primary index.</td>
</tr>
</tbody>
</table>

\textsuperscript{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. For a 20 AMP system, table cardinality would have to be less than 100 rows for the table to be considered small, for a 100 AMP system, less than 500 rows, and so on.
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.
- Nonindexed columns frequently 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 (see SQL Data Definition Language for syntax and usage information) 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 Database SQL 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.

You can view the statistics for a column or index using the Statistics Collection program of the Teradata Manager product or by running the HELP STATISTICS (Optimizer Form) statement. See SQL Data Definition Language for more information.

The description of some statistics depends on whether they describe an equal-height interval or a high-biased interval (see “Types of Interval Histograms Used By Teradata Database” on page 166).

Different statistics are stored for a column depending on whether its values are highly skewed or not, as indicated by the following table:

<table>
<thead>
<tr>
<th>IF the distribution of column values is ...</th>
<th>THEN its statistics are stored in this type of interval histogram ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonskewed</td>
<td>Equal-height</td>
</tr>
<tr>
<td>Highly skewed</td>
<td>High-biased</td>
</tr>
</tbody>
</table>

Note the use of the term estimate in the attribute descriptions documented by the following table. 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.
### How the Optimizer Uses Statistical Profiles

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date collected</td>
<td>Reported by HELP STATISTICS as Date. The date on which statistics were last collected.</td>
</tr>
<tr>
<td>Time collected</td>
<td>Reported by HELP STATISTICS as Time. The time at which statistics were last collected.</td>
</tr>
<tr>
<td>Number of rows</td>
<td>Reported by HELP STATISTICS as Number of Rows. An estimate of the cardinality of the table.</td>
</tr>
<tr>
<td>Number of nulls</td>
<td>Reported by HELP STATISTICS as Number of Nulls. An estimate of the number of rows with partial or completely null columns for the column or index column statistics set.</td>
</tr>
<tr>
<td>Number of all nulls</td>
<td>Reported by HELP STATISTICS as Number of All Nulls. An estimate of the number of rows with all columns null for the column or index column statistics set.</td>
</tr>
<tr>
<td>Number of intervals</td>
<td>Reported by HELP STATISTICS as Number of Intervals. The number of intervals in the frequency distribution histogram containing the column or index statistics.</td>
</tr>
<tr>
<td>Numeric flag</td>
<td>Reported by HELP STATISTICS as Numeric. Identifies whether the data type of the column set reported on is numeric or nonnumeric.</td>
</tr>
<tr>
<td>Sampled flag</td>
<td>Reported by HELP STATISTICS as Sampled. Identifies whether the statistics were collected from all rows in the table or from a sampled subset.</td>
</tr>
</tbody>
</table>

#### Code Definitions

<table>
<thead>
<tr>
<th>Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The data type is nonnumeric.</td>
</tr>
<tr>
<td>nonzero</td>
<td>The data type is numeric.</td>
</tr>
<tr>
<td>0</td>
<td>Statistics collected on all the rows.</td>
</tr>
<tr>
<td>nonzero</td>
<td>Statistics collected on a sampled subset of the rows.</td>
</tr>
</tbody>
</table>
### Attribute Information

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version number</td>
<td>Reported by HELP STATISTICS as Version. The version number of the statistics structure in effect when the statistics were collected.</td>
</tr>
<tr>
<td>Number of distinct values</td>
<td>Reported by HELP STATISTICS as Number of Uniques. An estimate of the number of unique values for the column.</td>
</tr>
<tr>
<td>Minimum value for the interval</td>
<td>Reported by HELP STATISTICS as Min Value. An estimate of the smallest value for the specified column or index in the interval.</td>
</tr>
<tr>
<td>Maximum number of rows per value</td>
<td>Not reported by HELP STATISTICS. An estimate of the maximum number of rows having the particular value for the column.</td>
</tr>
<tr>
<td>Typical number of rows per value</td>
<td>Not reported by HELP STATISTICS. An estimate of the most common number of rows having the particular value for the column.</td>
</tr>
</tbody>
</table>

### Equal-Height Interval Statistics

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum value for the interval</td>
<td>Reported by HELP STATISTICS as Max Value. An estimate of the largest value for the column or index in the interval.</td>
</tr>
<tr>
<td>Modal value for the interval</td>
<td>Reported by HELP STATISTICS as Mode Value. An estimate of the most frequently occurring value or values for the column or index in the interval.</td>
</tr>
<tr>
<td>Number of rows having the modal value</td>
<td>Reported by HELP STATISTICS as Mode Frequency. An estimate of the distinct number of rows in the interval having its modal value for the column or index.</td>
</tr>
<tr>
<td>Number of nonmodal values</td>
<td>Reported by HELP STATISTICS as Non-Modal Values. An estimate of the number of distinct nonmodal values for the column or index in the interval.</td>
</tr>
<tr>
<td>Number of rows not having the modal value</td>
<td>Reported by HELP STATISTICS as Non-Modal Rows. An estimate of the skewness of the distribution of the index or column values within the interval.</td>
</tr>
</tbody>
</table>
Chapter 2: Query Rewrite and Optimization
How the Optimizer Uses Statistical Profiles

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-Biased Interval Statistics</td>
<td></td>
</tr>
<tr>
<td>Maximum value for the interval</td>
<td>Reported by HELP STATISTICS as Max Value. If two values are stored in the interval, then Max is the value for the second loner in a two-loner interval.</td>
</tr>
<tr>
<td>Modal value for the interval</td>
<td>Reported by HELP STATISTICS as Mode Value. 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.</td>
</tr>
<tr>
<td>Number of rows having the modal value</td>
<td>Reported by HELP STATISTICS as Mode Frequency. The number of rows in the interval having the smaller loner value.</td>
</tr>
<tr>
<td>Number of nonmodal values</td>
<td>Reported by HELP STATISTICS as Non-Modal Values. A code indicating the number of loner values for the interval.</td>
</tr>
<tr>
<td>Code</td>
<td>Definition</td>
</tr>
<tr>
<td>-1</td>
<td>The interval has one loner value.</td>
</tr>
<tr>
<td>-2</td>
<td>The interval has two loner values.</td>
</tr>
<tr>
<td>Number of rows not having the modal value</td>
<td>Reported by HELP STATISTICS as Non-Modal Rows. The number of rows in the interval having the larger loner value.</td>
</tr>
</tbody>
</table>
Examples of How Initial 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.

Note that what is described here is only the initial estimation of cardinalities, which is based on the statistics stored in the interval histograms. The Optimizer adjusts its cardinality estimates dynamically during the course of query optimization based on information derived from various database constraints, query predicates, and hash and join indexes. Because of the way the system calculates these statistics, they are referred to as derived statistics.

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. Recorded as the modal value for the interval.
- Cardinality of the most frequent value in the interval. Recorded as the modal frequency for the interval.
- Cardinality of distinct values in the interval as determined from a full-table scan.
- Cardinality 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:
  \[ \text{Values not equal to most frequent} = \text{Number of values in interval} - 1 \]
  where the factor 1 indicates the number of distinct values that occur most frequently in the interval.
- Cardinality of rows not containing the most frequent value in the interval. Recorded as the nonmodal frequency for the interval.
The data used for the examples is tabulated in the following table. Notice that the total number of distinct values (the shaded cells of the table) is the same for all five intervals. This is definitive, in theory, for equal-height interval histograms. In practice, the cardinalities of intervals of an equi-height interval are only approximately equal.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Interval Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Instances of the most frequent value in the interval</td>
<td>16</td>
</tr>
<tr>
<td>Number of values not equal to the most frequent value in the interval</td>
<td>10</td>
</tr>
<tr>
<td>Number of distinct values in the interval</td>
<td>11</td>
</tr>
<tr>
<td>Number of rows in the interval having the most frequent value</td>
<td>50</td>
</tr>
<tr>
<td>Number of rows in the interval not having the most frequent value</td>
<td>200</td>
</tr>
<tr>
<td>Number of rows in the interval</td>
<td>250</td>
</tr>
<tr>
<td>Maximum value in the interval</td>
<td>25</td>
</tr>
</tbody>
</table>

The following picture illustrates these numbers graphically. The hatched area represents the number of rows for other values in the interval, and footnotes are defined in the table following the illustration.
Examples of How Initial Cardinality Estimates Are Made for Simple Queries Using Interval Histograms

The example sets that follow do not deal with derived statistics (see “Derived Statistics” on page 206). They are meant to indicate only how the Optimizer would make its cardinality estimates if it only had the statistics in the interval histograms as a basis for making those estimates.

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 199, 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.

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 \( \text{value} \geq \text{lower_limit} \ \text{AND} \ \leq \text{upper_limit} \), 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.

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

[50 rows]
Chapter 2: Query Rewrite and Optimization
Examples of How Initial Cardinality Estimates Are Made for Simple Queries Using Interval Histograms

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.

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.

Estimated cardinality of the response set = \( \frac{100}{2} = 50 \) 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 201 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 202 because the response set spans two histogram intervals.

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 \textit{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 \textit{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.

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

\textbf{Example 6}

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

\[
\text{SELECT } * \text{ FROM table}
\text{WHERE condition BETWEEN 45 AND 65;}
\text{[ 355 rows ]}
\]

The Optimizer knows that the quantity of rows having the \textit{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:

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.
6. End of process.

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

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

Note that all of these examples are meant to show only how interval histograms are used in query optimization. They do not factor in the methods used by derived statistics to make cardinality estimates even more accurate.
Chapter 2: Query Rewrite and Optimization
Derived Statistics

Derived Statistics

Introduction

The initial cardinality estimates based on interval histogram statistics, from which the Optimizer begins the process of query optimization, are adjusted dynamically throughout query processing to gain increasing accuracy by applying information derived from various database constraints, query predicates, and hash and join indexes. Because of the way the system calculates these statistics, they are referred to as derived statistics.

Definition: Derived Statistics

Derived statistics are snapshots, derived from base table interval histogram statistics, that the Optimizer initializes and transforms while it optimizes a query. They are cardinality estimates that are transformed from various constraint sources, including query predicates, CHECK and referential integrity constraints, and hash and join indexes, and then adjusted dynamically at each stage of the query optimization process. In other words, derived statistics represent the demographics of column values after applying query predicates and demographic information derived from other sources.

For example, the Join Planner needs to know the demographics of join columns after applying single-table predicates or after making a join. All this information is derived and propagated throughout query optimization using the derived statistics infrastructure, and it is used consistently in all cost and estimation formulas.

Derived statistics are propagated across optimization stages in a flat data structure associated with each relation accessed by the query. An entry in this structure is made for each base table statistic that is required in the query and for the information derived from other sources.

Each entry contains both static and dynamically adjusted information such as:

- Number of unique values
- High modal frequency
- Original interval histogram statistics
- Uniqueness flag

At the first stage of query optimization, the information in the derived statistics is identical to the base table statistical information stored in the interval histograms. The system adjusts the initial state information and derives new information during the different stages of join planning such as the following:

- After applying single-table predicates
- After doing a binary join
- After doing aggregation

Because the original interval histogram statistics are also attached to every derived statistics entry, the Optimizer always has access to the original column or index demographics if it needs them.
All Optimizer estimation and costing logic obtains its demographic information from a single source: the derived statistics data structure. This enables all stages of query optimization to see a single version of the truth.

The derived statistics framework also employs several techniques for dealing with stale statistics (see “Derived Statistics and Stale Statistics” on page 251), including comparing cardinality estimates obtained from a random AMP sample with the statistics stored in the relevant interval histogram, and bidirectional inheritance of statistics between a base table and its supporting indexes, using whichever set has the more recent collection timestamp (see “Statistical Inheritance” on page 216).

Bidirectional inheritance is the term used to describe how base tables and their underlying indexes are able to inherit and use existing statistics from one another when either database object in an index-table pair has no existing interval histogram statistics.

If both database objects have existing interval histogram statistics, the Optimizer assumes that the more recently collected statistics are the more accurate, so it uses the set with the more recent collection timestamp.

**How Derived Statistics Work**

Rather than reverting to the original interval histogram statistics at the beginning of each stage of the optimization process, the Optimizer propagates all newly derived statistics to subsequent phases of optimization, refining its estimates as it goes. This applies to both single-table cardinality estimates and to join cardinality estimates. Such dynamic readjustment of cardinality estimates greatly reduces the multiplicative error propagation that otherwise produces increasingly, and often significantly less accurate, join cardinality estimates at each stage of join space analysis.

- Derived statistics derive and adjust their estimates based on several criteria:

<table>
<thead>
<tr>
<th>IF the statistics are for ...</th>
<th>THEN they are based on the predicates for this type of operation ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>single tables</td>
<td>single-table.</td>
</tr>
<tr>
<td>join operations</td>
<td>joins.</td>
</tr>
<tr>
<td>aggregation operations</td>
<td>grouping keys.</td>
</tr>
</tbody>
</table>

For example:

- Deriving the number of unique values using single-table predicates.

```sql
SELECT *
FROM t1, t2
WHERE d1 = d2
AND d1 IN (1, 2, 3, 4);
```

It is possible to derive the number of unique values, 4, just from the single-table predicate of this query.
• Deriving the number of unique values using single-table predicates and existing multicolumn statistics.

Note that as a general rule, the system adjusts multicolumn demographics based on single-table predicates rather than vice versa. For example, suppose multicolumn statistics have been collected on \((c1, d1)\). In the following request, the system adjusts the demographics of \((c1, d1)\) based on the derived statistics for the single-table equality predicate \(d1 = d2\).

```sql
SELECT * 
FROM t1,t2 
WHERE d1 = d2 
AND c1 > 10;
```

• Deriving the number of unique values using partition elimination and multicolumn PARTITION statistics. In the following revision of the previous case, the second component of the predicate has been changed to an equality condition. If multicolumn statistics have been collected on \((c1, d1)\), the system adjusts the derived demographics of the new entry based on the single-table predicate as well as deriving new demographics for \(d1\).

```sql
SELECT * 
FROM t1,t2 
WHERE d1 = d2 
AND c1 = 10;
```

• Deriving statistics on join columns using join predicates.

```sql
SELECT x1, y1, COUNT(*) 
FROM t1,t2 
WHERE x1=x2 
AND y1=y2;
```

Assume for this case that the number of values for \((x1,y1)\) = 100 and the number of values for \((x2,y2)\)=50.

The Optimizer derives the logically correct join cardinality of \((x1,y1)\) as 50 and propagates that value to the join result for use in the next stage of the join ordering process.

• Deriving the number of unique values from aggregate join index cardinality.

The definition of the aggregate join index is as follows:

```sql
CREATE JOIN INDEX aggji AS 
SELECT c1, d1, SUM(b1) 
FROM t1 
WHERE x1>10 
AND y1=10 
GROUP BY c1, d1;
```

The query is the following SELECT request:

```sql
SELECT * 
FROM t1,t2 
WHERE t1.c1=t2.c2 
AND x1>10 
AND y1=10;
```

The Optimizer takes the cardinality of \((c1,d1)\) to be equal to the cardinality of the join index \(aggji\).
• Deriving the demographics of a base table from a sparse single-table join index defined on it.

The definition of the single-table join index is as follows:

```sql
CREATE JOIN INDEX stji AS
    SELECT c1,d1
    FROM t1
    WHERE x1 > 10
    AND y1 = 10;
```

The query is the following SELECT request:

```sql
SELECT *
FROM t1,t2
WHERE c1 = c2
    AND x1 > 10
    AND y1 = 10;
```

The Optimizer adjusts the demographics for `t1.c1` based on the interval histogram statistics it reads for `stji.c1`.

• Deriving hierarchical relationships from single- and multicolumn statistics.

Hierarchical relationships between values are used to update and propagate demographic changes based on single-table predicates.

Define the number of values in `x` as $Nx$.

Define the number of values in `y` as $Ny$.

Define the number of values in `(x,y)` as $Nxy$.

<table>
<thead>
<tr>
<th>IF ...</th>
<th>THEN the value mapping between ...</th>
<th>AND the value mapping between ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Nx = Nxy$</td>
<td>$Nx \rightarrow Ny$ is derived as 1 $\rightarrow$ $1$</td>
<td>$Ny \rightarrow Nx$ is derived as 1 $\rightarrow Nxy/Ny$</td>
</tr>
<tr>
<td>$Ny = Nxy$</td>
<td>$Ny \rightarrow Nx$ is derived as 1 $\rightarrow$ $1$</td>
<td>$Nx \rightarrow Ny$ is derived as 1 $\rightarrow Nxy/Nx$</td>
</tr>
</tbody>
</table>

Consider the following example:

```sql
SELECT *
FROM t1, t2
WHERE d1 = d2
    AND c1 = 10;
```

If the value mapping between `c1` and `d1` is $1 \rightarrow 5$, then after applying the single-table condition, the derived statistics framework adjusts the number of unique values for `d1` to 5.

• Deriving the number of unique values using query predicates and column correlations.

Assume a 1:20 mapping between `c1` and `d1`. The Optimizer can use this information together with the predicates in the following query to estimate the number of unique values:

```sql
SELECT *
FROM d1,d2
WHERE d1 = d2
    AND c1 = 1;
```

From the given information, the Optimizer derives the number of unique `d1` values, which it determines to be 20.
The Optimizer captures and propagates session-level derived statistics across multiple requests on materialized global temporary and volatile tables if the following criteria are all true:

- The multiple requests are all INSERT … SELECT statements.
- No DELETE or UPDATE operations occur between successive INSERT … SELECT statements.
- A SELECT query follows the succession of INSERT … SELECT statements without intervening DELETE or UPDATE operations against the target global temporary or volatile tables.

Teradata Database automatically collects session-level derived statistics for materialized global temporary and volatile tables that are populated with data during the current session. The system uses session-level derived statistics in the same way it uses other derived statistics to make cardinality estimates for, and cost evaluations of, SQL requests. These session-level statistics are maintained in the global memory for a session and are dropped when the session logs off.

For example, suppose you have a global temporary table `gt1` and a volatile table `vt1`. The Optimizer retains and propagates the derived statistics from the successive INSERT … SELECT statements to the subsequent query made against `gt1` and `vt1`.

```sql
INSERT INTO gt1
SELECT *
FROM t1,t2
WHERE a1=a2;

INSERT INTO gt1
SELECT *
FROM t3,t4
WHERE a3=a4;

INSERT INTO vt1
SELECT *
FROM t5,t6
WHERE a5=a6;

SELECT *
FROM gt1,vt1, orders, customers, ...
WHERE ...;
```

In this example, the Optimizer uses the statistics previously derived during the three preceding INSERT … SELECT operations for `gt1` and `vt1` in the succeeding query made against those tables.

Note that these are all individual SQL requests, not component statements of a multistatement request.

Also note the following counter indication: if there are existing statistics on a materialized global temporary table, then those statistics override any derived statistics.
Global temporary and volatile tables populated in the following ways qualify for session-level derived statistics:

- CREATE TABLE AS … WITH DATA
- One or multiple INSERT … SELECT requests
- CREATE TABLE AS … WITH DATA followed by one or multiple INSERT … SELECT requests.

Session-level histograms are also inherited if the source table in an INSERT … SELECT or CREATE TABLE AS … WITH DATA request is a single table with no predicates. Note that the inherited statistics do not override collected statistics, and if a table is updated by DELETE or UPDATE requests after it has been populated, the system invalidates its session-level statistics.

The Optimizer propagates derived statistics across the multiple SQL statements of a multistatement request under the following scenario:

```
INSERT INTO t3
SELECT a1,a2,b1,b2
FROM t1,t2
WHERE a1=a2
;SELECT *
FROM t4,t3
WHERE a4=a3;
```

The system propagates the statistics derived for target table `t3` from the INSERT … SELECT statement to `t3` for the SELECT statement in the next part of this multistatement request.

- The Optimizer acts proactively against stale statistics, using the following method to detect and adjust them.

If the cardinality for a table or index differs by more than \( n \) percent from the cardinality determined by a random-AMP sample, then that statistic is deemed to be stale.

The value of \( n \) is set in your Cost Profile using the RASHistogramDeviationPct or RASHistAbsDeviation flags or both. The default value is 10 percent because the guideline for when statistics should be refreshed is to recollect when there is a change in the number of rows for a table or index subtable of 10 percent or more.

See “Stale Statistics” on page 251 for further information.

The system changes the existing cardinality count to the cardinality determined from a random-AMP sample if all the following criteria are true:

- The primary index for the table, hash index, or join index is not skewed.
- The cardinality of the database object is greater than the number of AMPs for the system.
- The sampled cardinality is greater than \( n \) percent of the estimated total cardinality for the database object.

The value of \( n \) is set in your Cost Profile using the RASHistogramDeviationPct flag or RASHistAbsDeviation flags or both (see “Cost Profile Flags” on page 277). The default value is 10 percent because the guideline on when statistics should be refreshed is to recollect when there is a change in the number of rows for a table or index subtable of 10 percent or more.
When the system adjusts its cardinality estimate, it also adjusts its estimate of the number of unique values for rolling columns accordingly.

When both join index and base table statistics are available, the Optimizer uses the more current estimates of the two based on their respective collection timestamps.

- Single-table statistics, both on single columns and multiple columns, are inherited bidirectionally:
  - If a base table has statistics, but a nonsparse single-table join index or hash index subtable on that table has none, then the index inherits the base table statistics as its own.
  - If a nonsparse single-table join index or hash index subtable has statistics, but the base table on which the index is defined has none, then the base table inherits the index statistics as its own.

- Single-level PARTITION statistics are inherited bidirectionally for single-column statistics only.
  Teradata Database does not support the bidirectional inheritance of multilevel PARTITION statistics.

- Interval histogram statistics are inherited from their supersets when available.
  For example, consider the following scenario:
  - An index is defined on column \textit{x1}, but no statistics have been collected on it; therefore, there is no histogram for \textit{x1}.
  - Multicolumn statistics have been collected on \((x1,y1)\).
  - The query specifies a range predicate on \textit{x1}.

A random AMP sample is not useful for making a range estimate; therefore, the Optimizer uses the histogram that is available for \((x1,y1)\) to make its cardinality estimate, which it then stores in the \textit{x1} entry of the derived statistics data structure to be propagated to the next stage of the optimization process.
Derived Statistics Flow

The following table and join index definitions are used for the query that illustrates the flow of derived statistics usage by the Optimizer to generate more accurate cardinality estimates:

CREATE TABLE t1 (  
an1 INTEGER,  
b1 INTEGER,  
c1 CHARACTER(5),  
d1 DATE);

CREATE TABLE t2 (  
an2 INTEGER PRIMARY KEY,  
b2 INTEGER,  
c2 CHARACTER(1) CHECK (c2 IN ('M', 'F')),  
d2 DATE);

CREATE TABLE t3 (  
an3 INTEGER,  
b3 INTEGER,  
c3 CHARACTER(5),  
d3 INTEGER);

CREATE JOIN INDEX ji_t1 AS  
SELECT a1, d1  
FROM t1  
WHERE b1 > 10  
AND c1 = 'Teradata');

CREATE JOIN INDEX aji_t3 AS  
SELECT a3, d3, COUNT(*)  
FROM t3  
WHERE b3 < 100  
GROUP BY 1, 2);

Given these table and join index definitions, the flow chart that follows shows how the Optimizer uses derived statistics to more accurately estimate cardinalities for the following query:

SELECT *  
FROM t1, t2, t3  
WHERE b1 > 10  
AND c1 = 'Teradata'  
AND b3 < 50  
AND d1 = d2  
AND a2 = a3  
AND d2 = d3;
The stages of deriving various statistics for this query are as follows:

1. Refine and derive base table statistics using join index statistics.
   - Cap the number of unique values in \( t1.d1 \) at 1 500 using the join index statistics from \( ji_{t1} \) on column \( d1 \) (see “Using Single-Table Sparse Join Indexes” on page 217).
   - No statistics have been collected on \( t2.(a2, d2) \), but statistics have been collected on a superset of those statistics, \((a2, b2, d2)\), so the cardinality of that superset is stored in the derived statistics for \( t2.(a2, d2) \) and propagated to the next stage of the process (see “Using Subset and Superset Statistics” on page 226).
   - Inherit the statistics for table \( t3 \), for which no statistics have been collected, from the aggregate join index \( aji_{t3} \) (see “Statistical Inheritance” on page 216 and “Using Aggregate Join Indexes” on page 218).

2. Join tables \( t1 \) and \( t2 \), consuming the term \( d1 = d2 \) and producing the interim join relation \( R1 \).
   
   \( d1 \) and \( d2 \) are then merged into an \( EquiSet \) (see “Using Join Predicate Redundancy After Each Binary Join” on page 228), which takes the smaller of the two unique values cardinalities as \( \text{MIN}(200, 1500) \), or 200.
   
   The entries for \( d1 \) and \( d2 \) are then removed from the derived statistics set.

3. Join table \( t3 \) with the interim join relation \( R1 \), consuming the terms \( a2 = a3 \) and \( d2 = d3 \) and producing the following set of derived statistics cardinalities for join relation \( R2 \):

<table>
<thead>
<tr>
<th>Column Set</th>
<th>Number of Unique Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>((b1))</td>
<td>1000</td>
</tr>
<tr>
<td>((c1))</td>
<td>2000</td>
</tr>
<tr>
<td>((d1, d2, d3))</td>
<td>200</td>
</tr>
<tr>
<td>((a2, a3))</td>
<td>100</td>
</tr>
<tr>
<td>((a2, d2))</td>
<td>600</td>
</tr>
<tr>
<td>((a3, d3))</td>
<td>600</td>
</tr>
</tbody>
</table>
Chapter 2: Query Rewrite and Optimization

Derived Statistics

where:

<table>
<thead>
<tr>
<th>Column heading</th>
<th>Represents the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col</td>
<td>column set for which the number of unique values is derived by the Derived Statistics subsystem.</td>
</tr>
<tr>
<td>NUV</td>
<td>number of unique values for a column set as derived by the Derived Statistics subsystem.</td>
</tr>
</tbody>
</table>
Statistical Inheritance

Single-table nonsparse join and hash indexes inherit their statistics from their underlying base table if the statistics have not been collected on the index itself. This applies to single-column statistics, single-column-index statistics, multicolumn statistics, and multicolumn index statistics.

The following bullets summarize the available forms of statistical inheritance:

- Underlying base tables can inherit the statistics from any single-table nonsparse join and hash indexes that reference them. In this case, it becomes possible to inherit statistics from multiple qualified join and hash indexes.
- Multicolumn statistics are inherited from an underlying base table by a single-table nonsparse join index or hash index defined on it.
- Partitioning columns can inherit the statistics from the system-generated PARTITION column, but only if the table is partitioned by a single column partitioning clause defined without any expressions.
- Bidirectional inheritance of statistics is implemented for single-table nonsparse join indexes and hash indexes so that statistics collected on the single-table nonsparse join or hash index are inherited by their underlying base table. For this case, it is possible for there to be multiple qualified single-table nonsparse join or hash indexes from which a base table could inherit statistics.
- Bidirectional inheritance of statistics is implemented for PARTITION statistics, where those statistics are inherited by the partitioning column of a PPI table if the table is partitioned by a single column without any expressions.

The inheritance of statistics occurs when the derived statistics are built for the base relations. All statistical inheritance is bidirectional.
Deriving Column Demographics

Introduction

The Optimizer acquires derived statistics in several different ways. The next several topics introduce the more common acquisition methods of deriving statistics:

- “Using Single-Table Sparse Join Indexes” on page 217
- “Using Aggregate Join Indexes” on page 218
- “Using CHECK Constraints and Referential Integrity Constraints” on page 219
- “Using Single-Table Predicates” on page 222
- “Using Single-Table Predicates in the Absence of Statistics” on page 224
- “Using Single-Table Predicates and Multicolumn (PARTITION) Statistics” on page 224
- “Using Single-Column and Multicolumn Statistics to Discover Hierarchical Relationships in Denormalized Schemas” on page 225
- “Using Subset and Superset Statistics” on page 226
- “Using Join Predicates” on page 227
- “Using Join Predicate Redundancy After Each Binary Join” on page 228

Note that the description of derived statistics continues with “Discovering Join Indexes” on page 231, which is complex enough to be segregated into its own topic.

Using Single-Table Sparse Join Indexes

If there are single-table sparse join indexes that either fully or partially cover the single-table predicates of a relation, the Optimizer can use them to derive the demographics of the residual columns. This mechanism provides an automated way of deriving column correlations using single-table join indexes.

For example, suppose you have defined the following join index:

```sql
CREATE JOIN INDEX ji AS
SELECT c1, d1
FROM t1
WHERE x1 > 10
AND y1 =10;
```

You then submit the following query:

```sql
SELECT t1.c1, COUNT (*)
FROM t1, t2
WHERE t1.d1 = t2.d2
AND t1.x1 > 10
AND t1.y1 = 10
GROUP BY 1;
```

Using the statistics on `ji`, the system can derive the demographics of the columns `t1.c1` and `t1.d1` after applying the single-table predicates `t1.x1 > 10` and `t1.y1 = 10`. The newly derived demographics can then be used in join planning for the join column `t1.d1` and in aggregate estimates to determine the number of groups for the grouping column `t1.c1`.

SQL Request and Transaction Processing
Note that partially covering join indexes can also used to derive the demographics in all applicable cases. For example, suppose you define the following join index:

```sql
CREATE JOIN INDEX ji AS
    SELECT c1, d1
    FROM t1
    WHERE x1 > 10;
```

You then submit the following query:

```sql
SELECT t1.c1, COUNT (*)
FROM t1, t2
WHERE t1.d1 = t2.d2
AND t1.x1 > 10
AND t1.y1 = 10
GROUP BY 1;
```

In this case, the demographic information from the partially covering join index is used for columns `c1` and `d1`. If the demographic information is available from both the base table and the join index, then the join index information is given higher precedence because it also captures the correlation information with respect to some single-table predicates.

### Using Aggregate Join Indexes

The Optimizer can derive the unique values of grouping columns from aggregate join indexes. The derived values can then be used in join planning. This is another strategy to derive the column correlation information among the grouping column set automatically, and also after applying the single-table predicates.

For example, suppose you have defined the following aggregate join index:

```sql
CREATE JOIN INDEX ji AS
    SELECT a1, SUM (b1)
    FROM t1
    WHERE x1 > 10
    AND y1 = 10
    GROUP BY c1, d1;
```

You then submit the following query:

```sql
SELECT *
FROM t1, t2
WHERE t1.c1 = t2.c2
AND t1.d1 = t2.d2
AND x1 > 10
AND y1 = 10;
```

The cardinality of the aggregate join index is the number of unique values of the columns (`t1.c1`, `t2.d1`). If the unique values for these columns are already available from base table statistics, the system updates them with the information from the aggregate join index; otherwise, it creates new derived statistics entries. The unique values can then be used in join planning for estimating join cardinality, rows per value, skew detection, and so on.

Note that partially covered aggregate join indexes are also handled in the same way as nonaggregate single-table sparse join indexes, and that the statistics from aggregate join indexes are used automatically if that index replaces a derived table or view.
Using CHECK Constraints and Referential Integrity Constraints

CHECK and referential integrity constraints can also provide useful information to the Optimizer in the absence of statistics.

For example, consider a column called `patient_sex`, which can have only two values: M and F. Generally, users enforce the sanity of this column with a database CHECK constraint such as `patient_sex IN ('M', 'F')`.

If there are no statistics on this column and a query specifies the predicate `patient_sex = 'M'`, the system would normally assume that 10 percent of the rows qualify by default because of the equality condition, but because this column can have only two values, a more reasonable default is 50 percent:

\[
\text{Total qualifying rows} = \text{Total rows} \times \frac{1}{\text{NumUniqueVals}} = 50 \text{ percent}
\]

The Optimizer also considers CHECK constraints such as open or closed ranges, and IN lists when statistics are not available. Open ranges help in some scenarios when a query predicate can close the range. For example, if a CHECK constraint specifies the open range `x1 > 10`, and a query specifies the predicate `x1 < 20`, their combination produces a closed range that can then be used to derive the column demographics.

Similarly, if there are no statistics on the child table in a Primary Key-Foreign Key relationship, and statistics exist for the parent table PK column, many of its demographics, such as number of unique values, can be derived and used for the FK column in the child table. This information can be used for single-table estimates and in join planning for join cardinality and rows per value estimates. These kinds of estimates can also be useful for large child tables in those situations where collecting statistics is prohibitively expensive.

This mechanism can be effectively used in conjunction with nonenforced Referential Constraint referential integrity constraints (also known as soft referential integrity constraints) to provide child table demographic information to the Optimizer without collecting statistics.

Note that this mechanism assumes that the child table has all the values of the parent table. If there is only a subset of values, or if the value set in this join column is significantly skewed, then you should collect statistics on the child column to avoid skewed redistributions and underestimates of cardinalities.

The derived statistics framework collects this kind of information from CHECK and referential constraints in the prejoin planning stage of query optimization and propagates them from there to the join planning stage.

The following table definitions are used for the query that illustrates the flow of derived statistics usage by the Optimizer to generate more accurate cardinality estimates:

```sql
CREATE TABLE t1 (  
a1 INTEGER,  
b1 INTEGER,  
c1 CHARACTER(5),  
d1 DATE);
```
CREATE TABLE t2 (
    a2 INTEGER PRIMARY KEY,
    b2 INTEGER,
    c2 CHARACTER(1) CHECK (c2 IN ('M', 'F')),
    d2 DATE);

CREATE TABLE t4 (
    a4 INTEGER REFERENCES t2.a2,
    b4 INTEGER,
    c4 CHARACTER(1),
    d4 DATE);

Given these table definitions, the flow chart that follows shows how the Optimizer uses derived statistics based on CHECK and referential integrity constraints to more accurately estimate cardinalities for the following query using:

```sql
SELECT * 
FROM t1, t2, t4 
WHERE c1 = 'MSFT' 
AND b1 > 30 
AND b4 = b1 
AND a1 = a4 
AND c2 = c4;
```

The stages of deriving various statistics for this query are as follows:

1. Refine and derive base table statistics using CHECK and referential integrity constraints.
   - No statistics have been collected on `t4.a4`, but statistics have been collected on `t4.a2`, and there is a referential integrity constraint from `t4.a2` to `t4.a4`.
     Using this RI constraint to infer that the unique cardinalities must be identical for the two columns, the system inserts the number of unique values for `t4.a2` into the derived statistics for `t4.a4` (see “Statistical Inheritance” on page 216 and “Using CHECK Constraints and Referential Integrity Constraints” on page 219).
   - No statistics have been collected on `t2.c2`, but `t2.c2` has a CHECK constraint that limits its possible values to only 2 (M or F), so the system inserts that value into the derived statistics for `t2.c2` (see “Using CHECK Constraints and Referential Integrity Constraints” on page 219).

2. Join tables `t1` and `t4`, consuming the terms `b4=b1` and `a1=a4` and producing the interim join relation `R1`.

3. Join interim relation `R1` with `t2`, consuming the term `c2=c4` and producing the join relation `R2` with the following set of derived statistics cardinalities:

<table>
<thead>
<tr>
<th>Column Set</th>
<th>Number of Unique Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b1)</td>
<td>1000</td>
</tr>
<tr>
<td>(c1)</td>
<td>2000</td>
</tr>
<tr>
<td>(a1, a4)</td>
<td>100</td>
</tr>
<tr>
<td>(b1, b4)</td>
<td>1000</td>
</tr>
<tr>
<td>(c2, c4)</td>
<td>2</td>
</tr>
</tbody>
</table>
Chapter 2: Query Rewrite and Optimization

Deriving Column Demographics

**Table: Column heading and their representations**

<table>
<thead>
<tr>
<th>Column heading</th>
<th>Represents the…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col</td>
<td>column set for which the number of unique values is derived by the Derived Statistics subsystem.</td>
</tr>
<tr>
<td>NUV</td>
<td>number of unique values for a column set as derived by the Derived Statistics subsystem.</td>
</tr>
</tbody>
</table>
Using Single-Table Predicates

Single-table estimation logic determines the probable number of qualified unique values and qualified high modal frequency, along with the number of qualified rows, for every predicate. The Optimizer can use this information for cases where a column specified in the single-table predicate is also specified in subsequent operations such as joins and aggregations.

There are two ways to derive this information:

- Using the base table statistics
- Using the statistics on a single-table join index

By using the base table statistics, the system can derive the column demographics using the single-table predicate on the column specified in the predicate. However, the derived information might not be adjusted later based on other single-table predicates. This is in line with the assumption of column independence in the absence of column correlation information.

For example, suppose you have the following query:

```sql
SELECT *
FROM t1, t2
WHERE t1.d1 = t2.d2
AND t1.d1 BETWEEN '1999-01-01' AND '2000-01-01'
AND t1.c1 > 10;
```

Using base table statistics, the qualified number of unique values and qualified high modal frequency are derived and saved in the derived statistics entry for column `t1.d1`.

On the other hand, by using the available single-table join indexes, the Optimizer can get the demographics of the columns after applying all single-table predicates.

For example, suppose you have defined the following join index to support the previous query:

```sql
CREATE JOIN INDEX ji AS
SELECT t1.d1
FROM t1
WHERE t1.d1 BETWEEN '1999-01-01' AND '2000-01-01'
AND t1.c1 > 10;
```

If an interval histogram is available for the join index column `t1.d1`, then it is given higher precedence and captured in the derived statistics because it reflects the demographics of the column `t1.d1` after both single-table predicates have been applied.

The derived information is then reused in join planning stages such as join cardinality estimation, rows per value estimation, and skew detection.

In general, the Optimizer considers any predicate that can be used for cardinality estimates for a query plan with the exception of complicated predicates. The following table provides some examples of predicates the Optimizer does consider and the statistics it uses to make cardinality estimates for those predicates:
### Predicate Definitions

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS NULL</td>
<td>Self-defined.</td>
</tr>
<tr>
<td>IS NOT NULL</td>
<td>- If nulls are present, the value is calculated as ((\text{TotalValues} - 1)).&lt;br&gt;- If no nulls are present, the value is just (\text{TotalValues}).</td>
</tr>
<tr>
<td>TotalRows - Number of Nulls</td>
<td></td>
</tr>
<tr>
<td>Single equality condition</td>
<td>Self-defined.</td>
</tr>
<tr>
<td></td>
<td>- If Loner/Mode is in any interval, the value is its frequency.</td>
</tr>
<tr>
<td></td>
<td>- If Loner/Mode is <em>not</em> in any interval, the value is calculated as (\frac{\text{TotalRows}}{\text{TotalValues}}).</td>
</tr>
<tr>
<td>Index with equality conditions on all columns</td>
<td>Self-defined.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NE</td>
<td>(\text{TotalValues} - 1)</td>
</tr>
<tr>
<td></td>
<td>HighModeFreq of the distribution excluding the HighModeFreq for this value.</td>
</tr>
<tr>
<td>LT</td>
<td>Values count derived from qualified intervals.</td>
</tr>
<tr>
<td>LE</td>
<td>HighModeFreq from the qualified intervals.</td>
</tr>
<tr>
<td>GT</td>
<td></td>
</tr>
<tr>
<td>GE</td>
<td></td>
</tr>
<tr>
<td>BETWEEN</td>
<td>(\text{LowValues} + \text{HighValues} - \text{TotalValues})</td>
</tr>
<tr>
<td></td>
<td>HighModeFreq from the qualified range intervals.</td>
</tr>
<tr>
<td>ORed term</td>
<td>The number of elements in the ORed list.</td>
</tr>
<tr>
<td>IN list</td>
<td>The maximum frequency of all the ORed elements.</td>
</tr>
<tr>
<td>NOT IN term</td>
<td>(\text{TotalValues} - \text{Number of elements in NOT IN list})</td>
</tr>
<tr>
<td></td>
<td>The HighMode after excluding all IN list elements.</td>
</tr>
</tbody>
</table>

An example of a term type that is *not* considered is ORed terms that reference multiple columns such as \(x_1 = 10 \text{ OR } y_1 > 20\).
Using Single-Table Predicates in the Absence of Statistics

If single-table predicates such as EQ, IN lists, ORed predicates or closed ranges, are specified in a query, some information can be derived, such as the number of unique values, and the derived information can then be used later in join planning if column statistics are not available.

An example of where such statistics would be derived is the following query:

```sql
SELECT *
FROM t1, t2
WHERE t1.d1 = t2.d2
AND t1.d1 in (10, 20);
```

Derived statistics captures and propagates the number of unique values information to the Join Planner based on certain single-table predicates even when no statistics have been collected on this column.

Using Single-Table Predicates and Multicolumn (PARTITION) Statistics

The demographics of the join columns are required after applying the single-table predicates to a join request. The Optimizer can use multicolumn statistics to derive the demographics of the join columns using some single-table predicates. This is another automated way of deriving column correlations using multicolumn statistics.

For example, suppose you submit the following query:

```sql
SELECT *
FROM t1, t2
WHERE t1.d1 = t2.d2
AND t1.x1 > 10;
```

The Join Planner needs to know the demographics of the join column $t1.d1$ after applying the single-table predicate $t1.x1 > 10$. Derived statistics derives the demographics of $t1.d1$ if there are multicolumn statistics ($x1, d1$) on $t1$. For PPI tables, if there is partition elimination and the multicolumn PARTITION statistics are available, then the demographics of the join columns are derived based on the qualified partitions.

Note that the order of the columns in multicolumn statistics is important for these estimates. The columns are reordered internally based on their ascending field IDs irrespective of the order you specify for collecting multicolumn statistics. So if column $d1$ has smaller field id than column $x1$, the multicolumn statistics are ordered internally as ($d1, x1$). In this case, it is not possible to derive the demographics of $d1$ for a given predicate on $x1$. Note that for multicolumn PARTITION statistics the PARTITION column is always the first column because it always has an internal field ID of 0.

Some of the single-table predicates the Optimizer considers for deriving the column correlations are equality conditions ($x1 = 10$), IN lists ($x1 IN (10, 20, 30)$), simple ORed lists on the same columns ($x1 = 10 OR x1 = 20$), and range predicates ($x1 BETWEEN 10 AND 30, x1 > 10$), and so on.
Using Single-Column and Multicolumn Statistics to Discover Hierarchical Relationships in Denormalized Schemas

The derived statistics framework discovers hierarchical relationships for denormalized tables using statistics. The relationships are saved as value mappings in the derived statistics. The only mappings used to adjust demographics are those with 1 \( \rightarrow \) n relationships.

A combination of single and multicolumn statistics is required to detect these relationships. A change to a single column cascades through the entire hierarchical chain. For example, if the value mapping for \( x \rightarrow y \) is 1 \( \rightarrow \) 5 and the value mapping for \( y \rightarrow z \) is 1 \( \rightarrow \) 10, then if one value is removed from \( x \), 5 values are removed from \( y \), and 10 values are removed from \( z \).

More concretely, if the relationship between region \( \rightarrow \) nation is discovered to be 1 \( \rightarrow \) 5, then if one region is selected, the Optimizer detects that only 5 nations qualify. In other words, if one region is disqualified by a single-table predicate, the Optimizer removes 5 nations from the nation column.

The following example illustrates this logic. Consider the denormalized dimension table customer_nation_region, which is defined as follows:

```sql
CREATE SET TABLE cust_nation_region (  
c_custkey INTEGER,  
c_name VARCHAR(25) CHARACTER SET LATIN CASESPECIFIC,  
c_address VARCHAR(40) CHARACTER SET LATIN CASESPECIFIC,  
c_nationkey INTEGER,  
c_phone CHAR(15) CHARACTER SET LATIN CASESPECIFIC,  
c_acctbal DECIMAL(15,2),  
c_mktsegment CHAR(10) CHARACTER SET LATIN CASESPECIFIC,  
c_comment VARCHAR(117) CHARACTER SET LATIN CASESPECIFIC,  
c_maritalstatus CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,  
n_name CHAR(25) CHARACTER SET LATIN CASESPECIFIC,  
n_comment VARCHAR(152) CHARACTER SET LATIN CASESPECIFIC,  
r_regionkey INTEGER,  
r_name CHAR(25) CHARACTER SET LATIN CASESPECIFIC,  
r_comment VARCHAR(152) CHARACTER SET LATIN CASESPECIFIC)  
PRIMARY INDEX (c_custkey);
```

In this example, the relationships are derived from a combination of single- and multicolumn statistics. The following statistics are required to discover the relationships:

- Single-column on `r_regionkey`
- Single-column on `r_regionkey`
- Single-column on `r_regionkey`
- Multicolumn on `(r_regionkey, n_nationkey)`
- Multicolumn on `(n_nationkey, c_custkey)`

The relationships discovered are:

- region \( \rightarrow \) nation is 1 \( \rightarrow \) 5
- nation \( \rightarrow \) customer is 1 \( \rightarrow \) 24,000
You can see an example of this in the following query, which specifies a predicate that selects only one `regionkey` value, and its EXPLAIN text:

```sql
EXPLAIN SELECT n_nationkey
FROM cust_nation_region
WHERE r_regionkey = 1
GROUP BY 1;
```

Explanation

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>First, we lock a distinct TPCD_OCES3.&quot;pseudo table&quot; for read on a RowHash to prevent global deadlock for TPCD_OCES3.cust_nation_region.</td>
</tr>
<tr>
<td>2</td>
<td>Next, we lock TPCD_OCES3.cust_nation_region for read.</td>
</tr>
<tr>
<td>3</td>
<td>We do an all-AMPs SUM step to aggregate from TPCD_OCES3.cust_nation_region by way of an all-rows scan with a condition of (&quot;TPCD_OCES3.cust_nation_region.R_REGIONKEY = 1&quot;), grouping by field1 (TPCD_OCES3.cust_nation_region.N_NATIONKEY).</td>
</tr>
</tbody>
</table>

Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with low confidence to be 5 rows (525 bytes). The estimated time for this step is 0.62 seconds.

-> The total estimated time is 0.63 seconds.

From the known hierarchical relationship between region and nation, which is 1 → 5, the predicted number of nationkey rows is 5, which is verified by the EXPLAIN text highlighted in boldface type. By generalizing, if you were to write a request that selected two `regionkey` values, there would be 10 nationkey rows in the result set, and so on.

### Using Subset and Superset Statistics

If there are multiple join predicates, and the join columns have some degree of correlation, multicoloumn statistics are necessary to more accurately estimate join cardinality, rows per value, skew detection, and related statistics. Without having multicoloumn statistics, and assuming column independence, the Optimizer multiplies the individual numbers of unique values to get an estimate of the combined number of unique values. However, if multicoloumn statistics exist that are a superset of the join columns, they might provide more useful information about the column correlations than the column independence assumption would provide.

For example, consider the following query:

```sql
SELECT *
FROM t1, t2
WHERE t1.d1 = t2.d2
AND t1.x1 = t2.x2;
```

The system requires multicoloumn statistics on `(t1.d1, t1.x1)` and `(t2.d2, t2.x2)` to be able to make accurate cardinality and costing estimates. If there are individual statistics on `t1.d1`, with 100 unique values, and `t1.x1`, with 50 unique values, and assuming column independence, the unique values are multiplied to calculate the combined maximum number of unique values, $100 \times 50 = 5,000$, and having an upward bound of the total number of rows.

The Derived Statistics subsystem identifies such opportunities in the prejoin planning stage and derives dummy multicoloumn statistics based on supertsets. For example, if there is a multicoloumn statistics collection that is a superset, such as `(x1, y1, d1)`, with only 2,000 unique values, the Optimizer can cap the estimated unique value cardinality of 5,000 to a more accurate 2,000 by deriving a derived statistics entry `(x1, y1)` with 2,000 unique values, and then propagate the statistic to Join Planning.
Similarly, if there are individual entries and multicolunm statistics entries such as \((x_1), (d_1)\) and \((x_1, d_1)\) and single-table predicates such as \(x_1 \text{ IN } (1, 2) \text{ AND } d_1 < 20\), the Optimizer can update the demographics of the single-column entries while making its single-table estimates. Based on the updated information from the single-column entries, the multicolunm entry of derived statistics is updated and propagated.

**Using Join Predicates**

The Optimizer assumes join uniformity when it estimates join cardinalities and costs. In other words, it assumes that every value from the left relation in the join finds a match in the right relation if the number of values in the left relation is less than the number of right values in the right relation, and vice versa.

Based on this assumption, and given the join predicate \(t_1.d_1 = t_2.d_2\), if \(t_1.d_1\) has 100 unique values and \(t_2.d_2\) has 50 unique values, then after the join, \(t_1.d_1\) should have only 50 unique values. The Optimizer also considers multicolunm statistics for making adjustments. For example, given the following predicate and \((x_1, y_1)\) with 100 unique values and \((x_2, y_2)\) with 50 unique values, after the join, the number of \((x_1, y_1)\) unique values is adjusted to 50:

\[
t_1.x_1=t_2.x_2 \text{ AND } t_1.y_1=t_2.y_2
\]

The Derived Statistics subsystem performs this kind of analysis and adjusts the demographics of the join columns such as the number of unique values and the high modal frequency after each binary join and then propagates the adjusted demographic information to the next stages of the join planning process.

If column correlation information is available, the Optimizer uses it to adjust the other column demographics based on the adjustments to the join column. Otherwise, the join column is assumed to be independent of the other columns.

Assume that \(t_1.d_1\) has 100 unique values, \(t_2.d_2\) has 50 unique values, and \(t_3.d_3\) has 200 unique values.

Suppose you submit the following query:

```sql
SELECT t3.d3, COUNT (*)
FROM t1, t2, t3
WHERE t1.d1 = t2.d2
AND t1.d1 = t3.d3
GROUP BY 1;
```

Assuming the join order is \((t_1 \times t_2) \times t_3\), after the first join, the Derived Statistics subsystem adjusts the number of unique values of \(t_1.d_1\) to the minimum number of unique values \((t_1.d_1, t_2.d_2) = 50\).

After the second join, the Derived Statistics subsystem adjusts the number of unique values of \(t_3.d_3\) to 50, which is then used in the final aggregate estimate because \(t_3.d_3\) is specified as a grouping column.
Using Join Predicate Redundancy After Each Binary Join

Some join predicates can become redundant\(^{54}\) after the system has made one or more binary joins. The redundancy can result from either Transitive Closure or from user-defined query conditions. If such redundant predicates are not properly identified and handled during cardinality estimates and costing, they can have a negative effect on the optimization of the query, possibly leading to nonoptimal query plans.

For example, suppose you submit the following query:

```sql
SELECT *
FROM t1, t2, t3
WHERE t1.d1 = t2.d2
AND t1.d1 = t3.d3;
```

Transitive closure (see “Satisfiability and Transitive Closure” on page 102) derives a new predicate `t2.d2 = t3.d3` for this request. Assuming the join order `(t1 \times t2 \rightarrow j1) \times t3`, the join predicates for the second join are `j1.d1 = t3.d3` and `j1.d2 = t3.d3`. While estimating the number of unique values of `(d1,d2)` for join `j1`, the Optimizer needs to be aware that these two columns had already been equated in the previous join, and it should not multiply their individual unique values to get the combined number of unique values. Instead, it should use `\text{MIN}(j1.d1, j1.d2)` as the combined number of unique values.

The Derived Statistics infrastructure builds the appropriate entries by combining all the connected joined columns into EquiSets after every binary join. This way, the subsequent joins seamlessly handle the redundancy.

In join planning, an EquiSet is a set of columns that was equated in predicates from a previous join operation in the same query. Propagating EquiSets to subsequent join operations for reuse in a query is a fundamental component of the derived statistics framework.

For example, in the preceding example, an EquiSet derived statistics entry `(j1.d1, j1.d2)` exists after join `j1`, with the minimum number of values of `d1` and `d2`. When the subsequent join requests the unique values for the combination `(j1.d1, j1.d2)` the Optimizer automatically uses the existing EquiSet entry.

Handling Multiple Sources of Information

As demonstrated in the preceding sections, information such as the number of unique values and the high modal frequency can be taken or derived from multiple sources such as single-table join indexes, multicolumn statistics, random AMP samples, and interval histograms, among others. It is not trivial to determine which of the available sources is likely to be the most useful or trustworthy. The following criteria all play an important role in making the determination:

- The information source captures the highest correlation.
- The information source covers the greatest number of single-table predicates.
- The information source is the most current (or, phrased another way, the least stale) source of information available.

\(^{54}\) A predicate is redundant when it adds nothing to the overall selectivity for a query because its effect is equivalent to one or more other predicates that are also specified in that query.
For example, it is possible to have a join index producing noncorrelated demographic information covering multiple single-table predicates, while at the same time there might be a multicolumn statistic covering one single-table predicate that produces highly correlated information for a join column. Because of issues such as this, it is not possible to define precedence rules based on heuristics. To work around this restriction, the Optimizer quantifies the information and defines precedence dynamically based on the derived quantities.

- The first two criteria, highest correlation and greatest number of single-table predicates, can be quantified by using the number of unique values. This translates to using the information source that provides the fewest unique values.
- The third criterion, least stale statistics, is also considered and explained in “Stale Statistics” on page 251.

**Propagating Column Demographics To All Temporary and Committed Joins**

Column demographics are propagated to all temporary and committed joins using the derived statistics framework. As a result, demographics can be dynamically adjusted after each join, and the Join Planner does not need to reference base table interval histograms to retrieve the statistics.

**Propagating Column Demographics For Materialized Instances Within a Query Block**

If you specify complex predicates or outer joins in a query, they might become materialized into spool relations while the Optimizer is performing join planning within a query block.

Suppose you submit the following outer join query with a complex ON clause predicate:

```sql
SELECT *
FROM t1 INNER JOIN t2 ON (x1 = x2)
  LEFT OUTER JOIN t3 ON (x2 NOT IN (SELECT x4
                          FROM t4)
                        OR y3 = 10);
```

The ON clause predicate $x2 \text{ NOT IN } (SELECT x4 \text{ FROM } t4) \text{ OR } y3 = 10$ in this query makes this a complex outer join.55 One of the requirements for processing a complex outer join is to have only one left relation, so the left relations ($t1, t2$) are materialized into a single spool relation before the system processes the complex join. After materializing the left relations, Derived statistics propagates the projected column demographic information for the materialized instance to the subsequent stages of join planning.

55. If a subquery is specified in an ON clause, the predicate is classified as either semicomplex or complex.
Propagating Column Demographics For Aggregate and Ordered-Analytic Function Estimates

To be able to do aggregate and ordered-analytic cardinality estimates after a series of joins, the demographic information of the grouping columns needs to be propagated through the join operations until final aggregations are done. The Derived Statistics subsystem adjusts the demographic information based on single-table and join predicates and then propagates it forward until final aggregation completes.

For example, suppose you submit the following query:

```
SELECT t1.x1, t2.x2, COUNT(*)
FROM t1, t2
WHERE t1.x1 = 10
AND t1.y2 = t2.x2
GROUP BY 1, 2;
```

In the prejoin planning stage, derived statistics detects the single-table equality term on column \( x1 \) \((t1.x1=10)\) and adjusts the number of unique values to 1. Also while doing join planning, the number of unique values of \( t2.x2 \) is adjusted to the minimum number of values of \((t1.y2, t2.x2)\). This improves the aggregate cardinality estimates, which can benefit the overall plan if this query block is a spooled derived table or a view.

Propagating Column Demographics Across Query Blocks

Suppose you submit the following query:

```
SELECT *
FROM t1, (SELECT x2, x3, COUNT(*)
FROM t2, t3
WHERE x2 = 10
AND t2.x3 = t3.x3
GROUP BY 1, 2) AS dt
WHERE t1.x1 = dt.x3;
```

The derived statistics framework carries the column demographic information of the derived table \( dt \) for columns \( x2 \) and \( x3 \) by making adjustments based on the derived table predicates. The adjusted demographics are then propagated to the join planning for the outer block.
Discovering Join Indexes

Introduction

To support the use of join indexes for derived statistics, the Optimizer finds all single-table sparse join indexes whose cardinality and statistics are potentially useful for estimating values such as the selectivity of base table conditions and column correlations. The system also gathers together any nonsparse single-table join indexes and hash indexes that can be used for inheriting statistics by the base table. For a given query, the Optimizer identifies all single-table join indexes that can cover a query predicate.

The SELECT list and GROUP BY clause of the query are ignored at the discovery phase, and early coverage analysis is based only on its predicates. The Optimizer captures and retains the column mapping information from join indexes to the base table field IDs so they are available for qualifying the join index for different derived statistics applications.

The Optimizer searches for join indexes whose statistics or cardinality or both are usable for single-table condition selectivity estimates once per flattened query block in the prejoin planning stage. The qualified join indexes are then stored and can later be used for the planning of different join index rewrites.

For example, assume a query references tables t1, t2, t3, t4 with single-table join indexes j1 and j3 defined on t1 and t3, respectively.

The Join Planner can use these join indexes in the following three ways:

- The original query \{t1, t2, t3, t4\} can use the statistics from both j1 and j3.
- The rewritten query \{j1, t2, t3, t4\} can use the statistics from j3.
- The rewritten query \{t1, t2, j3, t4\} can use the statistics from j1.

Using Join Indexes for Single-Table Cardinality Estimations

The following cases describe how the nonaggregate single-table sparse join indexes are used to help make single-table cardinality estimates.

- The join index predicates are a subset or identical to those of the query predicates. For example, suppose you have the following query and join index definitions:

  ```sql
  SELECT * 
  FROM t1 
  WHERE a = 10 
  AND b BETWEEN 20 AND 30;
  ```

  ```sql
  CREATE JOIN INDEX ji AS 
  SELECT a 
  FROM t1 
  WHERE b BETWEEN 20 AND 30;
  ```
Even without an interval histogram on \( ji1 \), the Optimizer can use the cardinality of the join index as the number of rows selected by the predicate \( b \text{ BETWEEN 20 AND 30} \). The selectivity of the predicate \( b \text{ BETWEEN 20 AND 30} \) can be calculated by dividing the join index cardinality by the base table cardinality. However, without the base relation interval histograms, the qualified number of values and the high modal frequency of the column set must be derived from other sources.

The Optimizer can use a histogram on columns projected from the join index to estimate the selectivity of the residual conditions of a query that are not covered by the conditions of that join index. The combined selectivity of the matching and residual conditions that use join index statistics is the selected rows cardinality determined by applying residual terms divided by the base table cardinality.

For the preceding example of \( ji1 \), the Optimizer can use the histogram on \( ji1.a \) to find the number of rows that qualify \( a=10 \), which is also the number of rows that qualify for the combined predicate \( a=1 \text{ AND } b \text{ BETWEEN 20 AND 30} \).

Therefore, the combined selectivity is calculated as follows:

\[
\text{Combined selectivity} = \frac{\text{Cardinality}_{ji.a=10}}{\text{Cardinality}_{\text{base table}}}
\]

where:

| Equation element ... | Specifies the number of rows ...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Cardinality}_{ji.a=10} )</td>
<td>from ji that satisfy the predicate ( ji.a=10 ).</td>
</tr>
<tr>
<td>( \text{Cardinality}_{\text{base table}} )</td>
<td>in the base table.</td>
</tr>
</tbody>
</table>

- The join index contains all rows required to answer the query, but its predicates do not match the query predicates. In this case, the join index is usable for cardinality estimation purposes only if it has interval histograms for all the columns referenced in the nonmatching conditions of the query. For example, suppose you have the following query and join index definitions:

```sql
SELECT * 
FROM t1 
WHERE a=10 
AND b BETWEEN 20 AND 30 
AND c=11;
```

```sql
CREATE JOIN INDEX ji2 AS 
SELECT a 
FROM t1 
WHERE a>5 
AND b BETWEEN 20 AND 30;
```

The Optimizer uses \( ji2 \) for selectivity estimation only when there is an interval histogram on \( ji2.a \), and the combined selectivity of \( a=10 \text{ AND } b \text{ BETWEEN 20 AND 30} \) must be calculated using \( ji2 \).
The combined selectivity of \( a=10 \) AND \( b \) BETWEEN 20 AND 30 is, as before, calculated using the following equation:

\[
\text{Combined selectivity} = \frac{\text{Cardinality}_{ji.a=10}}{\text{Cardinality}_{\text{base table}}}
\]

The selectivity calculated using the cardinality or interval histogram of one join index is combined with selectivity calculated from other sources such as base table interval histograms or another join index.

In this case, although \( ji2 \) cannot be used to calculate the selectivity of \( a=10 \) AND \( b \) BETWEEN 20 AND 30, the cardinality of \( ji2 \) can be used as the upper bound on the rows selected by the predicates \( a=10 \) AND \( b \) BETWEEN 20 AND 30 or the entire WHERE clause for the query.

### Derive Column Correlations From Join Index Statistics

The Optimizer uses the number of unique values and the high modal frequency of join columns to make join cardinality, rows per value, skew, and other estimates. For example, consider the following query:

```sql
SELECT *
FROM t1, t2
WHERE x1= x2
AND y1=10;
```

The number of unique values on \( x1 \) after the predicate \( y1=10 \) is applied is crucial data for more accurately estimating the join cardinality, rows per value, and so on. The number of unique values of \( x1 \) after applying the predicate \( y1=10 \) is called the correlated values of \( x1 \) given \( y1=10 \).

A base table histogram on \( x1 \) can only provide the estimate before condition \( y1=10 \) is applied. Even a multidimensional histogram on \( (x1, y1) \) might not always provide accurate correlation information because of information missing from the multicolumn statistics such as column ordering, length limits on concatenated column values, and so on.

Single-table join indexes can be used to calculate qualified demographic information for join columns or grouping columns after applying the single-table conditions. In the previous example, if there is a join index \( ji3 \), defined as follows, it is possible to use the interval histogram for \( ji3.x1 \) to make an accurate estimate of the number of unique values and high modal frequency on \( x1 \):

```sql
CREATE JOIN INDEX ji3 AS
SELECT x1
FROM t1
WHERE y1=10;
```

Using the interval histogram for \( ji3.x1 \), the Optimizer can make accurate estimates of the number of unique values and high modal frequency of \( x1 \). This information is collected and propagated by the derived statistics framework.
The system can use a join index for deriving the column correlations when either of the following conditions is satisfied:

- The join index predicates are a subset of the query predicates.
- The join index predicates are an exact match to the query predicates.

When the join index has the same set of predicates as the single-table predicates in the query, the column correlation estimates are very accurate. In this case, the join index contains more rows than necessary to answer the query (satisfying the first bullet in the list), the estimate based on the join index histogram is still better than using the base table interval histogram statistics because the base table histogram represents the entire domain, whereas the join index histogram represents the domain only after applying single-table conditions.

**Using Aggregate Join Indexes to Derive Column Demographic Information**

The number of unique values plays an important role for estimating the join cardinality, rows per value, or number of rows returned by an aggregation operation or query. For example, suppose you have the following query:

```sql
SELECT x, y, SUM(z) FROM t1 WHERE a = 10 AND c > 20 GROUP BY x, y;
```

The cardinality of this query equals the number of unique values of \((x, y)\) after applying the WHERE clause predicate \(a = 10 \text{ AND } c > 20\). Two kinds of join indexes are useful for estimating cardinalities for this query:

- A join index with the same single-table predicate and GROUP BY columns as the join index \(ji4\).
  
  The cardinality of \(ji4\) is equal to the cardinality of the preceding query.

  ```sql
  CREATE JOIN INDEX ji4 AS
  SELECT x, y, COUNT(z) FROM t1 WHERE a = 10 AND c > 20 GROUP BY x, y;
  ```

- A join index with the same single-table predicate as the join index and specifying any GROUP BY clause.

  The cardinality of the join index, which can be determined either by random AMP sampling or from its primary index statistics, is the number of unique values of the grouping columns after applying the single-table conditions. The derived unique values can then be used in the join planning of the query. For example, the number of unique values of \((x1, y1)\) derived from the cardinality of \(ji5\) can be used in the join cardinality estimation of the query.
Consider the following query and join index definitions:

```sql
SELECT * 
FROM t1, t2 
WHERE x1 = x2 
AND y1 = y2 
AND a = 10 
AND c > 20;
```

```sql
CREATE JOIN INDEX ji5 AS 
SELECT x1, y1 
FROM t1 
WHERE a = 10 
AND c > 20 
GROUP BY 1, 2;
```

The Optimizer allocates an entry for the grouping columns of a query in the derived statistics framework if one does not already exist. Its number of unique values is the cardinality of the applicable aggregate join index, and that value is propagated forward so it can be used later in join planning.

When the aggregate join index predicates are a subset of the query predicates, the Optimizer can still use the estimate of the number of unique values derived from the aggregate join index because it is more reasonable than the estimate from the base table interval histogram.

### Single-Table Cardinality Estimates

Single-table cardinality estimation is independent of the Access Path Planner. Such cardinality estimates can be derived from any of the following bases:

- Single-table join indexes for cardinality estimates.
- Derived statistics, which has the information from domain-based constraints such as CHECK and referential integrity constraints.
- Complex single-table predicates.
- Combined multiple selectivities can handle overlapping and selectivities derived from different sources such as join index interval histograms and derived statistics.
- Whenever possible, column correlations collected from single-column and multicolumn statistics are used in combining selectivities.

### Using Single-Table Join Indexes

Single-table cardinality estimation uses single-table join indexes.
Using Histograms for Cardinality Estimation

An expression that can use histograms for cardinality estimation can contain either a unary or a binary operator. One of the operands must be either a column or a built-in functions or SQL operator from the following lists:

- UPPER
- LOWER
- NULLIFZERO/ZEROIFNULL
- SUBSTR
- MOD
- CONCAT on columns of same table
- Implicit or explicit data type conversions on a column

The supported operators are the following:

- =, >=, <=, >, <, <>
- IS [NOT] NULL
- [NOT] LIKE
- BETWEEN
- IN

For binary operators, the other operand can be a simple expression involving any of the following:

- Constants whose value can be calculated by the Parser.
- System USING request modifier data or built-in function data such as a CURRENT_DATE value.
- Simple expressions involving another column from the same table.

For example, the selectivity of the single-table predicate \( t1.x1 = t1.y1 \) can be estimated more reasonably by considering the overlapping values of those two columns than by using a default selectivity formula.

Selectivity estimates for LIKE predicates is limited to "abc%" patterns and very conservative estimation formulas.

The default ratio for “abc%” patterns can be adjusted in your Cost Profile by setting the value for the LIKEEstPartialIntRatio flag to enable LIKE predicate selectivity to be derived from a partially covering interval in an interval histogram (see “Cost Profile Flags” on page 277). LIKEEstPartialIntRatio specifies the fraction of the applicable rows that the system assumes satisfies the constraint. The applicable rows can be the entire table for a predicate like \( \text{WHERE descrip LIKE 'abc%'} \), or they can be a subset of the table for a predicate like \( \text{WHERE lastname LIKE 'abc%'} \). In the case of a proper subset, and having collected histogram statistics, the Optimizer considers only the applicable intervals. The default value is 0.125 of the qualified rows.
This default lowers the predicted number of rows that match a LIKE condition from the Type 1 default, and biases join plans toward a duplicated join geography (see “Join Distribution Strategies” on page 333). If you experience a performance regression because the system underestimates the number of rows matching a LIKE condition using the default LIKEEstPartialIntRatio value of 0.125, which then causes a different spool table redistribution to be chosen by the Optimizer, you can create a variable cost profile with a larger LIKEEstPartialIntRatio value to compensate.

There can be no single optimal value for this flag. The best value for LIKEEstPartialIntRatio depends on the table and the constant in a particular SQL request. You would anticipate, for example, that constraints such as WHERE sex LIKE 'fe%' and WHERE lastname LIKE 'jingleheimershmmit%' would differ greatly in their respective selectivities, and it is not possible for one value of LIKEEstPartialIntRatio to be the single best value for all SQL requests across all system workloads.

The following table indicates one possible way to handle the issue of widely varied predicate selectivities for the typical workloads submitted by multiple users:

<table>
<thead>
<tr>
<th>IF your site uses ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiple cost profiles for different users</td>
<td>you can specify different values for LIKEEstPartialIntRatio for different users as necessary to support the different query workloads routinely submitted by those users.</td>
</tr>
<tr>
<td>a single cost profile for all users</td>
<td>the value you specify for LIKEEstPartialIntRatio is used system-wide for all SQL requests.</td>
</tr>
</tbody>
</table>

The Optimizer considers full and partial coverage of all covering intervals individually, making it possible to make the selectivity calculations for “abc%” patterns still more accurate.

The Optimizer can use a multicolumn histogram for a group of predicates when the following statements are all true:

- The predicates are specified on the first \( n \) fields of the multicolumn histogram.
  
  This rule exists because there is an ordering dependency of the fields of a multicolumn histogram.

- The predicates must specify an equality condition except for the first field of the multicolumn histogram.

- If the predicate on the first field of a multicolumn is a nonequality condition, then the Optimizer uses the multicolumn histogram for this predicate only.

For example, a histogram on \((x, y, z)\) can be used to estimate the selectivity for predicate \(x > 100\) as well as \(x = 10\) AND \(y = 20\).

The Optimizer can also use the data derived by date extrapolation to enhance its cardinality estimates for date-related predicates (see “Using Extrapolation to Replace Stale Statistics” on page 254).
Combining Two Selectivities

The Optimizer can detect independent columns and calculate their combined selectivity as the product of their individual selectivities. Two categories of independent columns are defined:

- Every value of one column maps to all values of the other column.
  
  In this case, the number of combined values is the product of the number of values of individual column. Take column `nation` and column `market-segment` of the `customer` table as an example. Each nation participates in the business of all market segments, while business in each market segment is provided in all countries.

  During cardinality estimation, two columns are considered to be independent if their number of combined values is close to the number of values of the individual column. That is, column `c1` and `c2` are independent if their number of unique values satisfies the following inequality:

\[
\text{UniqueValues}(c1,c2) \geq \text{UniqueValues}(c1) \times \text{UniqueValues}(c2) \times \text{Multi2SnglValueRatio}
\]

where:

<table>
<thead>
<tr>
<th>Equation element</th>
<th>Specifies ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniqueValues (c1,c2)</td>
<td>Combined number of unique values in the column set (c1,c2).</td>
</tr>
<tr>
<td>UniqueValues (c1)</td>
<td>Number of unique values in column c1.</td>
</tr>
<tr>
<td>UniqueValues (c2)</td>
<td>Number of unique values in column c2.</td>
</tr>
<tr>
<td>Multi2SnglValueRatio</td>
<td>The value of the Multi2SnglValueRatio flag in your cost profile. By default, this value is 0.9.</td>
</tr>
</tbody>
</table>

- The value of one column is not constrained by the value of the other column, but given a value of the first column of two independent columns, the possible value of the first column is evenly distributed across all values of the second column.

  Column `market-segment` and column `account-balance` of the `customer` table are an example of such independent columns. On the one hand, the account balance is not bounded by a specific market segment even though one account balance, say $100.23, does not exist in all market segments.

  The following example is a sample of the histogram for `(account-balance, market-segment)`:

  Interval[12]  
  Max. Value: -341.39, 'AUTOMOB'  
  Mode Value: -349.49, 'MACHINER'  
  Interval[13]  
  Max. Value: -281.02, 'FURNITUR'  
  Mode Value: -312.03, 'MACHINER'  
  Interval[14]  
  Max. Value: -226.32, 'BUILDING'  
  Mode Value: -259.31, 'BUILDING'  

  ...  
  Interval[183]  
  Max. Value: 9082.16, 'MACHINER'  
  Mode Value: 9057.08, 'AUTOMOB'
For this kind of value independency, the Optimizer uses double safety tests to detect value independence. For the two algorithms described in the following paragraphs, assume that single-column statistics have been collected on ($c_1$), ($c_2$) and multicolmn statistics have been collected on ($c_1$,$c_2$).

- **Algorithm 1**
  
  If two columns are independent, given any two widely separated values of $c_1$, the value ranges of a corresponding column $c_2$ still overlap with the two values of $c_1$. This can be done by sampling two blocks of equal-height intervals in multicolmn statistics.
  
  Assume the first block is from interval $k$ to $k+n$, and the second block is from interval $j$ to $j+n$. Make sure that the two blocks have some gaps such that $j > k + n + p$. Check whether the ranges of $c_2$ in those two blocks of intervals overlap.
  
  For instance, in the histogram of (account-balance, market-segment), the value range of market-segment in Interval 12 to Interval 14 is from 'AUTOMOBIL' to 'MACHINER', while that of Interval 183 to Interval 185 is also 'AUTOMOBIL' to 'MACHINER'.
  
  The value ranges overlap, so the algorithm is valid.

- **Algorithm 2**
  
  If two columns are independent, then given any value of $c_1$, the range of $c_2$ should be closed to the whole range of $c_2$.
  
  This can be done by finding the max/min value of column $c_2$ from Interval $k$ to Interval $k+n$ of multicolmn stats ($c_1$,$c_2$), and then calculating the selectivity of $c_2$ BETWEEN min AND max using the histogram for $c_2$.
  
  The two columns are independent if the selectivity of $c_2$ BETWEEN min AND max is greater than CCFIndependentValueRatio, which by default is 0.6. The value of the CCFIndependentValueRatio flag can be adjusted in your cost profile.
  
  For instance, in the histogram of (account-balance, market-segment), the minimum value of market-segment in both interval blocks is 'AUTOMOBIL' and the maximum value is 'MACHINER'. The selectivity of market-segment BETWEEN 'AUTOMOBIL' AND 'MACHINER' is 1.0 from the histogram on the market-segment column.
  
  To be conservative, the system assumes that two columns are independent only if the outcomes of both Algorithm 1 and Algorithm 2 determine that the two columns are independent. The algorithm can also be used for two-column sets, not just two single columns.
There are three cases where the Optimizer cannot detect column independence:

- When the total size of all columns exceeds the histogram data row size limit of 16 bytes. In this case, multicolored histogram does not have all the information necessary to make an accurate determination of column independence, so the test for independence could fail.
- When independence detection is activated only when the confidence of the individual selectivities is high. That is, for predicates whose selectivity is not high confidence because no statistics have been collected on the columns, or because an expression is complex, the Optimizer does not attempt to detect column independence.
- When independence detection is activated only when the selectivity estimation of a predicate on an individual column is based on the base table histogram.

**Combined Selectivity for the Value-Mapping Case**

To more easily study this case, assume that both column $X$ and column $Y$ are evenly redistributed.

1. Given that $Y$ value $d_1$ maps to $X$ value $(c_1, c_2, ..., c_m)$, it is possible to calculate the number of rows selected by the predicate $(y = d_1 \text{ AND } x = c_i)$ similar to the second case in the previous topic.
Because $Y$ is evenly distributed, each $Y$ value maps to the same number of rows, which is the RowsPerValue, or RPV.

From the perspective of the whole table,

$$
\text{NumRowsSelected}((y = d1 \text{ AND } x = c_i) = S_y \times \frac{\text{TotalRows}}{m} = S_y \times \frac{(N_y \times \text{RPV}_y)}{m}
$$

From the perspective of the rows selected by the predicate $(x = c_i)$,

$$
\text{NumRowsSelected}(y = d1 \text{ AND } x = c_i) = S'_y \times \text{RowsSelected}(x = c_i)
$$

$$
= S'_y \times \frac{(\text{NumYValsRowsSelectedBy}(x = c_i))}{\text{RPVRowsSelectedBy}(x = c_i)}
$$

$$
= S'_y \times n \times \frac{\text{RPV}_y}{\text{NumXValsPerYVal}} = S'_y \times n \times \frac{\text{RPV}_y}{m}
$$

Because the same set of rows is selected in both cases,

$$
S_y \times \frac{(N_y \times \text{RPV}_y)}{m} = S'_y \times \left(n \times \left(\frac{\text{RPV}_y}{m}\right)\right)
$$

Rearranging and eliminating terms,

$$
S'_y = S_y \times \frac{N_y}{n}
$$

and

$$
S'_x = S_x \times \frac{N_x}{m}
$$

This means that the combined selectivity is calculated in either of the following ways:

$$
\text{Combined selectivity} = S_x \times S_y \times \frac{N_y}{n}
$$

$$
\text{Combined selectivity} = S_x \times S_y \times \frac{N_x}{m}
$$

When both $X$ and $Y$ are evenly distributed and the value mapping from $X$ to $Y$ is $m:n$, it follows that $\frac{N_x}{N_y} = \frac{m}{n}$ and therefore $\frac{N_y}{n} = \frac{N_x}{m}$.

To avoid the anomaly, the combined selectivity has an upper bound equal to the more highly selective of the two:

$$
\text{Selectivity}(P_x \&\& P_y) \leq \text{MIN}(S_x, S_y)
$$

This is because a conjoined predicate should never increase the selectivity of the other term.

End of process.
The following table summarizes these results:

<table>
<thead>
<tr>
<th>Predicate Px and Py from same column?</th>
<th>Selectivity(Px AND Py)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>0</td>
<td>Except range constraints</td>
</tr>
<tr>
<td>No</td>
<td>$S_x \times S_y \times N_y/n$ or $S_x \times S_y \times N_x/m$</td>
<td>- When value mapping from $X$ to $Y$ is $m:n$. - If $\text{Selectivity}(\text{Px AND Py}) &gt; \text{Min}(S_x, S_y)$, then set it to $\text{MIN}(S_x, S_y)$</td>
</tr>
<tr>
<td>No</td>
<td>$\text{AndFF}(\text{MIN}(S_x, S_y), \text{MAX}(F, S_x, S_y))$</td>
<td>A stochastic model based selectivity combination algorithm in $\text{AndFF}()$ is used when no column correlation is defined or discovered between $X$ and $Y$, where $F$ is the fudge factor.</td>
</tr>
</tbody>
</table>

The combined selectivity of two ORed predicates can be derived from their corresponding ANDed predicates:

$$\text{Selectivity}(\text{Px OR Py}) = S_x + S_y - \text{Selectivity}(\text{Px AND Py})$$

**Join Predicate Redundancy Handling**

A predicate is redundant if the evaluation of other predicates determines an identical result. If redundant predicates are not detected and handled properly during join planning, they can adversely affect cardinality and costing estimates, leading to nonoptimal query plans. Redundancy can occur either from internally-generated predicates derived from Transitive Closure (see “Satisfiability and Transitive Closure” on page 102) or from user-specified SQL predicates.

Transitive closure generates new predicates if the set of predicates given in the query logically implies other predicates. The added predicates provide more opportunities for single-table access paths and join planning.

For example, given the following query:

```sql
SELECT *
FROM t1, t2, t3
WHERE t1.x1 = t2.x2
AND t2.x2 = t3.x3;
```

Through Transitive Closure, the system can generate a third predicate, as follows:

```sql
SELECT *
FROM t1, t2, t3
WHERE t1.x1 = t2.x2
AND t2.x2 = t3.x3
AND t1.x1 = t3.x3; ← Derived predicate
```
Assume the join plan is as follows:

\[
\begin{align*}
& j_1 \ (t_1 \ \bowtie \ t_2) \ \text{ON} \ (x_1 = x_2) \\
& j_2 \ j_1 \ \bowtie \ t_3 \ \text{ON} \ (x_2 = x_3 \ \text{AND} \ x_1 = x_3)
\end{align*}
\]

where \( \bowtie \) indicates the JOIN operator.

When joining \((j_1 \ \bowtie \ t_3)\), one of the pair of join conditions is redundant, because evaluating either individually produces the same result set. This is because the previous join had already equated \(x_1\) to \(x_2\) using the condition \((x_1 = x_2)\).

The Optimizer algorithms that use the number of unique values on join columns recognize this, and consider the number of unique values for column combination \((j_1.x_1, j_1.x_2)\) as the minimum number of values of \(x_1\) and \(x_2\) based on the join uniformity assumption. Redundancy detection logic recognizes this, so it ignores the condition that has the greater number of unique values for unique value estimates.

In this example, if \(t_1.x_1\) has more unique values, then the Optimizer ignores the last condition, \((x_1 = x_3)\).

Redundancy detection works with derived statistics as follows.

After each join, whether temporary or committed, the Optimizer takes the following actions:

1. The connected components, or equated columns, using join predicates are grouped together and marked as EquiSets (see “Using Join Predicate Redundancy After Each Binary Join” on page 228 for a definition of EquiSets). These groups are either added as EquiSet entries to the derived statistics or merged with other EquiSets if they intersect. The individual column entries that formed the EquiSet are then removed from further consideration in the join plan.

   The demographic information such as number of unique values and High Modal Frequency is derived as follows and saved as part of derived statistics entry:

   \[
   \begin{align*}
   \text{Number of unique values} & = \text{Minimum number of values for all equated columns} \\
   \text{High modal frequency} & = \text{High modal frequency of the minimum number of values}
   \end{align*}
   \]

   Demographic information from EquiSet entries can be used either for an exact match or for a match on any subset of its columns. The next stage in the process is made aware of the EquiSet entries to enable it to make use of them in the most efficient way it can. The EquiSets also help aggregation estimates and outer block join planning because the system propagates them to outer blocks of the query.

   For example, after join \(j_1\) in the previous example, the EquiSet is \((j_1.x_1, j_2.x_2)\). This set can then be used to derive column demographics for \(j_1.x_1, j_2.x_2\), and the combination \((j_1.x_1, j_2.x_2)\).
The Join Planner might need to know the demographics of the different combinations of columns. To help the Optimizer with subsequent join planning, many combinations of entries can be derived if an EquiSet intersects with other multicol...
The processing of this query is shown in the following graphic:

<table>
<thead>
<tr>
<th>Graphic element</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUV</td>
<td>number of unique values.</td>
</tr>
</tbody>
</table>
Finding the Number of Unique Values

Join cardinality estimates, rows per value estimates, skew detection, aggregate estimates, Partial Group By estimates, and several other computations all require estimating the number of unique values for a given set of join or grouping columns.

The Optimizer makes an exhaustive search of all possible combinations of statistics to determine the best set of nonoverlapping statistics. Once it has been decided, that set is then used to find the Number of Unique Values, High Modal Frequency, and High AMP Frequency values at all stages of the optimization process.

The major goals of this algorithm are as follows:

- Find the set of nonoverlapping combinations of statistics that has the least number of unique values.
  
  If no complete coverage can be found, find the set of nonoverlapping combinations of statistics that covers the largest number of columns and has the least number of unique values.

- Find the set of nonoverlapping combinations of statistics that provides the highest High Modal Frequency and High AMP Frequency values by covering all the columns.

The set of combinations that provides the smallest Number of Unique Values might not be the same set that provides the best estimates for the High Modal Frequency and High AMP Frequency values because the High Modal Frequency and High AMP Frequency might not be available for some of the entries that are derived from sources other than base table interval histograms.

The number of unique values determined from partial statistics, where statistics do not cover all the hash or join columns, is considered for Rows Per Value and join cardinality estimates because it is a conservative approach, but the Optimizer does not consider this estimate for skew adjustment and Partial GROUP BY because it makes very aggressive cost estimates and, as a result, can cause performance problems when the costing estimates are grossly in error because of the overly aggressive method by which they were calculated. In other words, the Optimizer avoids Partial GROUP BY plans and does not attempt to make skew adjustments when the given hash or join columns are not fully covered by statistics.

The unique values discovery algorithm provides the following information:

- MinVals and its confidence level
  
  This estimate provides the absolute minimum number of values for the given collection of columns. The values are taken from a single derived statistics entry that covers the largest number of columns. If there are multiple entries that cover the same number of columns, the Optimizer selects the entry with the highest number of values.

The confidence levels for various entries are described in the following table:

<table>
<thead>
<tr>
<th>IF a usable derived statistics entry is ...</th>
<th>The confidence level is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>found</td>
<td>High confidence</td>
</tr>
<tr>
<td>not found</td>
<td>No confidence</td>
</tr>
</tbody>
</table>
• **BestVals and its confidence level**
  This provides the best number of values estimate that can be derived from the set of derived statistics entries that meets both of the following criteria:
  • Covers the greatest number of columns
  • Produces the least number of values
  Derived statistics entries that are either the same set, or a subset, of the given column collection are used to produce these values.
  The values can even be taken from a set of derived statistics entries that covers only a portion of the columns in some cases. This can happen when, for example, there are insufficient statistics to cover all the columns.
  The confidence levels for various entries are described in the following table:

<table>
<thead>
<tr>
<th>FOR this derived statistics entry situation ...</th>
<th>The confidence level is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a single entry covers all the columns</td>
<td>High confidence</td>
</tr>
<tr>
<td>either of the following:</td>
<td></td>
</tr>
<tr>
<td>• multiple entries must be combined to cover all the columns</td>
<td>Low confidence</td>
</tr>
<tr>
<td>• only a subset of the columns is covered</td>
<td></td>
</tr>
<tr>
<td>no usable derived statistics entries are found</td>
<td>No confidence</td>
</tr>
</tbody>
</table>

• **MaxVals and its confidence level**
  This estimate provides the maximum number of possible values for the given collection of columns. The derived statistics entries that are either a subset, a superset, an EquiSet, or intersecting entries are all considered for producing these values.
  If all columns are not covered by these entries, then default estimates based on the domain type are used for the noncovered columns, as described in the following table:

<table>
<thead>
<tr>
<th>FOR this derived statistics entry situation ...</th>
<th>The confidence level is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a single entry covers all the columns</td>
<td>High confidence</td>
</tr>
<tr>
<td>multiple entries must be combined to cover all the columns</td>
<td>Low confidence</td>
</tr>
<tr>
<td>default estimates are used to compute the values</td>
<td>No confidence</td>
</tr>
</tbody>
</table>

If MinVals is found with a High or Low level of confidence, then the Optimizer can always determine a BestVals statistic. However, it is also possible to determine MaxVals with High or Low confidence, but not be able to determine a BestVals or MinVals.

The Optimizer uses BestVals for its join cardinality and RowsPerValue estimates if it has Low or High confidence.

Partial GROUP BY, aggregations, and skew detection always use MaxVals.
The values are combined in two levels using a stochastic model:

1. The values from the same source are combined.
   The number of combined values is limited to the total rows of the source.

2. The combined values from different sources are themselves combined to get the final values.
   These are limited based on the total rows for the current set.

The system consumes the EquiSet entries and then derives additional combinations if any multicolumn statistics intersect with the EquiSets. For example, if the usable entries for the given hash columns \((x_1, x_2, y_2)\) are EquiSet \([x_1, x_2]\) and \((x_2, y_2)\]), then the Optimizer augments the multicolumn statistics entry \((x_2, y_2)\) with other EquiSet columns, producing the new entry \((x_1, x_2, y_2)\) dynamically.

The following examples describe the different possibilities.

**Example 1**
Consider the following derived statistics entries:

<table>
<thead>
<tr>
<th>Column Set</th>
<th>Number of Unique Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a_1, b_1))</td>
<td>10</td>
</tr>
<tr>
<td>((b_1, c_1))</td>
<td>15</td>
</tr>
<tr>
<td>((a_1, b_1, c_1))</td>
<td>20</td>
</tr>
</tbody>
</table>

If the hash columns are \((a_1, b_1, c_1)\), then the following values and confidence levels are derived by the Optimizer:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
<th>Confidence Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinVals</td>
<td>20</td>
<td>High confidence</td>
</tr>
<tr>
<td>BestVals</td>
<td>20</td>
<td>High confidence</td>
</tr>
<tr>
<td>MaxVals</td>
<td>20</td>
<td>High confidence</td>
</tr>
</tbody>
</table>
**Example 2**
Consider the following derived statistics entries:

<table>
<thead>
<tr>
<th>Column Set</th>
<th>Number of Unique Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a_1, b_1))</td>
<td>10</td>
</tr>
<tr>
<td>((c_1))</td>
<td>5</td>
</tr>
</tbody>
</table>

If the hash columns are \((a_1, b_1, c_1, d_1)\), then the following values and confidence levels are derived by the Optimizer:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
<th>Confidence Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinVals</td>
<td>10</td>
<td>High confidence</td>
</tr>
<tr>
<td>BestVals</td>
<td>50(^a)</td>
<td>Low confidence</td>
</tr>
<tr>
<td>MaxVals</td>
<td>combination of 50 and the demographic estimate for (d_1).</td>
<td>No confidence</td>
</tr>
</tbody>
</table>

\(^a\) Calculated from the product of the numbers of unique values for the column sets: 10 x 5 = 50.

**Example 3**
Consider the following derived statistics entry:

<table>
<thead>
<tr>
<th>Column Set</th>
<th>Number of Unique Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a_1, b_1, c_1, d_1))</td>
<td>100</td>
</tr>
</tbody>
</table>

If the hash columns are \((a_1, b_1, c_1)\), then the Optimizer derives the following values and confidence levels:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
<th>Confidence Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinVals</td>
<td>TotalRows</td>
<td>No confidence</td>
</tr>
<tr>
<td>BestVals</td>
<td>TotalRows</td>
<td>No confidence</td>
</tr>
<tr>
<td>MaxVals</td>
<td>100</td>
<td>Low confidence</td>
</tr>
</tbody>
</table>
The following flow chart shows the framework used to determine all these estimates and their accompanying costing logic:

Find number of unique values, high mode frequency and high AMP frequency given field collection

Field Collection

Single field? Yes

Get info from DS

No

Collect all usable derived statistics entries (including USable entries found?)

Full cover entry found? Yes

No

Usable entries found? No

Yes

Pick up NUV, HMF, HAF, etc

Single entry found? Yes

No

Combine NUV, HMF, HAF, etc

No overlapping entries? Yes

Find combination of DS entries that gives best NUV, HMF, and HAF

No

Exit

where:

<table>
<thead>
<tr>
<th>Graphic element</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>Derived Statistics</td>
</tr>
<tr>
<td>NUV</td>
<td>Number of Unique Values</td>
</tr>
<tr>
<td>HMF</td>
<td>High Modal Frequency</td>
</tr>
<tr>
<td>HAF</td>
<td>High AMP Frequency</td>
</tr>
</tbody>
</table>
Stale Statistics

Introduction

Interval histogram statistics are the initial source of column demographic information for the Optimizer to use in making its estimates of join cardinality, rows per value, skew adjustments, and several other pieces of information necessary for accurate query optimization.

The Optimizer also employs more intelligent methods of using statistics such as deriving column correlations using multicolumn statistics, adjusting and propagating the number of unique values after each binary join is made, and so on. The methods are developed and generalized in a consistent manner throughout the Optimizer.

Cost and cardinality estimation formulas and the strategies for deriving column correlations depend on the accuracy of the available statistics to produce optimal plans. The software might not produce reasonable plans if its statistical inputs are not accurate. The most common reason statistics are not accurate is because they are stale.

The process of detecting stale statistics and extrapolating data values (see “Using Extrapolation to Replace Stale Statistics” on page 254) when the Optimizer determines that the statistics it needs are stale is designed to reduce the frequency of collecting or recollecting statistics.

Derived Statistics and Stale Statistics

The definition of stale statistics and the problems faced by the Optimizer when dealing with them were introduced earlier in this chapter (see “How the Optimizer Determines Whether To Use Interval Histogram Statistics or a Random AMP Sample for Base Table Cardinality Estimates” on page 144, “How the Optimizer Determines Whether To Use Interval Histogram Statistics or a Random AMP Sample for NUSI Subtable Statistics” on page 147, and “An Example of How Stale Statistics Can Produce a Poor Query Plan” on page 150).

The following text describes the four principal ways the cardinality estimation subsystem handles the problem of stale statistics:

• Adjusting the total cardinality
  The Optimizer compares the row count from a random AMP sample against the row count from the interval histogram, and the final cardinality estimate is determined based on the following rules if the deviation is more than a predefined tunable Cost Profile parameter percentage.56
  • If the primary index is not skewed57 and if the deviation between the sample and the interval histogram row count is more than the defined Cost Profile percentage, the sample row count is taken as the cardinality of the table.
  • If no interval histogram on the primary index is available, the sampled row count is taken to be the table cardinality as is done by the current Optimizer software.

56. The default variation that can be tolerated is set to 5 percent.
57. This can be identified from the high modal frequency value stored in the interval histogram.
• Adjusting the number of unique values:
  • For indexes, the cardinality is estimated as the total number of unique values of the indexed columns.
    In this case, the number of unique values is adjusted in the same way as the total rows described in the first bullet under Adjusting the total cardinality, above.
  • If the adjusted table row count deviates more than a defined percentage from the histogram row count on unindexed columns, then the number of unique values, which is the principal value input to the join cardinality and costing related formulas, is, assuming a uniform distribution, either scaled up or down for the following scenarios:
    • The histogram is on a DATE column.
    • The histogram is on multiple columns that have a DATE column component.
    • The histogram is on a unique index.
    • The histogram is on a softly unique column.
      Soft uniqueness is defined as the case where the interval histogram statistics indicate that the number of unique values is close to the total cardinality for the table.

The scaling operation assumes a constant number of rows per value. For example, assume that the table cardinality is estimated to be 100 rows from a random AMP sample. If, for a column $x1$, the histogram indicates that the row count is 50 and the number of unique values is 25, meaning the number of Rows Per Value is 2, then the number of unique values is scaled up to:

$$\frac{\text{Cardinality}}{\text{Average rows per value}} = \frac{100}{2} = 50.$$ 

Note that the value scaling, whether up or down, cannot be done unconditionally for all columns. For example, columns like product_id, business_unit_id, and so on cannot have new adjusted values added to them because it makes no semantic sense.58

• Considering statistics collection timestamps
  While inheriting the statistics from the single-table nonsparse join or hash index by the parent base table and vice versa, when statistics are available for a column in both the source and destination interval histograms, the system uses the most recent statistics of the two as determined by a timestamp comparison.

• Adjusting the confidence level
  If the available histogram was created from sampled statistics, then the system always lowers the their confidence level.

You should always keep the statistics for skewed columns and indexes current.

The handling of stale statistics for single-table cardinality estimates is done using various extrapolation methods.

58. Their values are not only arbitrary, but also are not cardinal numbers, so it is meaningless to perform mathematical manipulations on them, particularly in this context.
Optimized Join Cardinality Estimates When There Is Single-Row Access to One of the Tables Using a Unique Index

If one of the source tables of a join has single-row access using a unique index, the Optimizer retrieves the row during the optimization phase, substitutes the values of the columns from the retrieved row, and makes its cardinality estimates based on the substitution.

For example, if a join condition has a form like

```
fact_table.yr_wk BETWEEN calendar.ytd_beg_wk AND calendar.ytd_end_wk
```

and the `calendar` table has single-row access path using either a UPI or a USI, the Optimizer fetches the row during the optimization phase, substitutes the actual values from the retrieved row for `ytd_beg_wk` and `ytd_end_wk`, and estimates the cardinality of the join based on the substitution.

```
EXPLAIN
SELECT *
FROM ordertbl, calendar
WHERE yr_wk BETWEEN ytd_beg_wk AND ytd_end_wk ← join predicate
AND calendar_date = 970101; ← UPI access to calendar table
```

Explanation

```
3) We do a single-AMP RETRIEVE step from TPCD_OCES3.calendar by way of the unique primary index "TPCD_OCES3.calendar.calendar_date = DATE '1997-01-01'" with no residual conditions into Spool 2 (all_amps), which is duplicated on all AMPs. The size of Spool 2 is estimated with high confidence to be 14 rows (1,134 bytes). The estimated time for this step is 0.01 seconds.
4) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an all-rows scan, which is joined to TPCD_OCES3.order_75pct by way of an all-rows scan with no residual conditions. Spool 2 and TPCD_OCES3.order_75pct are joined using a product join, with a join condition of "((TPCD_OCES3.ordertbl.YR_WK >= Ytd_Beg_wk) AND (TPCD_OCES3.ordertbl.YR_WK <= Ytd_End_wk))". The input table TPCD_OCES3.ordertbl will not be cached in memory, but it is eligible for synchronized scanning. The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 365,114 rows (140,934,004 bytes). The estimated time for this step is 2.55 seconds.
```

As you can see in step 4 of the query plan, the cardinality estimate for the range join predicate is computed by substituting the values from the row of the `calendar` table that was retrieved in step 3.
**Using Extrapolation to Replace Stale Statistics**

**Introduction**

Users often query their databases over date intervals for which one or both bounds on a predicate are dates for rows that have been inserted into a table since the last time statistics were collected for the date columns specified for that table in the predicate.

Statistics extrapolation enables you, for example, to submit a query specifying a date range predicate in which one or all of the specified dates is past the latest date stored in the statistical histograms for the DATE column set. To support such queries, the Optimizer applies an extrapolation technique to make a reasonable estimate for the rows that are inserted after statistics have been collected without requiring statistics to be recollected. Extrapolation methods for statistics can be viewed as a form of derived statistics (see “Derived Statistics” on page 206).

Note that the definition of a future date applies only to dates occurring between the time statistics were last collected on a DATE column and the current date. In other words, *future* is a narrowly defined, relative term, and extrapolation does *not* apply to data having true future dates that cannot exist at the time you submit your date-extrapolation-requiring queries. Rather, the term applies to dates that are otherwise in a statistical limbo.

The ability of the Optimizer to extrapolate statistics does *not* remove the need to recollect statistics.

**Relative Accuracy of Residual Statistics Versus Random AMP Sampled Statistics for Static Columns**

The relative accuracy of residual\(^{59}\) statistics for static columns (see “Rolling Columns and Static Columns” on page 258) with respect to a random AMP sample,\(^{60}\) and whether residual statistics should be considered to be stale, depends on several factors. Residual statistics are not necessarily also stale statistics.

For example, if the relative demographics for a column set do not change, then residual statistics are normally a reliable representation of current demographics. The comparison made by the derived statistics framework between residual statistics, if they exist, and a random AMP sample makes the final determination of whether the Optimizer considers the residual statistics to be stale or not.

It is possible for the cardinality of a table to grow by more than 10 percent,\(^{61}\) 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.

---

59. *Residual* meaning existing statistics that were not collected recently, implying the likelihood they no longer provide an accurate picture of column data. This is not necessarily a correct assumption, and if you are in doubt, you should use the Teradata Statistics Wizard to make a more accurate determination of the freshness of residual statistics.

60. In this section, the term *random AMP sample* refers to random *single* AMP sampling only.
When this is true, residual statistics can still be more accurate than a newly collected random AMP sample, though even a single-AMP random AMP sample, depending to some degree on whether the column being sampled is indexed or not, typically provides excellent cardinality estimates (see, for example, “Random AMP Sampling of NUSIs” on page 183).  

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 some significant number of 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, and the derived statistics framework will detect this.

On the other hand, consider a join scenario in which the Optimizer needs to estimate how many rows will qualify for spooling. 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.

Without the derived statistics framework being able to detect whether the residual statistics are stale or not, working with them could have meant that the Optimizer would have chosen a bad query plan instead of a new, faster plan. Or, worse still, the residual statistics could produce a new join plan that is much slower than the previous plan.

About Extrapolating Statistics

When data distribution is relatively uniform within the collected set of statistics, extrapolation can provide a reasonably accurate estimate of future cardinalities. However, if there are many spikes in the existing statistics, or if they are overly skewed, then you should recollect statistics on the column or index because extrapolation can produce inaccurate estimates in such situations and cannot be relied upon to produce accurate cardinality estimates.

61. This so-called Ten Percent Rule also applies at the partition level for partitioned primary indexes. If the number of changed partitions exceeds 10 percent of the total number of partitions in the PPI (in other words, if more than 10 percent of the rows are added to or deleted from a partition), then statistics should be recollected on the index. For PPI tables, any refreshment operation should include the system-derived PARTITION column.

62. The accuracy of the statistics collected from a random AMP sample also depends to a relatively small degree on the number of AMPs sampled, which is determined by the setting of an internal DBS Control flag. The possibilities range through 1, 2, or 5 AMPs, all AMPs on a node, or all AMPs on a system. Consult your Teradata support representative for details.
Chapter 2: Query Rewrite and Optimization
Using Extrapolation to Replace Stale Statistics

The Optimizer can only apply extrapolation for statistics on a single column or single-column index on a column. The performance impact for applying extrapolation in a request is minimal.

The Optimizer performs the following extrapolations in the order indicated when it determines that the statistics it needs are stale:

1. Extrapolation of table cardinalities.
2. Extrapolation of distinct values for rolling columns.
3. Extrapolation of rows for rolling and static columns.

See “Rolling Columns and Static Columns” on page 258 for definitions of these terms.

Note that the system only extrapolates row and distinct value cardinalities upward.

The most obvious application of extrapolated statistics, deriving extrapolated date column statistics to better optimize range queries over a future date or dates (see, for example, “Case 2: Closed Range Date Predicate 2” on page 262), improves the accuracy of the cardinality estimates the Optimizer uses to create the portion of the query plan for a request that applies to the given date predicate. For this application, the term future means any rows inserted into a table between the date statistics were last collected on the relevant date column or index and its extrapolated boundary date.

Extrapolating statistics does not do any of the following things:

- Change the procedure for collecting or dropping statistics
- Store the extrapolated statistics in the relevant interval histogram
- Affect the type of information reported by a HELP STATISTICS request

Note that date extrapolation does affect the specific cardinality information reported for a query EXPLAIN report, not by adding new terminology, but because of the different, and more accurate, cardinality estimates it produces.

The Optimizer considers extrapolating date statistics only when it detects that the existing statistics on the relevant date column or index are stale. In this particular case, the term stale applies to statistics that predate any of the values specified in a predicate that ranges over date values.

**Cost Profile Flag for Extrapolation**

You can control extrapolation of statistics by setting the ExtrapolateStatistics flag in your Cost Profile, which allows setting the ExtrapolateStatistics flag to TRUE or FALSE. The default setting is TRUE, meaning that extrapolation is enabled.

You can control whether extrapolation of statistics is enabled or not by setting the ExtrapolateStatistics flag in your Cost Profile. If the settings for ExtrapolateStatistics differ in the Cost Profile and the DBS Control record, the system defaults to the specification made for the Cost Profile. Extrapolation is enabled by default.
### Extrapolated Boundary for Date Statistics

The following terminology is used for the examples of extrapolated date statistics:

<table>
<thead>
<tr>
<th>Term</th>
<th>Equation Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extrapolated boundary</td>
<td>e</td>
<td>The upper boundary for extrapolated statistics for future date values.</td>
</tr>
<tr>
<td>Lower boundary on collected</td>
<td>l</td>
<td>The oldest date for which statistics have been collected on date values of a</td>
</tr>
<tr>
<td>statistics</td>
<td></td>
<td>column or index.</td>
</tr>
<tr>
<td>Upper boundary on collected</td>
<td>h</td>
<td>The most recent date for which statistics have been collected on date values</td>
</tr>
<tr>
<td>statistics</td>
<td></td>
<td>of a column or index.</td>
</tr>
<tr>
<td>Number of unique values</td>
<td>v</td>
<td>The number of unique values among the collected statistics value set.</td>
</tr>
<tr>
<td>Extrapolated number of unique</td>
<td>v′</td>
<td>The additional number of unique values in the extrapolated statistics</td>
</tr>
<tr>
<td>values</td>
<td></td>
<td>as estimated from a random AMP sample.</td>
</tr>
</tbody>
</table>

The equation for calculating the extrapolated boundary for date values is as follows:

\[ e = h + (h - l) \times \left( \frac{v'}{v - 1} \right) \]
where:

<table>
<thead>
<tr>
<th>Equation element …</th>
<th>Specifies the …</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e )</td>
<td>extrapolated boundary. This is the date beyond which no extrapolation of cardinality estimates is done.</td>
</tr>
<tr>
<td>( h )</td>
<td>highest, or most recent, date defining the interval over which statistics have been collected.</td>
</tr>
<tr>
<td>( l )</td>
<td>lowest, or least recent, date defining the interval over which statistics have been collected.</td>
</tr>
<tr>
<td>( v' )</td>
<td>extrapolated number of unique date values over the interval between the date statistics were last collected and the extrapolated boundary date. This is estimated by taking a random AMP sample on the date column or index.</td>
</tr>
<tr>
<td>( v )</td>
<td>number of unique date values in the existing statistics interval histogram defined by the interval between ( l ) and ( h ), inclusive.</td>
</tr>
</tbody>
</table>

### Rolling Columns and Static Columns

The Optimizer extrapolates cardinalities using different methods depending on whether the values being extrapolated are from a rolling column or a static column. The following table defines these terms:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rolling column</td>
<td>A column with a constant number of rows per value such as dates and timestamps. For these columns, the demographics of existing data never changes and only new data gets new distinct values. For example, suppose you have a column order_date in an orders table. The demographics of the old order dates never change, and only new orders are added to the table.</td>
</tr>
<tr>
<td>Static column</td>
<td>A column with varying rows per value such as product_id, business_unit_id, dept_id, and the like, and a constant number of unique values. The Optimizer assumes that few new values are inserted into a static column across time. For this type of column, newly added data can change the demographics of the historical or existing data. For example, orders that are newly added to an orders table can change the number of orders for a given product id. When a static column has stale statistics, the Optimizer can make a good extrapolated estimate if the system can estimate the table cardinality growth accurately.</td>
</tr>
</tbody>
</table>
The following graphic illustrates examples of extrapolation for rolling and static columns:

**Rolling Column**: Constant rows per value

**Before extrapolation**

<table>
<thead>
<tr>
<th>Distinct values</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

**After extrapolation**

<table>
<thead>
<tr>
<th>Distinct values</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>

**Static Column**: Varying rows per value

**Before extrapolation**

<table>
<thead>
<tr>
<th>Distinct values</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

**After extrapolation**

<table>
<thead>
<tr>
<th>Distinct values</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>

If the Optimizer detects stale statistics for a rolling column, it extrapolates the number of distinct values assuming that the number of rows per value is constant. The following chart provides a rough approximation of the algorithm used to estimate the extrapolated cardinalities of rolling and static columns:

- **Predicate**
- **Is predicate a rolling column?**
  - **Yes**
    - **Does cardinality from random-AMP sample deviate from histogram cardinality by more than threshold?**
      - **Yes**
        - **Extrapolate values**
      - **No**
        - **Do not extrapolate values**
  - **No**
    - **Number of distinct values**
    - **Generate query plan**
For static columns with stale statistics, the Optimizer can use extrapolation to replace the stale statistics with an extrapolated estimate provided that the added cardinality resulting from table growth can be estimated accurately.

Extrapolation thresholds are required to guard against inaccuracies in random-AMP samples. The threshold flags are set in your Cost Profile. The extrapolation threshold flags and their defaults are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>RASHistDeviationPct</td>
<td>Percent deviation between the table cardinality estimated from a random-AMP sample and the table cardinality stored in the interval histogram. Extrapolation is used to estimate the true cardinalities for predicates based on rolling columns when this percentage deviation is exceeded; otherwise, the Optimizer does not extrapolate new cardinality estimates.</td>
<td>10 percent</td>
</tr>
<tr>
<td>RASHistAbsDeviation</td>
<td>Absolute numeric deviation between the table cardinality estimated from a random-AMP sample and the table cardinality stored in the interval histogram. Extrapolation is used to estimate the true cardinalities for predicates based on rolling columns when this many rows of deviation is exceeded; otherwise, the Optimizer does not extrapolate new cardinality estimates.</td>
<td>10,000 rows</td>
</tr>
</tbody>
</table>

Note that all-AMP sampling overrides the default thresholds. The default settings for these flags can only be adjusted by Teradata support personnel.

The extrapolation logic for rolling columns has the following properties:

- Acts only when the Optimizer detects a stale histogram whose statistics are needed for a request.
- Calculates an upper boundary to limit the extrapolation to values it can estimate with confidence (see “Extrapolated Boundary for Date Statistics” on page 257).
- Estimates and then adds “missing” rows for both closed and open range intervals (see “Case 1: Closed Range Date Predicate 1” on page 261 - “Case 5: Open Range Predicate 3” on page 264).
- Extrapolates values for columns with very large highest date values, which translates to a wide final interval in the histogram (see “Case 6: Open Range Predicate With Wide Final Interval” on page 265, for example).

For example, if the interval histogram indicates the table cardinality as 500 rows with 250 unique values, but the table cardinality is actually 1,000 rows, the system scales the number of unique values up to 500 by assuming that the constant number of rows per unique value is 2, so \( \frac{2 \text{ rows}}{\text{value}} \times 250 \text{ unique values} = 500 \text{ unique values.} \)

63. A static column is one having a constant number of unique values, but a varying number of rows per value.
Furthermore, for any rolling column defined to be UNIQUE, the Optimizer scales the extrapolated number of distinct values upward to equal the table cardinality by default.

**Case 1: Closed Range Date Predicate 1**

Consider the following request over the closed range bounded on the lower end by July 17, 2007 and on the upper end by July 23, 2007:

```sql
SELECT *
FROM ordertbl
WHERE o_orderdate BETWEEN DATE '07-17-2007'
AND DATE '07-23-2007';
```

Statistics have been collected on `o_orderdate` over the date range of January 1, 2007 through July 19, 2007, inclusive.

Substituting the dates for the existing upper and lower bounds on collected statistics, the known number of unique values in those statistics, and the random AMP sample-based estimate of the number of additional unique values into the equation for determining a date extrapolation boundary, you obtain the following result:

\[
e = 2007 - 07 - 19 + \left( \left( 2007 - 07 - 2007 - 07 - 01 - 01 \right) \times \frac{20 \times 10^6}{200 \times 10^6} \right) = 2007 - 08 - 08
\]

So based on the assumptions made for calculating an extrapolated upper boundary, it is determined that the extrapolated boundary for the new statistics is August 8 2007, and the Optimizer estimates an approximate cardinality of 7 million rows for the result, which is more than twice the cardinality that would be estimated without extrapolation, which is 3 million rows.

The closed range predicate in the request is covered within this extrapolated boundary,

<table>
<thead>
<tr>
<th>Query date range</th>
<th>Data table (days)</th>
<th>Statistics collected (Average rows per day = 1 million for 200 days)</th>
<th>Extrapolated boundary = 08/08/07</th>
</tr>
</thead>
<tbody>
<tr>
<td>07/17/07</td>
<td>01/01/07 – 07/19/07</td>
<td>Estimated number of rows = 7 million</td>
<td>Extrapulated number of rows = 20 million</td>
</tr>
<tr>
<td>07/23/07</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Case 2: Closed Range Date Predicate 2

Consider the following request over the closed range bounded on the lower end by August 6, 2007 and on the upper end by August 11, 2007:

```sql
SELECT *
FROM ordertbl
WHERE o_orderdate BETWEEN DATE '08-06-2007'
    AND DATE '08-11-2007';
```

Statistics have been collected on `o_orderdate` over the date range of January 1, 2007 through July 19, 2007, inclusive.

Substituting the dates for the existing upper and lower bounds on collected statistics, the known number of unique values in those statistics, and the random AMP sample-based estimate of the number of additional unique values into the equation for determining a date extrapolation boundary, the extrapolated boundary for the new statistics is determined to be August 8, 2007, just as in “Case 1: Closed Range Date Predicate 1” on page 261.

Unlike the situation for Case 1, however, the upper bound on the closed range of the predicate falls outside the extrapolated boundary date for the statistics. The Optimizer is still able to estimate an approximate cardinality of 3 million rows for the result, which is 6 orders of magnitude better than the estimate that would be made without extrapolating statistics to the August 8, 2007 date, which would have been 1 row.
Case 3: Open Range Date Predicate 1

Consider the following request over the open range bounded on the lower end by July 16, 2007 and unbounded on the upper end:

```sql
SELECT *
FROM ordertbl
WHERE o_orderdate >= DATE '07-16-2007';
```

Statistics have been collected on `o_orderdate` over the date range of January 1, 2007 through July 19, 2007, inclusive.

Substituting the known data into the equation for determining a date extrapolation boundary returns August 8, 2007, as it does for all these date range predicate examples.64

Unlike the previous cases, the upper bound on the range of the predicate is open, so it falls outside the extrapolated boundary date for the statistics by definition. The Optimizer still is able to estimate an approximate cardinality of 24 million rows for the result, which is nearly an order of magnitude better than the estimate that would be made without extrapolating statistics to the August 8, 2007 date, which is 4 million rows.

64. Because they are all calculated using the same base of collected statistics and average number of rows inserted per day over the interval on which those statistics were collected.
Case 4: Open Range Date Predicate 2

Consider the following request over the open range bounded on the lower end by September 1, 2007 and unbounded on the upper end:

```
SELECT *
FROM ordertbl
WHERE o_orderdate >= DATE '08-04-2007';
```

The lower boundary on this interval is within the extrapolated boundary for the statistics on the `o_orderdate` column, so the Optimizer is able to estimate a cardinality for the result, which is approximately 5 million rows. This result is 6 orders of magnitude better than the estimate that would be made without extrapolating the statistics to the August 8, 2007 date, which is 1 row.

Case 5: Open Range Predicate 3

Consider the following request over the open range bounded by September 1, 2007 on the lower end and unbounded on the upper end:

```
SELECT *
FROM ordertbl
WHERE o_orderdate >= DATE '09-01-2007';
```

Because the lower boundary on this interval is past the extrapolated boundary for the statistics on the `o_orderdate` column, the Optimizer cannot estimate an approximate cardinality for the result, so it projects a result of 0 rows with or without using extrapolation.
Case 6: Open Range Predicate With Wide Final Interval

Consider the following request over the open range bounded by May 30, 2007 on the lower end and unbounded on the upper end:

```
SELECT *
FROM ordertbl
WHERE o_orderdate >= DATE ‘05-30-2007’;
```

Unlike the previous cases, the average number of rows inserted into `ordertbl` is a constant one million per day until May 31, 2007, at which point the number of new rows inserted into the table becomes very sparse. Because the dates for the projected values extend so far into the future, the last interval in the equal-height histogram is extremely wide.

A random AMP sample provides the Optimizer with enough information to detect the change in the rate of row insertion and to compensate accordingly in calculating an extrapolated date for the request, which is June 6, 2007, only 8 days past the date the rate of row insertion changed radically. As a result, the Optimizer uses an approximate extrapolated cardinality of 8 million rows when it develops its query plan, which is 4 times larger than the number that would be estimated without extrapolation.

Case 7: Static Column

Consider the following request specifies an IN list predicate with widely separated values that are specified over the static column `ps_suppkey`:

```
SELECT *
FROM partsupp
WHERE ps_suppkey IN (4, 35000, 40000);
```
Because \( \text{ps_suppkey} \) is a static column, the Optimizer assumes that virtually no new \( \text{ps_suppkey} \) values will be added to the \( \text{partsupp} \) table in the future, so it uses a simple random-AMP sample to determine the extrapolated number of new rows having the values 4, 35000, or 40000 for \( \text{ps_suppkey} \).

Assuming that the number of rows per value is a constant sum across all values of \( \text{partsupp} \), but that all or some of the needed values are not in the range of the values contained in the collected statistics, the system extrapolates a solution using one of three methods:

<table>
<thead>
<tr>
<th>IF the ...</th>
<th>THEN the number of rows per value is taken from ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>value is less than the MinValue in the interval</td>
<td>the first interval in the histogram.</td>
</tr>
<tr>
<td>value is greater than the MaxValue in the interval</td>
<td>the last interval in the histogram.</td>
</tr>
<tr>
<td>first or last intervals are either high-biased or single-value equal-height intervals</td>
<td>the table-level estimated rows per value based on a random AMP sample.</td>
</tr>
</tbody>
</table>

The estimated cardinality is then accumulated for each element of the IN list or OR list predicate to determine the final estimate.

In this particular case, there is no \( \text{ps_suppkey} \) value of 40000, and because \( \text{ps_suppkey} \) is a static column, the Optimizer assumes that the value 40000 is not likely to have been added since statistics were last collected. It uses a random-AMP sample to extrapolate an estimate of 100 additional rows per \( \text{ps_suppkey} \) value based on the existing average rows inserted daily per value.

The final approximate cardinality estimates are \( 1000+100=1100 \) rows for \( \text{ps_suppkey}=4 \) and \( 700+100=800 \) rows for \( \text{ps_suppkey}=35,000 \). Without extrapolation, the Optimizer estimates cardinalities of \( 1000 \) rows for \( \text{ps_suppkey}=4 \) and \( 700 \) rows for \( \text{ps_suppkey}=35,000 \), both of which are taken directly from the existing interval histogram statistics (see “Rolling Columns and Static Columns” on page 258).
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

Note that the unit for all Optimizer cost measures is time. The longer a given request takes to complete, the more costly it is. The column and index demographics gathered by the COLLECT STATISTICS (Optimizer Form) statement (see SQL Data Definition Language for syntax and usage information) and the statistics it computes permit the Optimizer to estimate the cardinalities of relations for single-table and join access and to better identify skew in tables, so it has more finely-tuned knowledge for generating plans using its cost estimator functions.

Example

Consider the following table fragment:

<table>
<thead>
<tr>
<th>state</th>
<th>serial_num</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>12345</td>
</tr>
<tr>
<td>28</td>
<td>23456</td>
</tr>
<tr>
<td>51</td>
<td>12345</td>
</tr>
<tr>
<td>51</td>
<td>23456</td>
</tr>
</tbody>
</table>

Assume the following query results in a full-table scan:

```sql
SELECT *
FROM table_x
WHERE state IN (28, 51)
AND serial_num IN (12345, 23456);
```

Now modify the request as follows:

```sql
SELECT *
FROM table_x
WHERE (state = 28 AND serial_num = 12345)
OR (state = 51 AND serial_num = 23456)
OR (state = 28 AND serial_num = 23456)
OR (state = 51 AND serial_num = 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.

- \((A \text{ OR } B) \text{ AND } (C \text{ OR } D)\)
- \((A \text{ AND } C) \text{ OR } (A \text{ AND } D) \text{ OR } (B \text{ AND } C) \text{ OR } (B \text{ AND } D)\)

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:

\((\text{NUSI} = \text{A OR NUSI} = \text{B}) \text{ AND } (X = 3 \text{ OR } X = 4)\)

In this case, CNF is more performant because the access path consists of two NUSI SELECTs with values of A and B. The condition \((X=3 \text{ OR } X=4)\) 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 \((\text{NUSI}_\text{A}, \text{NUSI}_\text{B})\) comprise a NUSI:

\((\text{NUSI}_\text{A} = 1 \text{ OR } \text{NUSI}_\text{A} = 2) \text{ AND } (\text{NUSI}_\text{B} = 3 \text{ OR } \text{NUSI}_\text{B} = 4)\)

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 column comparison using IN, such as the following.

\(\text{Field IN (Value}_1, \text{Value}_2, \ldots)\)

Internally, that expression is converted to CNF.

\(\text{Field} = \text{Value}_1 \text{ OR Field} = \text{Value}_2 \text{ OR} \ldots\)

Therefore, the same access path is generated for either form.
Example 4

Assume an expression involves a multiple-column comparison using an IN list, such as in the following example:

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

The Optimizer converts this syntax internally 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.

About Predicting Costs for the Optimizer

The purpose of cost prediction is to provide the Optimizer with accurate estimates of the costs to perform various operations in various ways. Type 2 costing is more accurate than the previous generation, Type 1 costing, so the Optimizer has better quality costing data to use in making cost-based decisions.

The cost of an operation is the service time required by subsystems to undertake an operation on an otherwise unloaded system. The Optimizer compares alternative ways of performing an operation based on estimates of CPU, I/O, and BYNET component service times and then chooses the alternative that has the lowest overall cost. There are practical situations where a particular approach should be excluded, or its likelihood greatly reduced, by increasing its predicted cost value, so a heuristic cost component is also included with the collection of various subsystem cost components.

Overall cost, which is a linear combination of the subsystem costs, is the value used by the Optimizer to compare the various alternatives available to it. The cost prediction formulas and methods maintain individual subsystem level values in order to make it possible to determine how much of a given overall cost is accounted for by each of its CPU, I/O, BYNET, and heuristic component values. This information makes it possible to distinguish between I/O-intensive and CPU-intensive approaches, enabling an analyst to have a clear view of error sources when a situation occurs in which a predicted cost value is grossly inaccurate.
You can capture subsystem cost values for individual steps in a query plan using either an SQL INSERT EXPLAIN request or the Teradata Index Wizard to funnel the appropriate information into the Query Capture Database table QuerySteps, in the following column set:

- EstCPUCost
- EstIOCost
- EstNetworkCost
- EstHRCost

See Chapter 6: “Query Capture Facility” and “INSERT EXPLAIN” in SQL Data Manipulation Language for additional information.

If query logging is enabled, the system captures the same information in DBQLStepTbl in the following column set:

- EstCPUCost
- EstIOCost
- EstNetCost
- EstHRCost

See Database Administration and “BEGIN QUERY LOGGING” in SQL Data Definition Language for additional information.

The corresponding overall cost values are captured in the Cost column of the QCD QuerySteps table and in the EstProcTime column of the DBQL DBQLStepTbl tables, respectively.

When predicting costs, there are two input data categories. First are values that characterize the runtime environment, and second are values which characterize the database entities involved in the request being optimized. The runtime environment includes hardware platform, operating system, I/O subsystem and database algorithms, essentially all the product set elements of a Teradata database system. I/O throughput rates, CPU speed, BYNET capacity, and code path lengths all are examples of runtime environment performance factors. In OCES, the runtime environment is characterized by the cost parameter values contained in an active Cost Profile. However, Cost Profiles do not cover database entity characteristics such as the number of rows in a table which satisfy some predicate or the number of unique values occurring in a column. During query planning, the Optimizer estimates database entity characteristics using various kinds of metadata, including statistics, random AMP samples, and a variety of estimates derived from these base data sources.

Cost Validation and Plan Validation are test procedures used to demonstrate that Type 2 costing is superior to Type 1 costing by showing that Type 2 predicted costs are more accurate than Type 1 predicted costs, and that this improved accuracy produces better query plans. Accuracy assessments compare predicted cost values to estimated actual cost values for the individual steps of a query plan.
For example, let $P$ and $A$ represent step-level predicted and estimated actual costs, respectively.

The following equations compute the prediction and normalized prediction errors for cost accuracy, respectively:

\[
\text{Prediction error} = PE = P - A
\]

\[
\text{Normalized prediction error} = NE = \frac{(P - A)}{A}
\]

When $PE > 0$, the true cost has been overestimated, and when $PE < 0$, the true cost has been underestimated.

To compare new and old costing methods, calculate the accuracy error ratio, which is defined as follows:

\[
\text{Accuracy error ratio} = AER = \left| \frac{NE_{\text{new}}}{NE_{\text{old}}} \right|
\]

A newer method is more accurate than an older method when $AER < 1$.

The philosophy of Optimizer cost prediction is always to overestimate, rather than underestimate, operation costs. Because of this philosophy, the desired pattern for both Type 1 and Type 2 predicted costs is to exceed (or be worse than) the actual costs, with Type 2 predicted costs being closer to the actual costs than their Type 1 counterparts.

### Optimizer Cost Functions

The Optimizer has a set of internal functions that return various cost factors used to compute optimal relation 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 required to access a relation or its indexes or both, 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.

To facilitate cost optimization, Teradata Database provides a framework for cost estimation within which cost function parameter values can be administered, updated, and deployed in a uniform and convenient way. The cost estimation framework provides the additional advantage of providing a mechanism for estimating the accuracy of environmental cost predictions compared with actual environmental costs. The comparative information can then be used to better tune parameter values and formula definitions to produce more accurate query plan cost estimates.

The sole purpose of the cost estimation framework information described in the pages that follow is to facilitate understanding of how the system works. Only Teradata field engineers are authorized to make site-specific modifications to the default Cost Profiles.
Terminology

The following terms and their definitions are necessary to fully understand the Cost Profile framework:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost Profile</td>
<td>See “Optimizer Cost Estimation Profiles” on page 272.</td>
</tr>
<tr>
<td>Cost Prediction Parameter</td>
<td>A Cost Profile parameter used by Optimizer cost functions to predict operation costs.</td>
</tr>
<tr>
<td>Cost Profile Activation</td>
<td>The process of initializing a Cost Profile instance given a particular Cost Profile instance specification (see “Cost Profile Activation” on page 289).</td>
</tr>
<tr>
<td>Cost Profile Instance</td>
<td>An encapsulated collection of parameter values that is sufficient for the Optimizer cost functions. Optimizer cost functions refer to such an instance rather than using individual variables or constant values.</td>
</tr>
<tr>
<td>Cost Profile Instance Specification</td>
<td>A dictionary specification of values for a subset of the parameters defined for the type of the instance. These values override those computed by the system.</td>
</tr>
<tr>
<td>Cost Profile Type Specification</td>
<td>A dictionary specification of all the parameters used for a given type of profile. The available types are the following:</td>
</tr>
<tr>
<td></td>
<td>• Type 1, defined as CostProfileTypes_v.ProfileTypeName ='Legacy') in the dictionary.</td>
</tr>
<tr>
<td></td>
<td>• Type 2, defined as CostProfileTypes_v.ProfileTypeName ='Type2') in the dictionary.</td>
</tr>
<tr>
<td>Initialization Parameter</td>
<td>A Cost Profile parameter used during the Cost Profile activation process (see “Cost Profile Activation”) to initialize cost parameter values (see “Cost Prediction Parameter”).</td>
</tr>
</tbody>
</table>

Optimizer Cost Estimation Profiles

An optimizer Cost Profile is a list of named values maintained in such a way that additional values can be added, and existing values can be deleted or changed, without a need to redefine dictionary structures. A profile definition consists of the following general attributes:

- A profile name
- A profile type
- A collection of named constant values
  The collection of constant values varies by type.
A profile is identified by name, and is an instance of its profile type. Instance specifications in the dictionary can provide alternative constant values.

There are two Cost Profile perspectives: specification and use. Cost Profile specifications are stored in several dictionary tables. They consist of two kinds of specifications:

- Type specifications
- Instance specifications

**Components of a Cost Profile**

A Cost Profile is all of the following things:

- A collection of values that characterize platform capabilities, runtime environment characteristics, database software characteristics, and costing strategies.
- A complete specification of all parameter values used by system cost prediction methods. There are two kinds of parameters:
  - Initialization parameters
    The system uses this set of parameters to initialize cost prediction parameter values only. It is not used by cost prediction methods.
  - Cost prediction parameters
    The system uses this set of parameters to predict the costs of various database operations.
- An object that encapsulates all information needed to initialize and operate a set of cost prediction methods.
- Different instances imply different cost values or even different algorithms for determining costs.
- Prediction APIs can use the currently active profile for their calculations.

See [SQL Data Manipulation Language](https://www.sql.com/data-manipulation-language) for documentation of the DIAGNOSTIC SET COSTS and DIAGNOSTIC HELP COSTS statements.

**Cost Profile Type Specifications**

A Cost Profile type specification is a list of parameter specifications, and a Cost Profile instance specification (see “Cost Profile Instance Specifications” on page 277) of that type is a list of parameter values assigned to a subset of those parameters. Two system tables, `DBC.CostProfileTypes` and `DBC.ConstantDefs`, store various components of a Cost Profile type specification. The system uses Cost Profiles to supply parameter values to Optimizer cost prediction calculations, and Cost Profile activation is the term that describes the process of mapping an instance from the dictionary to the internal data structures that make up an active Cost Profile. The Cost Profile activation process merges runtime environment data taken from the platform or from a Target Level Emulation (see Chapter 8: “Target Level Emulation”) cost segment with dictionary data for the particular Cost Profile being activated.
A Cost Profile type definition contains the following components:

- One row in `DBC.CostProfileTypes` to specify a type code and type name
- Multiple rows in `DBC.ConstantDefs` to define all of the parameters that can be used in a Cost Profile instance of that type

There are two Cost Profile types named `Legacy` (ProfileTypeId=1) and `Type2` (ProfileTypeId=2).

The default Cost Profile used by your system is determined by the DBS Control flag CostProfileId (see “Cost Profiles and The DBS Control Utility” on page 290 and Utilities).

The system uses the following dictionary tables to maintain cost estimation profiles:

<table>
<thead>
<tr>
<th>Cost Profile Dictionary Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBC.CostProfiles</td>
<td>Defines the profile instances and their associated types.</td>
</tr>
<tr>
<td>DBC.CostProfileTypes</td>
<td>Defines all the profile types.</td>
</tr>
<tr>
<td>DBC.ConstantDefs</td>
<td>Defines the constants that make up a profile type.</td>
</tr>
<tr>
<td>DBC.ConstantValues</td>
<td>Defines profile-specific overrides to specific constants in a profile type.</td>
</tr>
</tbody>
</table>

To interrogate the dictionary for the parameters that are defined for Type 1 and Type 2 Cost Profiles, respectively, you can issue either or both of the following SELECT requests:

```sql
SELECT constid, constcat, constname
FROM DBC.ConstantDefs
WHERE ProfileTypeId = 1
ORDER BY 1;
```

```sql
SELECT constid, constcat, constname
FROM DBC.ConstantDefs
WHERE ProfileTypeId = 2
ORDER BY 1;
```

The following example indicates typical default results for a Type 2 Cost Profile. The report for a Type 1 Cost Profile is similar.

```sql
SELECT constid, constcat, constname
FROM DBC.ConstantDefs
WHERE ProfileTypeId = 2
ORDER BY 1;
```

*** Query completed. 685 rows found. 3 columns returned.
*** Total elapsed time was 1 second.
Two important things to notice about this example are the following points:

- Each parameter has a unique name and ID value.
- Parameters are classified into two categories:

<table>
<thead>
<tr>
<th>Parameter Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Initialization parameter</td>
</tr>
<tr>
<td>C</td>
<td>Cost parameter</td>
</tr>
</tbody>
</table>

In some system contexts, parameters are accessed by their ID value, while in other contexts they are accessed by their names.

You can use the Cost Profile modification macros to define new Cost Profile instance specifications, but you cannot directly manipulate the underlying Cost Profile tables `DBC.CostProfiles` and `DBC.ConstantValues`. Similarly, you cannot change column values in the `DBC.CostProfileTypes` and `DBC.ConstantDefs` tables.

The Cost Profile specifications supplied by Teradata are all members of the Fixed category, defined as `DBC.CostProfiles_v.ProfileCat = 'F'`, while Cost Profile specifications created using the modification macros (see SystemFE Macros) are members of the Variable category, defined as `DBC.CostProfiles_v.ProfileCat = 'V'`. You cannot modify Fixed category specifications.
The system supports the following two cost estimation profile types:

- **Type 1**
  The Type 1 profile invokes cost prediction based on pre-Teradata Database 12.0 cost prediction formulas and methods.
  Its profile ID value is 1 and its name is *Legacy*.

- **Type 2**
  The Type 2 profile invokes cost prediction based on cost prediction formulas and methods that are standard for Teradata Database 12.0.
  Its profile ID value is 2 and its name is *Type2*.
  The Type 2 Cost Profile is the default.

---

**Cost Profile Dictionary Table**

<table>
<thead>
<tr>
<th><strong>Cost Profile Dictionary Table</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>DBC.CostProfiles</td>
<td>Defines the profile instances and their associated types.</td>
</tr>
<tr>
<td></td>
<td><strong>FOR this profile type ...</strong> <strong>THE value of ProfileTypeld is ...</strong></td>
</tr>
<tr>
<td></td>
<td>Legacy 1</td>
</tr>
<tr>
<td></td>
<td>Type2 2</td>
</tr>
<tr>
<td>DBC.CostProfileTypes</td>
<td>Defines all the profile types.</td>
</tr>
<tr>
<td></td>
<td><strong>FOR this profile type ...</strong> <strong>THE value of ProfileTypeName is ...</strong></td>
</tr>
<tr>
<td></td>
<td>Legacy 'Legacy'</td>
</tr>
<tr>
<td></td>
<td>Type2 'Type2'</td>
</tr>
<tr>
<td>DBC(ConstantDefs)</td>
<td>Defines the cost profile parameters that make up a cost profile type. Each parameter definition is a row in this table. Two categories exist:</td>
</tr>
<tr>
<td></td>
<td>• Type 1 legacy definitions. I/O, BYNET, and platform configuration parameters carried forward from previous constant definitions. These have the same names, semantics, and initialization logic as their Type 1 predecessors, but different ConstId values.</td>
</tr>
<tr>
<td></td>
<td>• Type 2 definitions that support Type 2 cost formulas. Newer parameters than those defined for Type 1 formulas.</td>
</tr>
<tr>
<td>DBC.ConstantValues</td>
<td>Defines profile-specific overrides to specific profile parameters in a profile type.</td>
</tr>
</tbody>
</table>

*See Data Dictionary for further information about these system tables.*
The following table defines the Type 2 Cost Profiles:

<table>
<thead>
<tr>
<th>Profile Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teradata12</td>
<td>Standard cost profile for 32-bit platforms.</td>
</tr>
<tr>
<td>T2_Linux64</td>
<td>Standard cost profile for 64-bit Linux platforms.</td>
</tr>
<tr>
<td>T2_Win64</td>
<td>Standard cost profile for 64-bit Windows platforms.</td>
</tr>
<tr>
<td>V2R6</td>
<td>Type 1 cost profile that costs query plans using pre-Teradata Database 12.0 formulas and methods for all platforms.</td>
</tr>
</tbody>
</table>

To enable Target Level Emulation for Type 2 profiles, you must set the following DBS Control flags as indicated:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnableCostProfileTLE</td>
<td>TRUE</td>
<td>See “Procedure To Enable Optimizer Cost Estimation Subsystem DIAGNOSTIC Statements With Target Level Emulation” on page 684 for details.</td>
</tr>
<tr>
<td>EnableSetCostProfile</td>
<td>a value other than 0.</td>
<td>See “Cost Profile Activation” on page 289 for details.</td>
</tr>
</tbody>
</table>

**Cost Profile Instance Specifications**

The two dictionary tables that store Cost Profile instance specifications are `DBC.CostProfiles` and `DBC.ConstantValues`.

An instance specification consists of one row in `DBC.CostProfiles` to define the name and type of a Cost Profile, and zero or more rows in `DBC.ConstantVals` to provide values for a subset of the Cost Profile parameters specified by the Cost Profile type specification. There are several Cost Profile instances of each type.

**Cost Profile Flags**

Many cost profile flags can also be set using internal DBS Control record flags. Whenever the Cost Profile and DBS Control record settings disagree, the system uses the value in the Cost Profile and ignores the setting for the comparable DBS Control flag.

You cannot change the settings for these flags yourself. If you think the settings should be changed for your site, contact your Teradata support representative for a consultation. Only a Teradata support representative can change the settings of internal Cost Profile flags.

For an isolated query regression, changing the default value to the Type 1 values for some or all of these flags might provide a useful workaround.

The two following tables list and describe the various Cost Profile flags related to cardinality estimation and join costing.
The following Cost Profile flags affect cardinality estimation:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Default Type 2 Setting</th>
<th>Default Type 1 Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>AggrEstCapPct</td>
<td>Provides a capping percentage for aggregate cardinality estimates when there are no statistics on the grouping columns. Valid range: 1 - 100</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>AssumedPILUniquePct</td>
<td>Provides the assumed uniqueness for a NUPI when no statistics are available. Used only when RevertEstimationLogic=0. Valid range: 50 - 100</td>
<td>75.0</td>
<td>75.0</td>
</tr>
<tr>
<td>CCFIndependentValueRatio</td>
<td>Sets the value mapping ratio used to detect independent columns. Used only when RevertEstimationLogic=0. Valid range: 0 - 1</td>
<td>0.6</td>
<td>not applicable</td>
</tr>
<tr>
<td>CCFMulti2SnglValueRatio</td>
<td>Sets the ratio between combined values and the multiplication of individual column values that is used to detect independent columns. Valid range: 0 - 1</td>
<td>0.9</td>
<td>not applicable</td>
</tr>
<tr>
<td>EnableSessLevelDrvStats</td>
<td>Enables or disables session-level derived statistics for global temporary and volatile tables. Used only when RevertEstimationLogic=0.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable session-level derived statistics for global temporary and volatile tables.</td>
</tr>
<tr>
<td>1</td>
<td>Enable session-level derived statistics for global temporary and volatile tables.</td>
</tr>
</tbody>
</table>
## EstWithSingleRowOpt

If a table has a single row access using a UPI or USI, determines whether the Optimizer can retrieve the row to derive more accurate cardinality estimates.

Used only when RevertEstimationLogic=0.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable the optimization.</td>
</tr>
<tr>
<td>1</td>
<td>Enable the optimization.</td>
</tr>
</tbody>
</table>

## ExtrapolateStatistics

Enables or disables the extrapolation of statistics when state statistics are detected.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable extrapolation.</td>
</tr>
<tr>
<td>1</td>
<td>Enable extrapolation.</td>
</tr>
</tbody>
</table>

## GroupAMPPISampling

Determines the number of AMPs to be sampled by a COLLECT STATISTICS (Optimizer Form) request to obtain cardinality estimates for a column set.

Used only when RevertEstimationLogic=0.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Single-AMP sampling.</td>
</tr>
<tr>
<td>1</td>
<td>Not used.</td>
</tr>
<tr>
<td>2</td>
<td>Two-AMP sampling.</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Five-AMP sampling.</td>
</tr>
<tr>
<td>6</td>
<td>Node-level sampling.</td>
</tr>
<tr>
<td>7</td>
<td>All-AMP sampling.</td>
</tr>
</tbody>
</table>
### Chapter 2: Query Rewrite and Optimization

#### Cost Optimization

**JoinCardSkewAdj**
- Enables or disables join cardinality estimate adjustments based on skew.
- **Setting** | **Description**
  - 0: Disable join cardinality skew adjustment.
  - 1: Enable join cardinality skew adjustment.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable join cardinality skew adjustment.</td>
</tr>
<tr>
<td>1</td>
<td>Enable join cardinality skew adjustment.</td>
</tr>
</tbody>
</table>

**LegacyJoinCardDefault**
- Enables or disables the legacy join cardinality estimate default, which is used in the absence of statistics or for complex join predicates.
- **Setting** | **Description**
  - 0: Disable the legacy join cardinality estimation default.
  - 1: Enable the legacy join cardinality estimation default.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable the legacy join cardinality estimation default.</td>
</tr>
<tr>
<td>1</td>
<td>Enable the legacy join cardinality estimation default.</td>
</tr>
</tbody>
</table>

**LIKEEstPartialIntRatio**
- Determines the default ratio for calculating the number of qualified rows from partially-covered intervals for LIKE predicates with an "abc%"- pattern.
- **Valid range**: 0 - 1

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Default Type 2 Setting</th>
<th>Default Type 1 Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>JoinCardSkewAdj</td>
<td>Enables or disables join cardinality estimate adjustments based on skew.</td>
<td>1</td>
<td>not applicable</td>
</tr>
<tr>
<td>LegacyJoinCardDefault</td>
<td>Enables or disables the legacy join cardinality estimate default, which is used</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>LIKEEstPartialIntRatio</td>
<td>Determines the default ratio for calculating the number of qualified rows from</td>
<td>0.125</td>
<td>0.125</td>
</tr>
<tr>
<td></td>
<td>partially-covered intervals for LIKE predicates with an &quot;abc%&quot;- pattern.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Flag** | **Description** | **Default Type 2 Setting** | **Default Type 1 Setting** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JoinCardSkewAdj</td>
<td>Enables or disables join cardinality estimate adjustments based on skew.</td>
<td>1</td>
<td>not applicable</td>
</tr>
<tr>
<td>LegacyJoinCardDefault</td>
<td>Enables or disables the legacy join cardinality estimate default, which is used</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>LIKEEstPartialIntRatio</td>
<td>Determines the default ratio for calculating the number of qualified rows from partially-covered intervals for LIKE predicates with an &quot;abc%&quot;- pattern.</td>
<td>0.125</td>
<td>0.125</td>
</tr>
</tbody>
</table>
### RASHistAbsDeviation

Determines the absolute number of rows difference between a random AMP sample cardinality estimate and histogram row counts.

If the difference exceeds this value, the Optimizer assumes that the histogram statistics are stale and overrides them with the estimates returned by a random AMP sample.

Even if the difference exceeds the value for the absolute deviation, the Optimizer still uses histogram statistics for its selectivity estimates, and those estimates are extrapolated.

A setting of 0 disables the check.

Valid range: \( \geq 0 \)

Used only when both of the following flag values are set like this:

- `RevertEstimationLogic=0`
- `ExtrapolateStatistics=1`

<table>
<thead>
<tr>
<th>Default Type 2 Setting</th>
<th>Default Type 1 Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000.0</td>
<td>10000.0</td>
</tr>
</tbody>
</table>

### RASHistDeviationPct

Determines the allowable percentage deviation between random AMP sample and histogram cardinality estimates.

If the deviation exceeds this value, the Optimizer assumes that the histogram statistics are stale and overrides them with the estimates returned by a random AMP sample.

A setting of 0 disables the check.

Valid range: 0 - 100

Used only when both of the following flag values are set like this:

- `RevertEstimationLogic=0`
- `ExtrapolateStatistics=1`

<table>
<thead>
<tr>
<th>Setting</th>
<th>Type 1 Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>not applicable</td>
</tr>
</tbody>
</table>
### Cost Optimization

The following Cost Profile flags affect join costing:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Default Type 2 Setting</th>
<th>Default Type 1 Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>RevertEstimationLogic</td>
<td>Enables or disables the current cardinality and selectivity estimation-related logic, including the use of derived statistics and single-table unique index access cardinality estimation.</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><strong>Setting</strong></td>
<td><strong>Description</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Use current cardinality and selectivity estimation logic.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Use pre-12.0 cardinality and selectivity estimation logic.</td>
<td></td>
</tr>
<tr>
<td>SoftUniquePct</td>
<td>If the uniqueness of a column is more than this percentage, it is assumed to be a rolling column (see “Rolling Columns and Static Columns” on page 258) and is qualified for distinct value or row extrapolation if its statistics are determined to be stale. A setting of 0 disables extrapolation. Valid range: 0 - 100 Used only when both of the following flag values are set like this: • RevertEstimationLogic=0 • ExtrapolateStatistics=1</td>
<td>95.0</td>
<td>95.0</td>
</tr>
<tr>
<td>ApplySpoolLimit</td>
<td>Enables or disables adding heuristic cost values for predicted excessive spool sizes.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><strong>Code</strong></td>
<td><strong>Description</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Disable spool space limit heuristics.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Enable spool space limit heuristics.</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 2: Query Rewrite and Optimization

#### Cost Optimization

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Enables condition-expression costing control level formulas.</td>
</tr>
<tr>
<td></td>
<td>Enables the ordering of terms based on cost.</td>
</tr>
<tr>
<td>2</td>
<td>Enables condition-expression costing control level formulas.</td>
</tr>
<tr>
<td></td>
<td>Disables the ordering of terms based on cost.</td>
</tr>
<tr>
<td>3</td>
<td>Disables condition-expression costing control level formulas.</td>
</tr>
<tr>
<td></td>
<td>Disables the ordering of terms based on cost.</td>
</tr>
</tbody>
</table>

### Table: CacheableBlocks

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Default Type 2 Setting</th>
<th>Default Type 1 Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheableBlocks</td>
<td>Specifies the maximum number of blocks that can be assumed to be in cache when the same set of blocks is reread during a join operation. Valid range: ≥ 0</td>
<td>10</td>
<td>not available</td>
</tr>
</tbody>
</table>

### Table: ConditionCosting

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Enables condition-expression costing control level formulas.</td>
</tr>
<tr>
<td></td>
<td>Enables the ordering of terms based on cost.</td>
</tr>
<tr>
<td>2</td>
<td>Enables condition-expression costing control level formulas.</td>
</tr>
<tr>
<td></td>
<td>Disables the ordering of terms based on cost.</td>
</tr>
<tr>
<td>3</td>
<td>Disables condition-expression costing control level formulas.</td>
</tr>
<tr>
<td></td>
<td>Disables the ordering of terms based on cost.</td>
</tr>
</tbody>
</table>

### Table: PreJCost

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable prejoin costing.</td>
</tr>
<tr>
<td>1</td>
<td>Enable prejoin costing for unique ID join planning paths.</td>
</tr>
<tr>
<td></td>
<td>Enable prejoin costing for nonunique ID join planning paths.</td>
</tr>
<tr>
<td>2</td>
<td>Enable prejoin costing for unique ID join planning paths.</td>
</tr>
<tr>
<td></td>
<td>Disable prejoin costing for nonunique ID join planning paths.</td>
</tr>
</tbody>
</table>

When enabled, cost estimates are made for any additional steps required on the input relations prior to the join such as sort/duplicate elimination, aggregation, and so on.
For an isolated query regression, changing the default value to the Type 1 values for some or all of these flags might provide a useful workaround.

The following Cost Profile flags affect aspects of request processing other than cardinality estimation and join costing:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Default Type 2 Setting</th>
<th>Default Type 1 Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrScanAxPath</td>
<td>Enables or disables a constraint scan access path. Used only when RevertEstimationLogic=0.</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Enable constraint scan access path for the following:</td>
</tr>
<tr>
<td></td>
<td>• Aggregation</td>
</tr>
<tr>
<td></td>
<td>• Single retrieve and join requests on base tables and join indexes</td>
</tr>
<tr>
<td></td>
<td>• Integration of LikeScan into constraint scans</td>
</tr>
<tr>
<td></td>
<td>• Integration of RangeScan into constraint scans for nonvalue-ordered column secondary indexes</td>
</tr>
<tr>
<td></td>
<td>• Bit map scans on indexes used for constraint scans.</td>
</tr>
<tr>
<td>1</td>
<td>Disable constraint scan access path.</td>
</tr>
</tbody>
</table>

| IndexFillFactor       | Sets the index block fill factor. The assumed index block size is this percentage of the maximum index block size. Valid range: 0.01 - 1.00 | 0.75 | not applicable |

| MaxJoinTables         | Determines the maximum number of tables that can be joined per query block. When a cost profile is activated, Teradata Database adjusts its value to ensure that it is less than or equal to the limit specified by the MaxJoinTables flag in the DBS Control record. MaxJoinTables is usually set for a limit of 128, but it could be as low as 64. Valid range: 0, 16 - 128 Default: 0 (A value of 0 implies using the system default upper limit) | 0 | not applicable |

<table>
<thead>
<tr>
<th>IF MaxJoinTables is set to this value ...</th>
<th>THEN the system sets the value to this value ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the DBS Control setting for MaxJoinTables.</td>
</tr>
<tr>
<td>$k$</td>
<td>$\text{MIN}(\text{MAX}(16,k),m)$.</td>
</tr>
<tr>
<td>Flag</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MaxJoinTables</td>
<td>where:</td>
</tr>
<tr>
<td>(continued)</td>
<td><strong>This variable</strong> …  <strong>Represents</strong> …</td>
</tr>
<tr>
<td>$k$</td>
<td>any valid setting for the MaxJoinTables variable cost profile.</td>
</tr>
<tr>
<td>$m$</td>
<td>the setting for the MaxJoinTables DBS Control flag.</td>
</tr>
<tr>
<td>MaxStatsInterval</td>
<td>Determines the maximum number of intervals allowed for a statistical histogram. Valid range: 1 - 200 Default: 200</td>
</tr>
<tr>
<td>ocsDebug</td>
<td>Determines the action to be taken if Optimizer sanity checks of demographic or costing information fail.</td>
</tr>
<tr>
<td>RevertJoinPlanning</td>
<td>Determines whether the Optimizer uses new join planning logic (FALSE) or older join planning logic (TRUE).</td>
</tr>
<tr>
<td>SpoolFillFactor</td>
<td>Sets the spool block fill factor. The assumed spool block size is this percentage of the maximum spool block size. Valid range: 0.01 - 1.00</td>
</tr>
<tr>
<td>SpoolLimitPercent</td>
<td>Determines the percentage of user-allocated spool space that is considered to be problematic. Values must be expressed in units of 1 percent.</td>
</tr>
<tr>
<td>TableFillFactor</td>
<td>Sets the table block fill factor. The assumed table block size is this percentage of the maximum table block size. Valid range: 0.01 - 1.00</td>
</tr>
</tbody>
</table>
Administering Cost Profiles

Cost Profiles are not meant to be customized for different classes of users because they reflect platform and database characteristics that are independent of any particular user or workload category. Under normal circumstances, the same Cost Profile is used for all system processing. The cost parameters obtained from the Cost Profile are an intrinsic component of Optimizer cost prediction algorithms, and these algorithms do not change for different usage classes.

To mitigate performance risks, it is possible for Cost Profile settings to revert selected new features, or even to fall back to Type 1 costing. Facilities are available to enable selective reversion. The facilities provide the capability to assign a specific Cost Profile to a user and collections of optimization control parameters defined for Type 1 and Type 2 Cost Profiles.

You can assign a Cost Profile to an individual user using the COST PROFILE option in a CREATE PROFILE or MODIFY PROFILE SQL request (see “CREATE PROFILE” and “MODIFY PROFILE” in SQL Data Definition Language for details). If you do this, the system assigns the specified Cost Profile to all sessions for that user.

Using this mechanism, it is possible to assign a variable Cost Profile defined using the variable Cost Profile creation macros to individual users and their sessions.

If necessary, you can assign the fixed Cost Profile named V2R6 to a user in order to revert to Teradata V2R6 cost prediction logic.

---

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Default Type 2 Setting</th>
<th>Default Type 1 Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>UseActualBlockSize</td>
<td>Determines whether to use the declared data block size for I/O costing.</td>
<td>1</td>
<td>not applicable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Do not use the declared data block size for I/O costing.</td>
</tr>
<tr>
<td>1</td>
<td>Use the declared data block size for I/O costing.</td>
</tr>
</tbody>
</table>

---

65. The most likely exception being the LIKEstPartialIntRatio flag (see “Using Histograms for Cardinality Estimation” on page 236).
Browsing and Modifying Cost Profiles

Teradata supplies several views for browsing Cost Profiles as well as several macros for modifying Cost Profile definitions. Access to these tools is restricted.

Teradata provides the following views for browsing Cost Profiles:

<table>
<thead>
<tr>
<th>View Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBC.CostProfiles_v</td>
<td>Reports cost profile instances defined for your system.</td>
</tr>
<tr>
<td>DBC.CostProfileTypes_v</td>
<td>Reports cost profile types defined for your system.</td>
</tr>
<tr>
<td>DBC.CostProfileValues_v</td>
<td>Reports all cost profile instance parameter values defined for your system.</td>
</tr>
</tbody>
</table>

Assuming that you have all the privileges required to access the appropriate dictionary tables, you can then use ordinary SQL SELECT statements to gather profile information through these views.

For example:

- The following request reports a summary view of all Cost Profiles in the data dictionary:
  
  ```sql
  SELECT *
  FROM DBC.CostProfiles_v;
  ```

- The following request reports all the constant values defined for any profile:
  
  ```sql
  SELECT *
  FROM DBC.CostProfileValues_v;
  ```

- The following request reports the types of profiles in the data dictionary:
  
  ```sql
  SELECT *
  FROM DBC.CostProfileTypes_v;
  ```

Of course, you can customize your Cost Profile queries to whatever level of refinement you require.

For example, suppose you want to produce a report of all the profiles defined for your site. You might code a query like the following to perform that task:

```sql
SELECT ProfileName (FORMAT 'X(16)', TITLE 'Name'),
       ProfileId (FORMAT '----9', TITLE 'Id'),
       ProfileCat (FORMAT 'X', Title 'Cat'),
       ProfileDesc (FORMAT 'X(64)', TITLE 'Description')
FROM DBC.CostProfiles_v
ORDER BY ProfileId;
```

The report produced by this request might look something like the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Id</th>
<th>Cat</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SysDefault</td>
<td>0</td>
<td>F</td>
<td>SysDefault DBS cost values only. Partial profil</td>
</tr>
<tr>
<td>V2R4</td>
<td>1</td>
<td>F</td>
<td>V2R4 DBS cost values only. Partial profile, no</td>
</tr>
<tr>
<td>V2R5</td>
<td>2</td>
<td>F</td>
<td>V2R5 DBS cost values only. Partial profile, no</td>
</tr>
<tr>
<td>V2R5_Solaris</td>
<td>3</td>
<td>F</td>
<td>V2R5 DBS cost values only. Specific to Solaris</td>
</tr>
<tr>
<td>V2R5_Array</td>
<td>19</td>
<td>F</td>
<td>Disk array cost values for V2R5 Default. Partial</td>
</tr>
<tr>
<td>V2R5_6283</td>
<td>10001</td>
<td>V</td>
<td>V2R5 CPU Costs with LSI_6283 disk array</td>
</tr>
</tbody>
</table>
Suppose you want to produce a report on the details of a single profile. You might code a query like the following to perform the task:

```sql
SELECT ConstName (TITLE 'Name'),
       ConstId (TITLE 'Id', FORMAT '--9'),
       ConstVal (TITLE 'Value', FORMAT 'ZZZZZ9.99999'),
       ConstCat (TITLE 'Cat')
FROM DBC.CostProfileValues_v
WHERE ProfileName = 'V2R5_6283'
ORDER BY ConstCat DESC, ConstName;
```

The report produced by this request might look something like the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Id</th>
<th>Value</th>
<th>Cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayRead128k</td>
<td>22</td>
<td>325.00000</td>
<td>I</td>
</tr>
<tr>
<td>ArrayRead16k</td>
<td>19</td>
<td>920.00000</td>
<td>I</td>
</tr>
<tr>
<td>ArrayRead32k</td>
<td>20</td>
<td>775.00000</td>
<td>I</td>
</tr>
<tr>
<td>ArrayRead4k</td>
<td>17</td>
<td>1,120.0000</td>
<td>I</td>
</tr>
<tr>
<td>OptRowAccessCost</td>
<td>41</td>
<td>0.00802</td>
<td>C</td>
</tr>
</tbody>
</table>

Teradata Database also provides a set of macros to list Cost Profile information using SQL LIKE operator criteria as follows:

- `DBC.ListCostProfiles`
- `DBC.ListCostProfiles1`
- `DBC.ListCostProfiles2`

See Data Dictionary for more information about these Cost Profile macros.

Teradata provides the following macros for creating variable Cost Profiles:

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBC.CreateNewCostProfile</td>
<td>Creates a new profile of a given type.</td>
</tr>
<tr>
<td>DBC.CopyCostProfile</td>
<td>Copies an existing profile definition to a new name. The copied definition</td>
</tr>
<tr>
<td></td>
<td>can then be modified using the <code>DBC.InsertConstantValue</code> macro.</td>
</tr>
<tr>
<td>DBC.InsertConstantValue</td>
<td>Inserts new constant values into a profile.</td>
</tr>
<tr>
<td>DBC.DeleteCostProfile</td>
<td>Deletes the definition for a profile instance.</td>
</tr>
<tr>
<td>DBC.MergeCostProfile</td>
<td>Merges the constant values in the source Cost Profile to the target Cost</td>
</tr>
<tr>
<td></td>
<td>Profile. This overwrites the corresponding target Cost Profile values.</td>
</tr>
</tbody>
</table>

These macros are the only way you can modify the `DBC.CostProfiles` and `DBC.ConstantValues` tables.

See Data Dictionary for more information about these views and macros.
Cost Profile Activation

The DBS Control CostProfileId flag specifies the standard default Cost Profile. If you change this value, then it applies to all new sessions. Dynamic cost profile activation is also possible. To dynamically activate an activate an alternative Cost Profile, issue an appropriate DIAGNOSTIC SET PROFILE request using the following general syntax (see SQL Data Manipulation Language for syntax details):

```
DIAGNOSTIC SET PROFILE profile_name ON FOR scope_level
```

DIAGNOSTIC SET PROFILE requests activate alternative Cost Profiles, which the system then uses for the entire SYSTEM or for subsequent requests in the current SESSION, depending on the scope level you specify.

You can control the availability of DIAGNOSTIC SET PROFILE requests by means of the EnableSetCostProfile flag of the DBS Control record. This flag controls usage of DIAGNOSTIC SET PROFILE statements to dynamically change Cost Profiles used for query planning (see SQL Data Manipulation Language for more information about the DIAGNOSTIC SET PROFILE statement).

The following table lists the valid EnableSetCostProfile values and their definitions:

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The system default Cost Profile cannot be changed. The profile can be either Type 1 or Type 2. The EnableCostProfileTLE DBS Control flag is set FALSE. The DIAGNOSTIC SET PROFILE statement is disabled. The system aborts attempts to submit DIAGNOSTIC SET PROFILE requests and returns an error to the requestor.</td>
</tr>
<tr>
<td>1</td>
<td>A Type 1 or Type 2 nondefault Cost Profile can be activated at the SESSION and REQUEST levels only.</td>
</tr>
<tr>
<td>2</td>
<td>A Type 1 or Type 2 nondefault Cost Profile can be activated at all levels.</td>
</tr>
</tbody>
</table>

The default value for EnableSetCostProfile is 0.

See the chapter on the DBS Control utility in Utilities for details.

Cost Profile DIAGNOSTIC Statements

Several DIAGNOSTIC statements can be used to display which Cost Profiles are active or to list the parameter values for a particular active Cost Profile. There are also several DIAGNOSTIC statements that can be used to activate a nondefault Cost Profile, but only Teradata field engineers have the access privileges required to use them.

See SQL Data Definition Language for information about the SQL Cost Profile DIAGNOSTIC statements.
Cost Profiles and The DBS Control Utility

There are three DBS Control Record flags (see Utilities) that are particularly relevant to Cost Profile selection and use. The following table summarizes these flags:

<table>
<thead>
<tr>
<th>DBS Control Record Flag Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CostProfileId</td>
<td>Controls the Cost Profile identifier value the system uses to select the standard Cost Profile.</td>
</tr>
<tr>
<td>EnableCostProfileTLE</td>
<td>Controls availability of the DIAGNOSTIC request to activate both a TLE cost segment and a Cost Profile at the same time.</td>
</tr>
<tr>
<td>EnableSetCostProfile</td>
<td>Controls availability of the DIAGNOSTIC request to change which Cost Profile is selected.</td>
</tr>
</tbody>
</table>

When a session starts, the system assigns a Cost Profile to it. The assigned Cost Profile is the one used by the Optimizer for optimizing queries either for the duration of a session or until it is overridden.

The term *standard default cost profile* refers to the Cost Profile the system assigns to a new session, and the CostProfileId flag in the DBS Control Record contains the standard default Cost Profile identifier.

There are four candidate standard default fixed Cost Profiles. Their names and identifiers are listed in the following table:

<table>
<thead>
<tr>
<th>Cost Profile Name</th>
<th>Cost Profile ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teradata12</td>
<td>36</td>
</tr>
<tr>
<td>T2_Linux64</td>
<td>21</td>
</tr>
<tr>
<td>T2_Win64</td>
<td>37</td>
</tr>
<tr>
<td>V2R6</td>
<td>35</td>
</tr>
</tbody>
</table>

The DIPOCES script sets up these and several other Cost Profiles. The first three of the preceding four profiles are intended for normal use, while any others are for special purpose uses.

You can use the following SELECT request to produce a report of the available Cost Profile names and their identifier values:

```
SELECT ProfileId, ProfileName
FROM DBC.CostProfiles_v
ORDER BY 1;
```

When a new session starts, the system uses the value in the DBS Control record CostProfileId flag to select the Cost Profile for that session. The problem then becomes determining the method to be used to specify the desired value for that flag.
There are three possible ways to modify the value of CostProfileId:

- Automatic selection by the system.
- Explicit selection by a DBA.
- Dynamic selection by a programmer.

These approaches to the problem are summarized in the following table:

<table>
<thead>
<tr>
<th>IF you use this approach to selecting a CostProfileId value ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic selection</td>
<td>the system automatically determines its platform type and selects the appropriate standard default Cost Profile for that platform.</td>
</tr>
<tr>
<td>Explicit selection</td>
<td>a DBA uses the DBS Control utility to set CostProfileId to the desired value.</td>
</tr>
<tr>
<td>Dynamic selection</td>
<td>someone with appropriate privileges submits a DIAGNOSTIC SET PROFILE request to activate a specific Cost Profile at a particular scope level.</td>
</tr>
</tbody>
</table>

The following table provides high-level procedures for initializing a system to use either the standard default Type 1 Cost Profile or to select a different profile based on its platform type:

<table>
<thead>
<tr>
<th>TO configure a system ...</th>
<th>Follow this procedure ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>to use the default standard Type 1 Cost Profile</td>
<td>1  Set CostProfileId == 35</td>
</tr>
<tr>
<td></td>
<td>2  You have two choices for the immediacy of instantiating the new Cost Profile:</td>
</tr>
<tr>
<td></td>
<td>IF you want to configure the system so that ...</td>
</tr>
<tr>
<td></td>
<td>cost profiles change to the standard default Cost Profile only when users next log on to a new session</td>
</tr>
<tr>
<td></td>
<td>the Cost Profile for all running sessions is set to the standard default Cost Profile at the same time</td>
</tr>
<tr>
<td></td>
<td>3  End of procedure.</td>
</tr>
</tbody>
</table>
### TO configure a system ...

to select a standard Cost Profile based on its platform type

<table>
<thead>
<tr>
<th>Platform</th>
<th>Default Cost Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit Linux</td>
<td>T2_Linux64</td>
</tr>
<tr>
<td>64-bit Windows</td>
<td>T2_Win64</td>
</tr>
<tr>
<td>anything else</td>
<td>Teradata12</td>
</tr>
</tbody>
</table>

CostProfileID is set to the ID value for the selected Cost Profile.

### Follow this procedure ...

1. Set CostProfileId == 0
2. You have two choices:
   - Do nothing.
     The default Cost Profile is then used only after the user logs off the current session and later logs onto a new session.
   - Reset the database.
     This changes the default Cost Profile for all users/sessions immediately.

See the embedded tables for “to use the default standard Type 1 Cost Profile” on page 291 for details.

3. End of procedure.

When the procedure has been completed, the system uses the following Cost Profile, depending on the system type:

### explicitly using DBS Control

1. Set CostProfileId == value
   where value is the ID number of the default Cost Profile you want to configure.
   For example, if you set the value for CostProfileId to 36, then the default Cost Profile becomes Teradata12.
2. You have two choices:
   - Do nothing.
     The default Cost Profile is then used only after the user logs off the current session and later logs onto a new session.
   - Reset the database.
     This changes the default Cost Profile for all users/sessions immediately.

See the embedded tables for “to use the default standard Type 1 Cost Profile” on page 291 for details.

3. End of procedure.
## 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 it is likely, but not certain, that an accurate cardinality estimate is available to the Optimizer. If statistics are stale, then the derived statistics framework should minimize any problems with them.

- The Optimizer uses primary index statistics to estimate the cardinality of 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 percent of the total number of rows are added or deleted.

When no statistics are available, or when the table has no primary index, 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 178).

For the following reasons, this might not be an accurate estimate of the cardinality of the table:

- 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.

<table>
<thead>
<tr>
<th>TO configure a system ...</th>
<th>Follow this procedure ...</th>
</tr>
</thead>
</table>
| dynamically using a DIAGNOSTIC SET PROFILE request | 1 Set EnableSetCostProfile to either of the following values:  
* 1  
* 2  

See “Cost Profile Activation” on page 289 for the definitions of these flags. |

2 Submit a DIAGNOSTIC SET PROFILE request that specifies the name of the desired Cost Profile and a scope level (see SQL Data Manipulation Language for the syntax and other details of the DIAGNOSTIC SET PROFILE statement).  

If you specify a scope level of SYSTEM, then the specified Cost Profile becomes the standard default Cost Profile and the system saves its identifier in the CostProfileId DBS Control flag.  

This new setting (but not those set with scope levels of IFP, SESSION, or REQUEST) is persistent, and is retained across database resets and SysInit operations (see Utilities). |

3 End of procedure. |
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.

Stale 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 constraint in the WHERE clause of a query on a value from an indexed column.

If multiple 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 request 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:

- For NoPI tables, column access is done using secondary or join indexes. If no secondary or join indexes are defined for the column in question, then access is always done using a full-table scan.
- 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 columns 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.column_1 = 1 AND table_1.column_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 Database 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 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.

<table>
<thead>
<tr>
<th>Cost Factor Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>External cost parameters</td>
<td>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.</td>
</tr>
<tr>
<td>Performance constants</td>
<td>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.</td>
</tr>
</tbody>
</table>
## External Cost Parameters

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

<table>
<thead>
<tr>
<th>External Cost Parameter Group</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizer weights and scales</td>
<td>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.</td>
</tr>
<tr>
<td>CPU cost variables</td>
<td>CPU costs for accessing, sorting, or building rows using various methods.</td>
</tr>
<tr>
<td>Disk delay definitions</td>
<td>Elapsed times for various disk I/O operations. Measured in milliseconds.</td>
</tr>
<tr>
<td>Disk array throughput</td>
<td>Throughputs for disk array I/O operations. Measured in I/Os per second.</td>
</tr>
<tr>
<td>Network cost variables</td>
<td>Message distribution and duplication costs and overheads. Measured in milliseconds.</td>
</tr>
<tr>
<td>Optimizer environment variables</td>
<td>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.</td>
</tr>
<tr>
<td>Miscellaneous cost parameters</td>
<td>DBS Control 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.</td>
</tr>
<tr>
<td>Performance constants</td>
<td>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.</td>
</tr>
<tr>
<td>PDE system control files</td>
<td>Lists various system control files used by PDE. Used to initialize the appropriate GDOs with the appropriate values at startup.</td>
</tr>
<tr>
<td>DBS control files</td>
<td>Lists various database control files found in the DBS Control record. Used to initialize the appropriate GDOs with their respective appropriate values at startup.</td>
</tr>
<tr>
<td>TPA subsystem startup initializations</td>
<td>Initializes or recalculates various Optimizer parameters, including the optimizer target table GDO used by the target level emulation software (see Chapter 8: “Target Level Emulation”).</td>
</tr>
</tbody>
</table>
Chapter 2: Query Rewrite and Optimization
Partition Elimination

**Partition Elimination**

**Introduction**

Partition elimination is a method for enhancing query performance against PPI tables by skipping partitions that do not contain rows that meet the search conditions of a query. Partition elimination is an automatic optimization in which the Optimizer determines, based on query conditions and a partitioning expression, that some partitions for that partitioning expression cannot contain qualifying rows; therefore, those partitions can be skipped during a file scan.

Partitions that are skipped for a particular query are called eliminated partitions.

When there are multiple partitioning expressions, partition elimination at each of the levels is combined to further reduce the subset of data that must be scanned. The greatest benefit of a PPI is obtained from partition elimination.

When query conditions allow partition elimination by the Optimizer, the form of partition elimination is called *static* (see “Static Partition Elimination” on page 300).

When partition elimination is determined while substituting USING or built-in function values in a cacheable plan, the form of partition elimination is called *delayed* (see “Delayed Partition Elimination” on page 307).

When query conditions reference values in other tables that allow for partition elimination to be done as the query is executed, the form of partition elimination is called *dynamic* (see “Product Join With Dynamic Partition Elimination” on page 366).

When there are multiple partitioning expressions, the system combines partition elimination at each of the levels to further reduce the number of data subsets that need to be scanned. For most applications, the greatest benefit of a PPI is obtained from partition elimination.

The system can eliminate partitions from search consideration at any number of levels or at a combination of levels.

The full cylinder reads optimization is not supported when there is partition elimination for either single-level or multilevel partitioning. Therefore, there are trade-offs to consider in using partitioning. The Optimizer does use partition elimination when possible, and does not cost a full-table scan with full cylinder reads against the cost of reading a subset of partitions using block reads. In most cases, it is reasonable to assume that partition elimination provides a greater benefit than a full-table scan with full cylinder reads.
Types of Partition Elimination

Teradata Database supports several different types of partition elimination:

- **Static** (see “Static Partition Elimination” on page 300)
  
  When query conditions are such that they allow partition elimination to be specified by the Optimizer during the early stages of query optimization, the form of partition elimination used is referred to as *Static Partition Elimination*. Any single-table constraints on partitioning columns can be used for static partition elimination, including those on the system-derived column `PARTITION` or any of the members of the system-derived `PARTITION#Ln` column set.66

- **Delayed** (see “Delayed Partition Elimination” on page 307)
  
  When query conditions are based on a comparison derived in part from USING request modifier variables or from the result of a built-in function, it is not possible for the Optimizer to reuse a cached query plan as it would otherwise do because a cached plan needs to be general enough to handle changes in search condition values in subsequent executions.

  In this case, the Optimizer applies partition elimination at a later point in the optimization process, at the time it builds the finalized query plan from a cached plan using the values for this specific execution of the plan. This form of partition elimination is referred to as *Delayed Partition Elimination*.

- **Dynamic** (see “Product Join With Dynamic Partition Elimination” on page 366)
  
  When query conditions reference values in other tables that would allow partition elimination as the query is executed, partition elimination is performed dynamically by the AMP database software after a query has already been optimized and while it is executing. This form of partition elimination is referred to as *Dynamic Partition Elimination*.

  Dynamic partition elimination can also be used to simplify and enhance join performance by selecting the least costly method from a set of join methods especially designed to be used with partition elimination.

Partition elimination methods can be mixed within the same query. For example, Static Partition Elimination can be used for some partition levels, while Dynamic Partition Elimination can be used for other levels, and some levels might not require *any* partition elimination. Some individual partition levels might even benefit from a mix of multiple forms of partition elimination.

---

66. Where the value of *n* ranges from 1 through 15, inclusive. See *Database Design* and *SQL Data Definition Language* for more information about the system-derived `PARTITION` columns.
Static Partition Elimination

Introduction

Static Partition Elimination, especially when applied at multiple levels, can significantly reduce the number of data blocks the system must read and process to respond to a query. Static Partition Elimination is based on constant conditions on the partitioning columns and system-derived columns PARTICITION and PARTICITION#L_n.

The system applies Static Partition Elimination to each level independently, and combines the result into a single partition elimination list. If the system cannot eliminate any partitions at a level, then it must process all the partitions for that level. The same algorithms as used for single-level partitioning are used at each level. The exception to this is conditions on system-derived column PARTICITION#L_n, which the partition elimination algorithms consider for level _n_ instead of conditions on system-derived column PARTICITION. The system considers conditions on both the system-derived column PARTICITION and the system-derived column PARTICITION#L_1 for single-level partitioning.

You should always run EXPLAIN request modifiers and measure actual system performance to verify that the desired partition elimination and performance occur for a candidate partitioning scheme and workload. If the results of your verification tests are not acceptable, or if they are not at the performance levels you expected, you should consider using a different partitioning scheme, or even using no primary index partitioning, and then make other physical database design changes or query modifications to obtain the level of performance you need and want.

67. The term constant conditions refers to predicate conditions such as equality, inequality, and BETWEEN.
68. Such as adding constant conditions on the partitioning columns.
Examples

The queries in the following example set all use the multilevel PPI table defined by the following CREATE TABLE statement.

```sql
CREATE TABLE markets (
    productid INTEGER NOT NULL,
    region BYTEINT NOT NULL,
    activity_date DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    revenue_code BYTEINT NOT NULL,
    business_sector BYTEINT NOT NULL,
    note VARCHAR(256)
) PRIMARY INDEX (productid, region)
PARTITION BY (
    RANGE_N(region BETWEEN 1 AND 9 EACH 3),
    RANGE_N(business_sector BETWEEN 0 AND 49 EACH 10),
    RANGE_N(revenue_code BETWEEN 1 AND 34 EACH 2),
    RANGE_N(activity_date BETWEEN DATE '1986-01-01' AND DATE '2007-05-31' EACH INTERVAL '1' MONTH));
```

Note the following details about the partitioning of this table:

- The following four levels of partitioning are defined with the indicated number of partitions per level:

<table>
<thead>
<tr>
<th>Number of Partitioning Level</th>
<th>Name of Column Defining Partition Level</th>
<th>Number of Partitions Defined at This Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>region</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>business_sector</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>revenue_code</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>activity_date</td>
<td>257</td>
</tr>
</tbody>
</table>

- The total number of partitions defined for the table is the maximum: 65,535, calculated as follows:

\[
\sum \text{partitions} = P_{L1} \times P_{L2} \times P_{L3} \times P_{L4}
\]

\[
= 3 \times 5 \times 17 \times 257
\]

\[
= 65,535
\]
where:

<table>
<thead>
<tr>
<th>Equation element …</th>
<th>Specifies …</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_{Ln} )</td>
<td>the number of partitions at level ( n ).</td>
</tr>
</tbody>
</table>

For the sake of simplicity, the comments associated with the example queries make the following assumptions:

- Rows are evenly distributed among the partitions.
- Many data blocks per partition can be combined.

If the example queries were not able to capitalize on Static Partition Elimination, they would all require all-AMP, full-table scans to be processed.

**Example 1**

For the following query, the system reads only partition 2 at level 1 on all AMPs. As a result, it reads roughly 1/3 (33.3 percent) of the data blocks for the `markets` table because there are 3 partitions at level 1, and the system needs to read only one of them to access all of the qualified rows.

Note that partition 2 includes rows with a value of 5 or 6 for `region`. The system necessarily reads those rows as it reads partition 2, but they do not qualify for the query predicate, so they are not returned in the results set.

```sql
SELECT * 
FROM markets 
WHERE region = 4;
```

**Example 2**

For the following query, the system reads only partitions 4 and 5 at level 2 on all AMPs. As a result, it reads roughly 2/5 (40 percent) of the data blocks for the `markets` table because there are 5 partitions at level 2, and the system needs to read 2 of those 5 to access all of the qualified rows.

Note that partition 4 includes rows whose value for `business_sector` is 30. The system necessarily reads those rows as it reads partition 4, but they do not qualify for the query predicate, so they are not returned in the results set.

```sql
SELECT * 
FROM markets 
WHERE business_sector > 30;
```
Example 3

For the following query, the system reads only partitions 1, 2, and 3 at level 3 on all AMPs. As a result, it reads roughly 3/17 (17.6 percent) of the data blocks for the *markets* table because there are 17 partitions at level 3, and the system needs to read only 3 of them to access all of the qualified rows.

Note that partition 3 includes rows who value for revenue_code is 6. The system necessarily reads those rows as it reads partition 5, but they do not qualify for the query predicate, so they are not returned in the results set.

```
SELECT *
FROM markets
WHERE revenue_code < 5;
```

Example 4

For the following query, the system reads only partitions 50 and 51 at level 4 on all AMPs. As a result, it reads roughly 2/257 (0.78 percent) of the data blocks for the *markets* table because there are 257 partitions at level 4, and the system needs to read only 2 of them to access all of the qualified rows.

Note that both partition 50 and partition 51 includes rows that do not meet the predicate for the query. The system necessarily reads those rows as it reads the partitions, but they do not qualify for the query predicate, so they are not returned in the results set.

```
SELECT *
FROM markets
WHERE activity_date >= DATE '1990-02-12'
AND   activity_date <= DATE '1990-03-28';
```

Example 5

For the following query, the system reads partitions 4 and 5 at level 2 on all AMPs and partition 2 at level 1 on all AMPs. As a result, it reads roughly 2/15 (13.3 percent) of the data blocks for the *markets* table because there are 15 partitions at combined levels 1 and 2,^69^ 3 at level 1 and 5 at level 2, and the system needs to read only 2 of them to access all of the qualified rows.

Note that all of these partitions contain some rows that do not qualify for the query predicate, so the system does not return them in the results set.

```
SELECT *
FROM markets
WHERE region = 4
AND   business_sector > 30;
```

---

^69^ The combined total is 3 x 5 = 15.
Example 6

For the following query, the system reads the following set of partitions at the indicated levels on all AMPs: Two partitions of level 4, partitions 50 and 51, in three partitions at level 3, partitions 1, 2, and 3, and in two partitions at level 2, partitions 4 and 5.

<table>
<thead>
<tr>
<th>Level Number</th>
<th>Number of Partitions Read at This Level</th>
<th>Partition Numbers Read Within the Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4, 5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>50, 51</td>
</tr>
</tbody>
</table>

As a result, it reads roughly \( \frac{12}{21,845} \) (0.05 percent) of the data blocks for the markets table because there are 21,845 partitions\(^{70}\) at combined levels 2, 3, and 4, and the system needs to read only 12 of them\(^{71}\) to access all of the qualified rows.

Note that all of these partitions contain some rows that do not qualify for the query predicate, so the system does not return them in the results set.

```sql
SELECT *
FROM markets
WHERE business_sector > 30
AND revenue_code < 5
AND activity_date >= DATE '1990-02-12'
AND activity_date <= DATE '1990-03-28';
```

---

70. The combined total is \( 5 \times 17 \times 257 = 21,845 \).
71. The combined total is \( 2 \times 3 \times 2 = 12 \).
Example 7

For the following query, the system reads the following set of partitions at the indicated levels on all AMPs: Two partitions of level 4, partitions 50 and 51, in three partitions at level 3, partitions 1, 2, and 3, in two partitions at level 2, partitions 4 and 5, in one partition of level 1, partition 2.

<table>
<thead>
<tr>
<th>Level Number</th>
<th>Number of Partitions Read at This Level</th>
<th>Partition Numbers Read Within the Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>• 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>• 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>• 50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 51</td>
</tr>
</tbody>
</table>

As a result, it reads roughly \( \frac{12}{65,535} \) (0.18 percent) of the data blocks for the `markets` table because there are 65,535 partitions\(^{72}\) at combined levels 2, 3, and 4, and the system needs to read only 12 of them\(^{73}\) to access all of the qualified rows.

Note that all of these partitions contain some rows that do not qualify for the query predicate, so the system does not return them in the results set.

```
SELECT *
FROM markets
WHERE region = 4
AND business_sector > 30
AND revenue_code < 5
AND activity_date >= DATE '1990-02-12'
AND activity_date <= DATE '1990-03-28';
```

\(^{72}\) The combined total is \(3 \times 5 \times 17 \times 257 = 65,535\).

\(^{73}\) The combined total is \(1 \times 2 \times 3 \times 2 = 12\).
Example 8

For the following query, the system reads one partition at level 2 on all AMPs: partition 1. As a result, it reads roughly \( \frac{1}{5} \) (20.0 percent) of the data blocks for the *markets* table because there are 5 partitions at level 2, and the system needs to read only one of them to access all of the qualified rows.

The partition the system must read contains the rows having a value for *business_sector* between 0 and 9, inclusive.

Note that this partition contains some rows that do not qualify for the query predicate, so the system does not return them in the results set.

```sql
SELECT *
FROM markets
WHERE PARTITION#L2 = 1;
```

Example 9

For the following query, the system reads one combined partition on all AMPs because the specified predicate is an equality condition on the combined partition that is equal to 32,531. As a result, it reads only \( \frac{1}{65,525} \) (0.15 percent) of the data blocks for the *markets* table because there is only one combined partition that has the value 32,531, and the system needs to read only the data block that contains combined partition number 32,531 to access all of the qualified rows.

The partition the system must read contains the rows defined by the following value set:

<table>
<thead>
<tr>
<th>PARTITION#Ln</th>
<th>PARTITION Value Where Qualified Rows Are Stored</th>
<th>Conditions Mapped To This PARTITION Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 region BETWEEN 4 AND 6</td>
<td>region BETWEEN 4 AND 6</td>
</tr>
<tr>
<td>2</td>
<td>3 business_sector BETWEEN 20 AND 29</td>
<td>business_sector BETWEEN 20 AND 29</td>
</tr>
<tr>
<td>3</td>
<td>8 revenue_code BETWEEN 15 AND 16</td>
<td>revenue_code BETWEEN 15 AND 16</td>
</tr>
</tbody>
</table>

```sql
SELECT *
FROM markets
WHERE PARTITION = 32531;
```
Delayed Partition Elimination

Introduction

Partition elimination can occur with conditions comparing a partitioning column to a USING request modifier variable, including host variables, or with built-in function such as CURRENT_DATE. Note that the system does not cache such plans because cached plans must be suitably generalizable to handle changes in the cached expressions in subsequent executions of the plan (see “Peeking at the Request Cache” on page 36). However, there are still optimization opportunities available for such conditions. For example, in certain cases, the system can delay partition elimination until it builds the finalized plan from a cached plan using the values for this specific execution of the plan.

Equality Conditions That Define a Single Combined Partition

Delayed Partition Elimination occurs for USING request modifier variables and built-in functions for equality over all partitioning levels on all partitioning columns. In other words, constraints must exist such that a single partition of the combined partitioning expression is specified when the system applies the values of the USING request modifier variable or built-in function to build a finalized plan from a cacheable plan.

Some of the equality conditions might be constant conditions. No restrictions are defined on the form of the partitioning expressions; however, the following restriction does exist on the form of an equality constraint: it must be a simple equality condition between a partitioning column and any of the following:

- USING request modifier variable
- Built-in function
- Constant expression

There must be an ANDed equality condition on the partitioning columns such that a single partition of the combined partitioning expression is specified.
For example, consider the following table definition:

```sql
CREATE TABLE markets (
    productid INTEGER NOT NULL,
    region BYTEINT NOT NULL,
    activity_date DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    revenue_code BYTEINT NOT NULL,
    business_sector BYTEINT NOT NULL,
    note VARCHAR(256)
) PRIMARY INDEX (productid, region)
PARTITION BY
    RANGE_N(region) BETWEEN 1 AND 9 EACH 3,
    RANGE_N(business_sector) BETWEEN 0 AND 49 EACH 10,
    RANGE_N(revenue_code) BETWEEN 1 AND 34 EACH 2,
    RANGE_N(activity_date) BETWEEN DATE '1986-01-01' AND DATE '2007-05-31' EACH INTERVAL '1' MONTH)
```

This is the same table that was defined in “Examples” on page 301.

For the sake of simplicity, the comments associated with the example queries make the following assumptions:

- Rows are evenly distributed among the partitions.
- Many data blocks per partition can be combined.

If the example queries were not able to capitalize on Delayed Partition Elimination, they would all require all-AMP, full-table scans to be processed unless otherwise noted.

For the following query, which specifies both a set of USING request modifier values and a built-in function, the system reads one combined partition\(^74\) on all AMPs. As a result, it reads only some of the data blocks for the `markets` table because with all partition levels specified in the predicate, all 65,535 partitions are eligible to be scanned, but the system needs to read only the one combined partition of those possible 65,535 partitions to access all of the qualified rows.

Note that the combined partition includes rows whose values for the query predicate do not qualify. The system necessarily reads those rows as it reads the combined partition, but because they do not qualify for the query predicate, the system does not return in the result set.

```sql
USING (r BYTEINT, b BYTEINT, rc BYTEINT)
SELECT *
FROM markets
WHERE region = :r
    AND business_sector = :b
    AND revenue_code = :rc
    AND activity_date = CURRENT_DATE;
```

\(^74\) Combined because the conditions in the predicate are on all partition levels of the table.
# Relational Query Optimization References

The following references are helpful for acquiring a basic understanding of how relational query optimizers work:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origins of Relational Query Optimizers: Peterlee Relational Test Vehicle (PRTV)</td>
<td>P.A.V. Hall, “Optimization of Single Expressions in a Relational Data Base System,” <em>IBM Journal of Research and Development</em>, 20 (3):244-257, 1976. Available as a free PDF download from the following URL: <a href="http://domino.research.ibm.com/tchjr/journalindex.nsf/4ac37cf0bdc4dd6a85256547004d47e1/1f0b4f94986b761a85256bfa0067f7a6?OpenDocument">http://domino.research.ibm.com/tchjr/journalindex.nsf/4ac37cf0bdc4dd6a85256547004d47e1/1f0b4f94986b761a85256bfa0067f7a6?OpenDocument</a> S.J.P. Todd, “The Peterlee Relational Test Vehicle—a system overview,” <em>IBM Systems Journal</em>, 15(4):285-308, 1976. Available as a free PDF download from the following URL: <a href="http://domino.research.ibm.com/tchjr/journalindex.nsf/e90fc5d047e64ebf85256bfa0066919c/8ce0a2b6d80786d985256bfa00685ad6?OpenDocument">http://domino.research.ibm.com/tchjr/journalindex.nsf/e90fc5d047e64ebf85256bfa0066919c/8ce0a2b6d80786d985256bfa00685ad6?OpenDocument</a> The PRTV was a truly relational prototype system developed at the IBM Research Laboratories in Peterlee, England. Its funding was stopped for unknown reasons in favor of the less-true-to-the-relational-model System R prototype that was being developed in California. Unfortunately, very little documentation of the project, most of which was in the form of internal IBM technical reports, is available. These two papers describe the PRTV optimizer, which, like the rest of the PRTV architecture, was quite advanced for its time.</td>
</tr>
<tr>
<td>Topic</td>
<td>Reference</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Karel Youssefi and Eugene Wong, “Query Processing In a Relational Database Management System,” <em>Proceedings of 5th International Conference on Very Large Data Bases</em>, 409-417, 1979.</td>
</tr>
<tr>
<td>Topic</td>
<td>Reference</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
Two fairly high-level product overviews of various newer IBM query optimizers. Stephen Brobst is now a Teradata Database Fellow. |
An older, but representative, technical description of the early 90s versions of the DB2 optimizer running under the OS/390, AIX, and OS/2 operating systems. |
A technical description of the SQL query optimizer for the early 90s version of the relational database management system running on the AS/400 departmental computer. Note that all the IBM relational database management systems are now called DB2, though they do not share a common code base. |
Presents a concise summary of the general principles of cost-based query optimization for relational database management systems. |
This textbook presents an excellent overview of optimization in a rigorous, but less formal manner than Ullman (1982). |
Extends Jarke and Koch (1984) by adding a great deal of material on optimization in a parallel relational environment. |
Also published as *Optimization of SQL Queries for Parallel Machines*, Ph.D. dissertation, Department of Computer Science, Stanford University, 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 a slightly revised version of the Ph.D. dissertation submitted by the author to Stanford University. |
<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
</table>
An excellent overview of the basic principles of query optimization in a relational environment. |
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. |
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. |
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. |
A basic, somewhat dated, overview of relational database statistics: how they are gathered and how they are used. |
### Topic | Reference
--- | ---

  Provides an overview of the general concept of histograms, their use as synopsis data structures for query optimization in relational database management systems, and some possible future applications for data mining.


  Demonstrates that high-biased interval histograms are always optimal for estimating join cardinalities for a commonly used form of equijoin.


  Introduces the concept of statistical histograms (for query optimization in relational database management systems).


  Provides a capsule review of older types of interval histograms used in relational systems and introduces a few new types, including compressed histograms.


  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 Database were developed by this group.
### Topic | Reference
--- | ---


Various approaches to the question of whether statistical samples are sufficiently accurate to support query optimization. The short answer is yes.

Proves that join optimization errors due to out of date statistics propagate at an exponential rate as a function of the number of joins in a query.
<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A collection of 32 papers on various applications of materialized views to query processing in general and query optimization in particular.</td>
</tr>
<tr>
<td></td>
<td>A comprehensive introduction to the mathematical foundations of relational database management with applications.</td>
</tr>
<tr>
<td></td>
<td>A review of the performance of common SAT algorithms.</td>
</tr>
<tr>
<td></td>
<td>The Knuth and Sipser texts cover complexity theory and the Landau notation as it applies to topics in computer science. The Landau text is written from a purely mathematical perspective.</td>
</tr>
<tr>
<td></td>
<td>Two survey papers on descriptive complexity by the leading computer scientist in the field.</td>
</tr>
</tbody>
</table>
Chapter 2: Query Rewrite and Optimization

Relational Query Optimization References

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
</table>
A free and continuously updated PDF electronic edition of this book is available at the following URL: [http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/GraphTheoryIII.pdf](http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/GraphTheoryIII.pdf).  
Note that the PDF version of the book cannot be printed.  
An assortment of introductory textbooks on graph theory. The Chartrand and Trudeau books are published as inexpensive paperbound editions. |
The algorithm developed by Rosenkrantz and Hunt is the best known for solving problems of satisfiability and transitive closure. It is based on the so-called shortest path algorithm of Floyd, which is often referred to as the Floyd-Warshall algorithm.  
The method is a graph analysis algorithm for finding shortest paths in a weighted, directed graph, which itself is related to work by the logician Stephen Kleene and the computer scientist Stephen Warshall. |
Ahmad Ghazal and Thu Pham, *Teradata Block Optimization*, Teradata Database Orange Book 541-0003787.  
Bill McKenna and Ahmad Ghazal, *Query Rewrite*, Teradata Database Orange Book 541-0006382.  
Tony Rollins and Ranjan Priyadarshi, *OCES Type 2 Costing*, Teradata Database Orange Book 541-0006384. |
This chapter describes the fundamentals of join planning and optimization.

The information provided is designed to help you to interpret EXPLAIN reports more accurately.
Chapter 3: Join Planning and Optimization
Optimizer Join Plans

Optimizer Join Plans

Introduction
Selecting the optimum method to join relations 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 344.

There are many methods for joining relations, 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 \( \Theta \) join, where \( \Theta \) is any valid nonequality condition\(^1\)), the estimated cardinality of the relations 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.

Components of a Join Plan
Join planning involves three core components:

- Selecting a join method.
  There are often several possible methods that can be used to make the same join. For example, it is usually, but not always, less expensive to use a Merge Join rather than a Product Join. The choice of join method often has a major effect on the overall cost of processing a query.

- Determining an optimal join geography.
  Different methods of relocating rows to be joined can have very different costs. For example, depending on the size of the relations in a join operation, it might be less costly to duplicate one of the relations rather than redistributing it.

- Determining an optimal join order.
  Only two relations can be joined at a time. The sequence in which relation pairs\(^2\) are joined can have a powerful impact on join cost.

---

1. As the term is generally defined, a \( \Theta \) join is not necessarily made on an inequality condition, but the term generally implies an inequality.

2. In this context, a table could be joined with a spool file rather than another table. The term table is used in the most generic sense of the word, and the logical term relation is often used as a substitute for any table, view, or spool file.
The following table outlines some of the terminology introduced by join planning:

<table>
<thead>
<tr>
<th>Term Type</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bind</td>
<td>An equality constraint between two columns of different relations. Normally used to determine whether there are conditions on an index.</td>
<td>(t1.a = t2.a)</td>
</tr>
<tr>
<td>Cross</td>
<td>A predicate with expressions on different relations. A cross term can be generated for a condition of the form column_name=expression, which is referred to as a half cross term. The Optimizer can use a half cross term as a constraint if it is specified on an index. The form conversion(column)=expression can be used for the same purpose if conversion(column) and column are hashed the same.</td>
<td>(t1.a = t2.a + 1)</td>
</tr>
<tr>
<td>Exists</td>
<td>A predicate that specifies an EXISTS condition.</td>
<td></td>
</tr>
<tr>
<td>Explicit</td>
<td>A predicate defined on a constant. Normally used to determine whether there are conditions on an index.</td>
<td>(t1.a = 1)</td>
</tr>
<tr>
<td>Minus</td>
<td>A predicate that specifies a MINUS condition.</td>
<td></td>
</tr>
</tbody>
</table>
| Miscellaneous | Literally a group of terms that do not belong to any of the other categories. The set of miscellaneous terms includes the following:  
  • Inequality terms.  
  • Equality terms on either the same set of relations or on nondisjoint sets of relations.  
  • ANDed terms.  
  • ORed terms.  
  • Other terms (for example, terms expressed over more than 2 relations).  
  A miscellaneous term can be generated for a conversion(column) = constant condition. If conversion(column) and (column) hash the same, then the miscellaneous term points to an explicit term.  
  For example, if the operation undertaken by the conversion is a simple renaming of the column. In this case, the miscellaneous term is used to determine whether a constraint is specified on an index column. |                       |
| Outer join  | An outer join term.                                                                                                                                                                                      |                       |
The first thing the Optimizer looks for when planning a join is connecting conditions, which are predicates that connect an outer query and a subquery. The following are all examples of connecting conditions:

\[
(t1.x, t1.y) \text{ IN } (\text{SELECT } t2.a, t2.b \text{ FROM } t2) \\
\rightarrow (t1.x \text{ IN } \text{Spool1.a AND (t1.y IN Spool1.b)}
\]

\[
(t1.x, t1.y) \text{ IN } (\text{SELECT } t2.a, \text{constant FROM } t2) \\
\rightarrow (t1.x \text{ IN } \text{Spool1.a AND (t1.y=spool1.constant)}
\]

\[
(t1.x, \text{constant}) \text{ NOT IN } (\text{SELECT } t2.a, t2b \text{ FROM } t2) \\
\rightarrow (t1.x \text{ NOT IN Spool1.a AND (constant NOT IN Spool1.b)}
\]

The following information provides a general overview of how the Optimizer analyzes conditions to determine the connections between relations in a join operation:

- There is a direct connection between two relations if either of the following conditions is found:
  - An ANDed bind, miscellaneous, cross, outer join, or minus term that satisfies the dependent info between the two relations.
  - A spool file of a noncorrelated subquery EXIST condition connects with any outer relation.
  - An ANDed miscellaneous or explicit term on a single relation is pushed to the relation.
  - A term on no relation is pushed to a relation.
  - An ORed term that references some subqueries and a single relation is associated with that relation as a complex term.
  - All relations that are referenced in an ORed term that specifies subqueries and more than one relation are put into complex set.
  - All relations that are specified in some join condition are marked as connected.
  - Assume selection and projection are done if a relation is spooled before join, for example:

\[
\text{SELECT t1.x1} \\
\text{FROM t1, t2} \\
\text{WHERE y1=1} \\
\text{AND x1= x2;}
\]

- Find the following information about all relations in the set of input relations:
  - Its row size after applying projection.
  - Its cardinality after applying selection conditions.
  - The cost to read it based on the previously determined row size and cardinality.
  - Its output row cost.
  - The maximum cardinality is used to estimate the cost of a nested join.
  - The poorest cardinality estimate is displayed in the EXPLAIN text for the query.
  - Its primary index if it has one.
  - A pointer to table descriptor of useful indexes.
  - The set of input relations for the join.
• A flag to indicate whether the rows are sorted by the primary index
• The set of connected relations, for example:
  
  ```sql
  SELECT t1.x1
  FROM t1, t2
  WHERE y1=1
  AND  x1= x2;
  ```
• Find the following information about all base table relations in the set of input relations:
  • The relation row size.
  • The relation cardinality.
  • The selection condition list.
  • The projection list.
  • The best possible access paths (using calls to access planning functions).
• Find the following information about all spool relations in the set of input relations:
  • The spool cardinality
  • The selection condition list.
  • Its assignment list.
  • Its spool number.

**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 “Join Methods” on page 361 and the pages following for more information about each of these join methods.
Limit on the Number of Tables or Single-Table Views That Can Be Joined

Excluding self-joins, as many as 128 tables or single-table views can be joined per query block.\(^3\)

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 relation, including those using correlation names, counts against the limit of 128 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 128 tables and the outer query is limited to 127 tables, the 128\(^{th}\) 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 128 tables, its outer query limited to 127 (127 plus the result of its inner query), and the parent outer query to 127 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 128 per query block, the cumulative number of tables referenced in the course of optimizing the query can be considerably greater than 128.

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 128 table join limit.

Join Relations on the Same Domain

Always join tables and views 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 (see SQL Functions, Operators, Expressions, and Predicates for information about implicit data type conversions). The conversion process is resource-intensive and thus a performance burden.

Distinct user-defined data types (UDTs) are a good method of defining domains that also bring strong typing into the picture, eliminating the possibility of implicit type conversions unless the UDT designer explicitly codes them.

3. The maximum number of tables and single-table views that can be joined per query block is determined by the value of the MaxJoinTables cost profile option parameter, which ranges from a minimum of 16 to a maximum of 128, and the MaxJoinTables performance flag of the DBS Control record, which ranges from a minimum of 64 to a maximum of 128. See Utilities for details.
This positive is balanced by the negative that you cannot define constraints on UDT columns. Because of this limiting factor, whether distinct UDTs are a good way for you to define your domains or not is a matter of balancing several factors and determining which method, if either, best suits domain definitions in your development environment.

See SQL External Routine Programming and SQL Data Definition Language for information about UDTs. See Database Design for information about using UDTs to define domains.

### Ensure You Have Allocated Enough Parser Memory for Large Joins

If you plan to join more than 32 relations in a request, use the DBS Control Utility (see Utilities) to increase the value for the DBS Control Record MaxParseTreeSegs performance flag to something greater than the default 1,000 (for 32-bit systems) or 2,000 (for 64-bit systems) parse tree memory segments allocated to Parser memory for code generation.

<table>
<thead>
<tr>
<th>FOR systems with this address size</th>
<th>THE minimum value for MaxParseTreeSegs is</th>
<th>THE default value for MaxParseTreeSegs is</th>
<th>AND the maximum value for MaxParseTreeSegs is</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>12</td>
<td>1,000</td>
<td>6,000</td>
</tr>
<tr>
<td>64-bit</td>
<td>24</td>
<td>2,000</td>
<td>12,000</td>
</tr>
</tbody>
</table>

The MaxParseTreeSegs flag defines the maximum number of 64 Kbyte parse tree memory segments the Parser allocates when parsing an SQL request. Teradata Database allocates the parse tree segments dynamically as they are needed, so this flag does not unnecessarily preallocate memory that might not be used.

### Evaluating Join Costing

Because that the Optimizer is cost-based, it evaluates the relative costs of the available join methods (see “Join Methods” on page 361) to determine the least expensive method of joining two relations.

Product joins (see “Product Join” on page 363) tend to be the most costly join method, and the system uses them only when it produces a better plan than a merge join (see “Merge Join” on page 372). For equality joins only, a hash join (see “Hash Join” on page 397) can be the least costly join method, particularly if the smaller relation of the join pair is small enough that it can all fit into memory. In this case, a special hash join called a dynamic hash join can be used (see “Dynamic Hash Join” on page 404).

As an example, consider an equality join. In this case, the smaller relation is duplicated on all AMPS and then every row in the smaller relation is compared with every row in the larger relation. The total number of comparisons to do the join is the product of the number of rows in the smaller relation and the number of rows in the larger relation.
An optimal solution for reducing the number of comparisons made by a Product Join is to assign the smaller relation rows to a hash array and then use a Hash Join instead of a Product Join. In this case, each row in the right relation does a simple table look up in the hash array to see if an entry exists or not.

<table>
<thead>
<tr>
<th>IF there is ...</th>
<th>THEN the system ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>no match in the hash array</td>
<td>discards the row and makes no further comparisons.</td>
</tr>
<tr>
<td>a match in the hash array</td>
<td>continues processing to determine if the remaining join conditions match.</td>
</tr>
</tbody>
</table>

No other entries in the hash array need to be checked.

For an equality Product Join, if the estimated cardinality of the small relation is 5 rows and the actual cardinality is 500 rows, then each row in the right relation would have to make 500 comparisons instead of the estimated 5 comparisons. This is a difference of two orders of magnitude.

For a Hash Join, each row in the right relation only needs to make one probe into the hash array. To make this join even more efficient, the system can compute the row-hash code dynamically instead of taking an extra step to create a spool file containing the row hash code based on the join columns. This optimization duplicates the left relation and generates the right relation hash codes dynamically (see “Dynamic Hash Join” on page 404).

Replacing the equality Product Join with a Dynamic Hash Join also allows for other optimizations. For example, suppose a system has more than 100 AMPs. The smaller relation in a join has 100 rows evenly distributed across the AMPs.

Consider the following three join options for these relations. The options are numbered for reference purposes, not for precedence:

1. The Optimizer can generate a join plan to redistribute the larger relation and do a Product Join with approximately one row per AMP.
   The join can be on the primary index of the small relation, which means redistributing the small relation is unnecessary, or the join could be on a nonprimary index set of columns. In the latter case, the small relation will also have to be redistributed. After redistributing the larger relation, only one comparison per row, on average, is needed because there is only one row per AMP in the small relation.

2. The Optimizer can decide to duplicate the smaller relation and then do a product join with 100 rows so that every row in the large relation requires 100 comparisons.
   The usefulness of this option depends on the size of the larger relation.

3. The Optimizer can decide to duplicate the smaller relation and then sort both the large and small relations followed by a merge join.

Option 1 is faster than Option 2. In Option 1, rows from the large relation are read from the disk and redistributed to another AMP, so there is a higher preparation cost, but a much lower join cost than Option 2.
Similarly, there is a higher preparation cost in Option 3, than Option 2 because the large relation needs to be sorted. However, by replacing the equality Product Join with a Hash Join, the smaller relation can be duplicated instead of redistributing the larger relation. The comparison costs are approximately the same because only one comparison per row is required in both cases. The larger relation only needs to be read once and it does not have to be sorted. In other words, this method provides the benefit of a lower preparation cost as well as a lower join cost by using a Hash Join.

Note that the ordering of joins is a separate process from the selection of methods to use for those joins. See “Evaluating Join Orders” on page 344 for information about how the Optimizer evaluates join orders.

**When the Optimizer Uses Secondary Indexes in a Join Plan**

Unique secondary indexes (USIs) and nonunique secondary indexes (NUSIs) 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 \( n \)-way Joins**

If more than two relations are specified in a join, the Optimizer reduces that join to a series of binary joins and then attempts to determine the most cost effective join order.

For example, assume you submitted the following multitable request:

```sql
SELECT ...
FROM A, B, C, D
WHERE ...;
```

The Optimizer generates join plans like the three diagrammed in the following graphic.\(^4\) 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.

---

\(^4\) Relational database management systems do not perform physical 3-way joins as the join plan at the extreme left of the diagram suggests by joining the AB spool file with relations C and D. The 3-way join plan in the diagram is provided for conceptual purposes only.
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 relations being joined.

\[
\text{Number of join orders} = n! 
\]

For a four-relation 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-relation join:

```
  a b c d
 /   /   \
\ abc \d  \ab \c
  \  / \  / \  / \n  a b
```

**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 Relations To Be Joined” on page 330), so the Optimizer uses several heuristics to prune the search space. For example, with the exception of star joins, unconstrained joins are not considered.

Because the optimization of join orders is known to be NP-complete (Ibaraki and Kameda, 1984), all query optimizers employ sets of heuristic devices to constrain the join space to a reasonable size. This is a necessity because the complexity of the join order space is on the order of \( O\left(\frac{2(n-1)!}{(n-1)!}\right) \) (Seshadri et al., 1996).

where:

<table>
<thead>
<tr>
<th>Equation element ...</th>
<th>Represents the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O )</td>
<td>Landau complexity symbol of number theory and computation theory, meaning “order of.”</td>
</tr>
<tr>
<td>( n )</td>
<td>number of relations in the join.</td>
</tr>
</tbody>
</table>

Also see “Possible Join Orders as a Function of the Number of Relations To Be Joined” on page 330.

Complexity can be further reduced to \( O(n^3) \) by pursuing only bushy plans that exclude Cartesian products, to \( O(3^n) \) for bushy plans inclusive of Cartesian products, and to \( O(n2^n) \) for left-deep searches (Ono and Lohman, 1990).
The following equation calculates the total number of join orders (before pruning) for \( n \) relations based on a bushy join tree search:

\[
\text{Number of join orders} = n! \left( \frac{2^{(n-1)}}{(n-1)!} \right)^n
\]

The following term in this equation represents the number of combinations:

\[
\binom{2^{(n-1)}}{(n-1)}
\]

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 following term:

\[
\frac{n!}{k!(n-k)!}
\]

where:

<table>
<thead>
<tr>
<th>This variable ...</th>
<th>Represents this term ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( 2^{(n-1)} )</td>
</tr>
<tr>
<td>( k )</td>
<td>( (n-1) )</td>
</tr>
</tbody>
</table>

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

\[
\text{Number of join orders} = 4! \cdot \frac{(2^{(4-1)})!}{(4-1)!} \cdot \frac{1}{4} = 120
\]

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 relation 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 relations. The number of permutations of four relations is \( 4! \) or 24. If the relations are named \( a, b, c, \) and \( d \), then those 24 permutations are as follows:

\( abcd, abdc, acbd, acdb \ldots dcba \)
Because the system can only join two relations at a time, these 24 permutations must be further partitioned into all their possible binary join orders as follows:

\[
\text{abcd} \Rightarrow (((\text{ab})\text{c})\text{d})
\]

\[
((\text{ab})\text{(cd)})
\]

\[
(\text{a}(\text{b}(\text{cd})))
\]

\[
((\text{a}(\text{bc}))\text{d})
\]

\[
((\text{a}(\text{b}(\text{cd})))
\]

and so on.

The following graphic illustrates one possible sequence of binary joins of relations \(a, b, c,\) 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:

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 \(abcd\).

4. End of process.

Depending on table demographics and environmental costs, the Optimizer could just as likely produce the following join sequence:

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 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 Relations To Be Joined**

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

To help paint a picture of how vast the number of combinations becomes as the number of tables and single-table views increases, the following table uses the metaphor of grains of sand.

<table>
<thead>
<tr>
<th>Number of relations Joined</th>
<th>Number of Possible Join Orders</th>
<th>log(Possible Join Orders)</th>
<th>Number of Grains of Sand</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>12.0</td>
<td>1.079</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>120.0</td>
<td>2.079</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>$1.8 \times 10^{10}$</td>
<td>10.255</td>
<td>6 sand boxes</td>
</tr>
<tr>
<td>16</td>
<td>$2.0 \times 10^{20}$</td>
<td>20.301</td>
<td>All the sand grains of all the beaches and all the deserts in the world</td>
</tr>
<tr>
<td>64</td>
<td>$1.2 \times 10^{124}$</td>
<td>124.079</td>
<td>All the sand grains required to fill the known universe</td>
</tr>
<tr>
<td>128</td>
<td>$3.966 \times 10^{556}$</td>
<td>656.598</td>
<td>All the sand grains required to fill several parallel universes</td>
</tr>
</tbody>
</table>

The following graph shows the log-linear plot of the first 5 rows of data in this table: the expansion in the number of possible join orders as a function of the number of relations 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 nulls in the *shipnum* column).

<table>
<thead>
<tr>
<th>Key</th>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK</td>
<td>Primary key</td>
<td></td>
</tr>
<tr>
<td>FK</td>
<td>Foreign key</td>
<td></td>
</tr>
<tr>
<td>UPI</td>
<td>Unique primary index</td>
<td></td>
</tr>
<tr>
<td>SA</td>
<td>System-assigned</td>
<td></td>
</tr>
</tbody>
</table>
The projected columns of all 4,000,000 rows of the parts table participate in the join if the following query is submitted.

\[
\text{SELECT} \ldots \\
\text{FROM} \text{ Shipment, Part} \\
\text{WHERE} \text{ Shipment.ShipNum} = \text{ Part.ShipNum};
\]

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

\[
\text{SELECT} \ldots \\
\text{FROM} \text{ Shipment, Part} \\
\text{WHERE} \text{ Part.ShipNum IS NOT NULL} \\
\text{AND} \text{ Shipment.ShipNum} = \text{ Part.ShipNum};
\]

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Shipments</th>
<th>Statistics</th>
<th>Parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>500 rows</td>
<td>ShipNum</td>
<td>4,000,000  rows</td>
<td>PartNum</td>
</tr>
<tr>
<td>PK/FK</td>
<td>PK, SA</td>
<td>PK/FK</td>
<td>PK, SA</td>
</tr>
<tr>
<td>UPI</td>
<td></td>
<td></td>
<td>UPI</td>
</tr>
<tr>
<td>Dist values</td>
<td>500</td>
<td>Dist values</td>
<td>4.0*10^6</td>
</tr>
<tr>
<td>Max rows/val</td>
<td>1</td>
<td>Max rows/val</td>
<td>1</td>
</tr>
<tr>
<td>Typ rows/val</td>
<td>1</td>
<td>Typ rows/val</td>
<td>3.99 x 10^6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>
Join Geography

Introduction

Join geography is a term that describes the relocation, if any, of rows required to perform a join. Remember that for two rows to be joined, they must be on the same AMP, so the Optimizer must determine the least costly row relocation strategy, and the relative cardinalities of the two relations in any join is the principal cost that must be considered. This is one of the main factors that compels having fresh statistics available for the Optimizer.

How Stale Statistics Can Affect Join Planning Negatively

Suppose a query joins tables $a$ and $b$, and because these tables have different primary indexes, or because one table has a primary index and the other does not, the rows that must be joined are on different AMPs. Some sort of relocation of rows is required to make this join.

The last time statistics were collected on these tables, their respective cardinalities were 1,000 and 75,000 rows. These cardinalities clearly call for the Optimizer to relocate the few rows from table $a$ to the AMPs containing the many rows of table $b$.

However, since statistics were last collected on these tables, unforeseen events have caused large shifts in the ratio of their cardinalities, and they are now 50,000 for table $a$ and 3,500 for table $b$, as summarized by the following table:

<table>
<thead>
<tr>
<th>Table</th>
<th>Cardinality When Statistics Last Collected</th>
<th>Current Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1,000</td>
<td>50,000</td>
</tr>
<tr>
<td>b</td>
<td>75,000</td>
<td>3,500</td>
</tr>
</tbody>
</table>

Based on the statistics available to it, the Optimizer will choose to relocate the rows of table $a$, which is clearly a far more costly relocation to make given the actual cardinalities of tables $a$ and $b$ today.

This is obviously an extreme example, but it illustrates why it is critical to keep fresh statistics on all of your frequently accessed tables.

Join Distribution Strategies

The method the Optimizer chooses to use to distribute rows that are to be joined depends on several factors, but primarily on the availability of indexes and statistics. Keep the following facts about joins in mind:

- Join costs increase as a function of the number of rows that must be relocated, sorted, or both.
- Join plans for the same pairs of relations can change as a function of changes in the demographic properties of those relations.
There are four possible join distribution strategies the Optimizer can choose among, and more than one strategy can be used for any join depending on circumstances. The four join distribution strategies are the following:

- Redistribution of one or both relations in the join.
  The EXPLAIN phrase to watch for is *Redistribute*.
- Duplication of one or both relations in the join.
  The EXPLAIN phrase to watch for is *Duplicate*.
- Locally building a spool copy of a relation.
  The EXPLAIN phrase to watch for is *Locally Build*.
- Sorting a relocated relation.
  The EXPLAIN phrase to watch for is *Sort*.

There is also a fifth option, which is not to redistribute any rows. This option represents the degenerate case of row redistribution where no redistribution is required to make the join, and it occurs only when the primary indexes of the two relations are such that equal-valued primary index rows of both relations hash to the same AMPs.

### Examples of Different Row Redistribution Strategies

The following set of examples demonstrates four different row redistribution strategies, three using Merge Join and a fourth using Product Join:

- “Example 1: Redistribute The Rows Of One Table For a Merge Join” on page 334.
- “Example 2: Duplicate And Sort the Rows of the Smaller Table On All AMPs, Build and Sort a Local Copy of the Larger Table For a Merge Join” on page 336.
- “Example 3: No Row Redistribution Or Sorting For a Merge Join Because The Join Rows Of Both Tables Are On the Same AMP” on page 338.
- “Example 4: Duplicate the Smaller Table On Every AMP For a Product Join” on page 339.

Notice that the primary index is the major consideration used by the Optimizer in determining how to join two relations and deciding which rows to relocate.

### Example 1: Redistribute The Rows Of One Table For a Merge Join

This example illustrates a Merge Join strategy, which includes redistributing the rows of one table and sorting them by the row hash value of the join column. A relocation strategy is pursued because the join condition in the query is on only one of the primary indexes of the joined tables (see “Scenario 2: The Join Column Set Is The Primary Index Of Only One Of The Tables In The Join” on page 342).

The query that generates a Merge Join is the following:

```sql
SELECT *
FROM employee AS e
INNER JOIN department AS d
ON e.dept = d.dept;
```
The two tables to be joined are defined as follows:

<table>
<thead>
<tr>
<th>employee</th>
<th>name</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum</td>
<td>name</td>
<td>dept</td>
</tr>
<tr>
<td>PK</td>
<td></td>
<td>FK</td>
</tr>
<tr>
<td>UPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Higa</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>Kostamaa</td>
<td>310</td>
</tr>
<tr>
<td>3</td>
<td>Chiang</td>
<td>310</td>
</tr>
<tr>
<td>4</td>
<td>Korlapati</td>
<td>400</td>
</tr>
<tr>
<td>5</td>
<td>Sinclair</td>
<td>150</td>
</tr>
<tr>
<td>6</td>
<td>Kaczmarek</td>
<td>400</td>
</tr>
<tr>
<td>7</td>
<td>Eggers</td>
<td>310</td>
</tr>
<tr>
<td>8</td>
<td>Challis</td>
<td>310</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>department</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>dept</td>
<td>name</td>
</tr>
<tr>
<td>PK</td>
<td></td>
</tr>
<tr>
<td>UPI</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>Payroll</td>
</tr>
<tr>
<td>200</td>
<td>Finance</td>
</tr>
<tr>
<td>310</td>
<td>Manufacturing</td>
</tr>
<tr>
<td>400</td>
<td>Engineering</td>
</tr>
</tbody>
</table>

The following graphic shows how individual rows would be relocated to make this join for a 4-AMP system:

**Spool file after redistribution of the row hash of e.dept.**

**Department table rows hash distributed on d.dept.**
The example is run on a 4-AMP system. The query uses an equijoin condition on the \textit{dept} columns of both tables. The system copies the \textit{employee} table rows into spool and redistributes them on the row hash of \textit{e.dept}. The Merge Join occurs after the rows to be joined have been relocated so they are on the same AMPs.

The relocation strategy occurs when one of the tables is already distributed on the join column row hash. The Merge Join is caused by the join column being the primary index of one (\textit{dept}), but not both, of the tables.

**Example 2: Duplicate And Sort the Rows of the Smaller Table On All AMPs, Build and Sort a Local Copy of the Larger Table For a Merge Join**

This example illustrates a different Merge Join strategy that consists of duplicating and sorting the smaller table in the join on all AMPs and locally building a copy of the larger table and sorting it on the \textit{employee} table row hash (see “Scenario 2: The Join Column Set Is The Primary Index Of Only One Of The Tables In The Join” on page 342).

This query and tables used for this example are the same as those used for “Example 1: Redistribute The Rows Of One Table For a Merge Join” on page 334, but the Optimizer pursues a different join geography strategy because the statistics for the tables are different. If the Optimizer determines from the available statistics that it would be less expensive to duplicate and sort the smaller table than to hash redistribute the larger table, it chooses the strategy followed by this scenario.

The query that generates a Merge Join is the following:

\begin{verbatim}
SELECT *
FROM employee AS e
INNER JOIN department AS d
ON e.dept = d.dept;
\end{verbatim}

The two tables to be joined are defined as follows:

<table>
<thead>
<tr>
<th>employee</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum</td>
<td>name</td>
</tr>
<tr>
<td>PK</td>
<td>Higa</td>
</tr>
<tr>
<td>UPI</td>
<td>Kostamaa</td>
</tr>
<tr>
<td>1</td>
<td>Chiang</td>
</tr>
<tr>
<td>2</td>
<td>Korlapati</td>
</tr>
<tr>
<td>3</td>
<td>Sinclair</td>
</tr>
<tr>
<td>4</td>
<td>Kaczmarek</td>
</tr>
<tr>
<td>5</td>
<td>Eggers</td>
</tr>
<tr>
<td>6</td>
<td>Challis</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
The following graphic shows how individual rows would be relocated to make this join for a 4-AMP system:

Department table rows hash distributed on d.dept, the UPI.

<table>
<thead>
<tr>
<th>AMP1</th>
<th>AMP2</th>
<th>AMP3</th>
<th>AMP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>150 Payroll</td>
<td>310 Manufacturing</td>
<td>200 Finance</td>
<td>400 Engineering</td>
</tr>
</tbody>
</table>

Spool file after duplicating and sorting on d.dept row hash.

<table>
<thead>
<tr>
<th>AMP1</th>
<th>AMP2</th>
<th>AMP3</th>
<th>AMP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>150 Payroll 200 Finance 310 Manufacturing 400 Engineering</td>
<td>150 Payroll 200 Finance 310 Manufacturing 400 Engineering</td>
<td>150 Payroll 200 Finance 310 Manufacturing 400 Engineering</td>
<td>150 Payroll 200 Finance 310 Manufacturing 400 Engineering</td>
</tr>
</tbody>
</table>

Employee table rows hash distributed on e.enum, the UPI.

<table>
<thead>
<tr>
<th>AMP1</th>
<th>AMP2</th>
<th>AMP3</th>
<th>AMP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Kaczmarek 400 8 Challis 310</td>
<td>4 Korlapati 400 3 Chiang 310</td>
<td>1 Higa 200 7 Eggers 310</td>
<td>5 Sinclair 150 2 Kostamaa 310</td>
</tr>
</tbody>
</table>

Spool file after locally building and sorting on the e.dept row hash.

<table>
<thead>
<tr>
<th>AMP1</th>
<th>AMP2</th>
<th>AMP3</th>
<th>AMP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Challis 310 6 Kaczmarek 400</td>
<td>3 Chiang 310 4 Korlapati 400</td>
<td>1 Higa 200 7 Eggers 310</td>
<td>5 Sinclair 150 2 Kostamaa 310</td>
</tr>
</tbody>
</table>

The system first duplicates the department table and then sorts it on the dept column for all AMPS. Next, the employee table is built locally and sorted on the row hash value of dept.

The final step in the process is to perform the actual Merge Join operation.
Example 3: No Row Redistribution Or Sorting For a Merge Join Because The Join Rows Of Both Tables Are On the Same AMP

This example also illustrates a Merge Join strategy. This particular strategy does not require any duplication or sorting of rows because the joined rows of both tables hash to the same AMPS. This is a good example of using a NUPI for one table on the same domain as the UPI of the other table. This ensures that the rows having the same primary index values all hash to the same AMP (see “Scenario 1: The Join Column Set Is The Primary Index Of Both Relations In The Join” on page 341). When the join condition is on those primary indexes, the rows to be joined are already on the same AMP, so no relocation or sorting need be done. All the system has to do is compare the rows that are already on the proper AMPS.

This example is run on a 4-AMP System.

The query that generates this Merge Join strategy is the following:

```sql
SELECT *
FROM employee AS e
INNER JOIN employee_phone AS p
ON e.num = p.num;
```

The two tables to be joined are defined as follows:

<table>
<thead>
<tr>
<th>employee</th>
<th>employee_phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum</td>
<td>area_code</td>
</tr>
<tr>
<td>PK</td>
<td></td>
</tr>
<tr>
<td>UPI</td>
<td>FK</td>
</tr>
<tr>
<td>1</td>
<td>Higa</td>
</tr>
<tr>
<td>2</td>
<td>Kostamaa</td>
</tr>
<tr>
<td>3</td>
<td>Chiang</td>
</tr>
<tr>
<td>4</td>
<td>Korlapati</td>
</tr>
<tr>
<td>5</td>
<td>Sinclair</td>
</tr>
<tr>
<td>6</td>
<td>Kaczmarek</td>
</tr>
<tr>
<td>7</td>
<td>Eggers</td>
</tr>
<tr>
<td>8</td>
<td>Challis</td>
</tr>
</tbody>
</table>
The following graphic shows how individual rows would be relocated to make this join for a
4-AMP system:

**Employee table rows hash distributed on e.enum, the UPI.**

<table>
<thead>
<tr>
<th>AMP1</th>
<th>AMP2</th>
<th>AMP3</th>
<th>AMP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Kaczmarek 400 8 Challis 310</td>
<td>4 Korlapati 400 3 Chiang 310</td>
<td>1 Higa 200 7 Eggers 310</td>
<td>5 Sinclair 150 2 Kostamaa 310</td>
</tr>
</tbody>
</table>

**Employee_Phone table rows hash distributed on enum, the NUPI.**

<table>
<thead>
<tr>
<th>AMP1</th>
<th>AMP2</th>
<th>AMP3</th>
<th>AMP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 203 8337461 8 301 2641616 8 301 6675885</td>
<td>4 415 6347180 3 408 3628822</td>
<td>1 213 3241576 1 213 4950703</td>
<td>5 312 7463513</td>
</tr>
</tbody>
</table>

**Example 4: Duplicate the Smaller Table On Every AMP For a Product Join**

This example illustrates a Product Join strategy, which includes duplicating the smaller table
rows on every AMP of a 4-AMP system. The tables used are the same as those used in
“Scenario 2: The Join Column Set Is The Primary Index Of Only One Of The Tables In The
Join” on page 342, and the query is the same as well with the exception that the join condition
in that scenario is an equijoin, while the join condition in this query is a \( \theta \) join, where \( \theta \) is \( > \).

The query that generates a Product Join is the following:

```
SELECT *
FROM employee AS e
INNER JOIN department AS d
ON e.dept > d.dept;
```
The two tables to be joined are defined as follows:

<table>
<thead>
<tr>
<th>employee</th>
<th>name</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum</td>
<td>name</td>
<td>dept</td>
</tr>
<tr>
<td>PK</td>
<td>Higa</td>
<td>200</td>
</tr>
<tr>
<td>UPI</td>
<td>Kostamaa</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>Chiang</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>Korlapati</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>Sinclair</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Kaczmarek</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>Eggers</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>Challis</td>
<td>310</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>department</th>
<th>dept</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>dept</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>PK</td>
<td>UPI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>Payroll</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>Finance</td>
</tr>
<tr>
<td></td>
<td>310</td>
<td>Manufacturing</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>Engineering</td>
</tr>
</tbody>
</table>

The Product Join is caused by the nonequijoin condition `e.dept > d.dept` (see “Product Join” on page 363). Based on the available statistics, the Optimizer determines that the `department` table is the smaller table in the join, so it devises a join plan that distributes copies of all those rows to each AMP. `employee` table rows, which are hash distributed on their UPI `enum` values, are not relocated. The Product Join operation returns only the rows that satisfy the specified join condition for the query after comparing every row in the `employee` table with every row in the duplicated `department` table.

The following graphic shows how individual rows are relocated to make this join for the 4-AMP system:

Department table rows hashed on d.dept, the UPI.

Spool file after duplicating Department table rows.

Employee table rows hash distributed on e.enum, the UPI.
Relocation Scenarios

The primary index is the major consideration used by the Optimizer in determining how to join two tables and deciding which rows to relocate, though for tables that have no primary index, other criteria must be considered.

Three general scenarios can occur when two primary-indexed relations are joined using the Merge Join method (see "Merge Join" on page 372), as illustrated by the following table.

The scenarios are ranked in order of best to worst, with 1 being best, where best means least costly:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Scenario</th>
<th>WHEN the join column set is</th>
<th>FOR these relations in the join operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>the primary index</td>
<td>both.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>the primary index</td>
<td>one.</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>not the primary index</td>
<td>neither.</td>
</tr>
</tbody>
</table>

The three scenarios are described in the topics that follow.

**Note:** The examples in these sections assume that all join columns come from the same domain.

**Scenario 1: The Join Column Set Is The Primary Index Of Both Relations In The Join**

This is the best case scenario because the rows to be joined are already located on the same AMP. Equal primary index values always hash to the same AMP, so there is no need to relocate rows to other AMPs. The rows are also already sorted in row hash order because of the way the file system stores them. With no need to sort or relocate rows, the join can be performed immediately with very little cost.

For example, consider the following query:

```sql
SELECT ...
FROM table_1 AS t1
INNER JOIN table_2 AS t2
ON t1.col_1 = t2.col_1;
```

The join in this query is defined on the primary index columns for both tables, as you can see from the example tables presented below. Because the primary index values for the rows are equal, they hash to the same AMP. The result is that no row relocation is required to make the join.

<table>
<thead>
<tr>
<th>table_1</th>
<th>col_2</th>
<th>col_2</th>
<th>col_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI</td>
<td>100</td>
<td>214</td>
<td>433</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>table_2</th>
<th>col_2</th>
<th>col_2</th>
<th>col_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI</td>
<td>100</td>
<td>725</td>
<td>002</td>
</tr>
</tbody>
</table>
Scenario 2: The Join Column Set Is The Primary Index Of Only One Of The Tables In The Join

In this case, only one table has its rows on the target AMPs. The rows of the second table must be redistributed to their target AMPs by the hash code of the join column value. If the table is small, the Optimizer might decide to simply duplicate the entire table on all AMPs instead of hash redistributing individual rows. In either case, the system copies some or all of the rows from one table to their target AMPs. If the PI table is the smaller table, the Optimizer might choose to duplicate it on all AMPs rather than redistributing the nonprimary index table.

For example, consider the following query:

```sql
SELECT ...
FROM table_3 AS t3
INNER JOIN table_4 AS t4
ON t3.col_1=t4.col_2;
```

The join in this query is defined on the primary index column for `table_3` and on a nonprimary index column for `table_4`, as you can see from the example relations presented below. Because of this, the rows for one of the tables must be relocated to make the join.

In the example, the rows from `table_4` are chosen for relocation to their target AMPs by hash code and placed into a spool file, probably because the joined row cardinality of `table_4` is much smaller than that of `table_3`. If the joined row cardinality of `table_4` is significantly smaller than that of `table_3`, the Optimizer might even decide to duplicate the entire smaller table rather than hash redistributing a subset of its rows.

<table>
<thead>
<tr>
<th>table_3</th>
<th>table_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_1</td>
<td>col_1</td>
</tr>
<tr>
<td>col_2</td>
<td>col_2</td>
</tr>
<tr>
<td>col_3</td>
<td>col_3</td>
</tr>
<tr>
<td>PI</td>
<td>PI</td>
</tr>
<tr>
<td>255</td>
<td>867</td>
</tr>
<tr>
<td>345</td>
<td>255</td>
</tr>
<tr>
<td>225</td>
<td>566</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spool</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_1</td>
</tr>
<tr>
<td>col_2</td>
</tr>
<tr>
<td>col_3</td>
</tr>
<tr>
<td>PI</td>
</tr>
<tr>
<td>867</td>
</tr>
<tr>
<td>255</td>
</tr>
<tr>
<td>566</td>
</tr>
</tbody>
</table>

5. A small table is defined as a table whose cardinality is less than 5 times the number of AMPs in the system. For a 20-AMP system, table cardinality would have to be less than 100 rows for the table to be considered small, for a 100-AMP system, less than 500 rows, and so on.

6. There is no rule that forces the rows of the nonprimary index table to be relocated. The decision depends entirely on the available statistics for both relations in the join.
Scenario 3: The Join Column Is The Primary Index Of Neither Table In The Join

This is the worst case scenario because if neither column is a primary index, then the rows of both must be redistributed to their target AMPs. This can be done either by hash redistribution of the join column value or by duplicating the smaller table on each AMP. In either case, this approach involves the maximum amount of data movement. Your choice of a primary index should be heavily influenced by the amount of join activity you anticipate for the table.

For example, consider the following query:

```sql
SELECT ...
FROM table_5 AS t5
INNER JOIN table_6 AS t6
ON t5.col_2=t6.col_3;
```

The join in this query is defined on the nonindexed `col_2` of `table_5` and on the nonindexed `col_3` of `table_6`, as you can see from the example relations presented below. Because of this, the rows for both relations must be relocated to their target AMPs to make the join.

In the example, the rows from both `table_5` and `table_6` are placed in spool files so they can be relocated to their target AMPs by hash code. If the joined row cardinality of one of the relations is significantly smaller than that of the other, the Optimizer might even decide to duplicate the entire smaller table rather than hash redistributing a subset of its rows.

<table>
<thead>
<tr>
<th>table_5</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>col_1</td>
<td>col_2</td>
</tr>
<tr>
<td>PI</td>
<td>456</td>
<td>777</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>table_6</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>col_1</td>
<td>col_2</td>
</tr>
<tr>
<td>PI</td>
<td>993</td>
<td>228</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spool</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>col_1</td>
<td>col_2</td>
</tr>
<tr>
<td>PI</td>
<td>456</td>
<td>777</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spool</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>col_1</td>
<td>col_2</td>
</tr>
<tr>
<td>PI</td>
<td>993</td>
<td>228</td>
</tr>
</tbody>
</table>
Evaluating Join Orders

Introduction

While it is possible to select an optimum join order when a small number of relations 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.

Join order evaluation is another critical aspect of query optimization that is highly dependent on accurate statistics. Ioannidis and Christodoulakis (1991) demonstrated conclusively that errors in join plan estimation due to making inaccurate assumptions about the cardinalities of intermediate results in the process propagate exponentially as a function of the number of joins a query makes. These inaccurate assumptions are made as the result of using out of date statistical and demographic data.

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.

1. Use a recursive greedy algorithm with a one-join lookahead to evaluate the existing join conditions.

<table>
<thead>
<tr>
<th>IF the following conditions are found ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The cost of the one-join lookahead plan exceeds a threshold value.</td>
<td>Generate a five-join lookahead plan and evaluate its cost.</td>
</tr>
<tr>
<td>• There exists one relation larger than the threshold.</td>
<td></td>
</tr>
<tr>
<td>• The query has no outer join requirement.</td>
<td></td>
</tr>
</tbody>
</table>

2. Keep the less costly plan generated.

3. Evaluate the new existing join conditions.

<table>
<thead>
<tr>
<th>IF the following conditions are found ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The cost of the current plan exceeds some threshold.</td>
<td>Generate a better plan that uses star join optimization if such a plan can be found.</td>
</tr>
<tr>
<td>• A star join might be required.</td>
<td>Otherwise, go to Stage 4.</td>
</tr>
</tbody>
</table>
4 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 relations.
- Join a nonconnected relation only after all connected relations have been joined.

The Optimizer does not evaluate all possible combinations of relations because the effort would exceed any optimizations realized from it. For example, a 10-way join has \( 17.6 \times 10^9 \) possible combinations, and a 64-way join has \( 1.2 \times 10^{124} \) possible combinations! The answer to the vernacular question “Is the juice worth the squeeze?” would definitely be “No” in both of these cases.

In its pursuit of combinations that are driven mainly by join conditions, the Optimizer might overlook the best possible join plan, but as a general rule, that does not happen.

5 Create a temporary join relation before doing the next lookahead.

<table>
<thead>
<tr>
<th>IF a unique ID is ...</th>
<th>THEN do the following ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>not used</td>
<td>1  Build the assignment list for the temporary relation.</td>
</tr>
<tr>
<td></td>
<td>2  Use it to estimate the following things:</td>
</tr>
<tr>
<td></td>
<td>• Spool size</td>
</tr>
<tr>
<td></td>
<td>• Output row cost</td>
</tr>
<tr>
<td></td>
<td>3  Map the term information of the residual term list to refer to the new relation.</td>
</tr>
<tr>
<td></td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>• Spool size</td>
</tr>
<tr>
<td></td>
<td>• Output row cost</td>
</tr>
</tbody>
</table>
Evaluate the best current join plan.

<table>
<thead>
<tr>
<th>IF ...</th>
<th>THEN ...</th>
</tr>
</thead>
</table>
| the following conditions are found for the first two joins of the best join plan:  
  - The joins involve one relation connected with two relations not connected to each other  
  - Neither join is a Product Join | try a compromise join where the two unconnected relations are Product-joined in the first join and that result is then joined with a third relation. |
| the compromise join plan is less costly than the best current join plan | replace the first two joins on the best join plan with the compromise plan. |

The compromise plan is designed to enable a star join plan.

A compromise join is also considered during binary join planning when one or more IN condition are specified on one of the relations in the join. Call this `table_a`.

The list of values associated with the set of IN list conditions is product joined with the other relation, `table_b`, and that result is then joined with `table_a`. If `table_b` happens to be the result of joining two unconnected relations, then the resultant join plan is a star join plan similar to what is described in “Star and Snowflake Join Optimization” on page 442.

Consider the following schema and query, for example:

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Table Type</th>
<th>Relevant Index Columns</th>
<th>Index Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DailySales</td>
<td>Fact</td>
<td><code>sku_id</code> <code>locn_nmbr</code> <code>day_dt</code></td>
<td>NUPI</td>
</tr>
<tr>
<td>CorpDay</td>
<td>Dimension</td>
<td><code>day_dt</code></td>
<td>UPI</td>
</tr>
<tr>
<td>Locn</td>
<td>Dimension</td>
<td><code>locn_nmbr</code></td>
<td>UPI</td>
</tr>
<tr>
<td>AllocSku</td>
<td>Dimension</td>
<td><code>sku_id</code></td>
<td>UPI</td>
</tr>
</tbody>
</table>

Now consider how the Optimizer treats the following query against these tables:

```sql
SELECT SUM(sell_amt)  
FROM dailysales AS a, locn AS l  
WHERE a.locn_nmbr = l.locn_nbr  
AND l.dvsn_code = 'C'  
AND a.day_dt IN (990101, 1010101, 971201);  
```

During binary join planning for the `DailySales` and `Locn` tables, the Optimizer considers a compromise join by first using a Product Join to join `Locn` with the list of `day_dt` values specified by the IN list condition, and then joining that result with `DailySales`. 

The compromise plan is designed to enable a star join plan.
7 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>
<thead>
<tr>
<th>IF the conditions are...</th>
<th>THEN the...</th>
</tr>
</thead>
<tbody>
<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 relation with the result of the first join via the primary index for the newly joined relation.</td>
</tr>
</tbody>
</table>

8 Generate connection information for new relations based on the predicates.
9 Iterate Stages 1 through 8 until only one active relation remains.
10 End of process.

**Example**

Consider the following query:

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

The first join diagram in the following graphic illustrates the connections among the relations 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 **step 6 on page 346** that when the following conditions are true, the Optimizer tries to make a compromise join where the two unconnected relations are product-joined and the result is joined with the third relation:

- The first two joins of the best plan involve a relation connected with two relations *not* connected to one another
- Neither join is a Product Join.
The second join diagram 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 relations \( t2 \) 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 join diagram 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 352), Teradata Database 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 Optimizer search of join space is driven by join conditions. To this end, it 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.

1. Evaluate 3-way joins for the least costly join.\(^7\)
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.
6. End of process.

\(^7\) All joins are binary. The term 3-way join is used here to refer to the evaluation of the possible join orders among three relations at a time, not to making a ternary join of the three.
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:

<table>
<thead>
<tr>
<th>This binary join ...</th>
<th>Has this relative cost ...</th>
<th>And produces this relation as a result ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A - B</td>
<td>1</td>
<td>AB</td>
</tr>
<tr>
<td>AB - C</td>
<td>4</td>
<td>ABC</td>
</tr>
<tr>
<td>D - F</td>
<td>1</td>
<td>DF</td>
</tr>
<tr>
<td>DF - E</td>
<td>2</td>
<td>DFE</td>
</tr>
<tr>
<td>ABC - DFE</td>
<td>5</td>
<td>ABCDEF</td>
</tr>
</tbody>
</table>
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 relations are explicitly stated in the following table:

<table>
<thead>
<tr>
<th>This binary join ...</th>
<th>Has this relative cost ...</th>
<th>And produces this relation as a result ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>D - E</td>
<td>1</td>
<td>DE</td>
</tr>
<tr>
<td>D - F</td>
<td>2</td>
<td>DF</td>
</tr>
<tr>
<td>DE - F</td>
<td>5</td>
<td>DEF</td>
</tr>
<tr>
<td>DF - E</td>
<td>2</td>
<td>DEF</td>
</tr>
<tr>
<td>ABC - DFE</td>
<td>5</td>
<td>ABCDEF</td>
</tr>
</tbody>
</table>

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 relations \( D \) and \( F \) first following by joining the resulting relation \( DF \) with relation \( E \), 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 relations 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.
Chapter 3: Join Planning and Optimization

Partial GROUP BY Block Optimization

Introduction

The Optimizer costs all alternative access and join plans. Where appropriate, aggregations are split, with part being done before the join and part being done after the join. This means less work for the join as well as less aggregation work after the join. The idea is to reduce the cardinalities as early as possible, which also helps reduce spool use.

In a join query plan for a query with a GROUP BY clause, Partial GROUP BY operations are used to reduce the cardinalities of base tables, join indexes, and intermediate spool files, thereby reducing processing required in performing the join query plan. To further enhance efficiency, certain of the partial group operations, including an intermediate Partial GROUP BY operation, a final GROUP BY operation, or both can be eliminated in response to certain predefined conditions.

The Partial GROUP BY optimization can be done almost at any stage of the join plan. An early GROUP BY reduces the rows of the base relations or the intermediate join relations during the optimization process. However, GROUP BY and Merge Join operators have a tendency to compete with one other for use in the lowest cost plan because both apply significant row reduction. Therefore, it is extremely important for the Optimizer to cost the alternatives.

All evidence shows that the Partial GROUP BY optimization reduces the processing and I/O times for typical requests. The more duplicate grouping values a table has, the greater the reduction in I/O processing which, in some cases, can be as much as a factor of 100 times (see “Performance Analysis” on page 359).

The Optimizer has two additional paths to reduce the number of rows included in the join:

- Early GROUP BY
  
  The Optimizer does all of the aggregation prior to the join.

- Partial GROUP BY
  
  The Optimizer does some of the aggregation before the join and the rest after the join. It also learns when to skip the last aggregation to improve the performance.

Partial GROUP BY Optimization

Consider the following typical query:

```sql
SELECT b1, c1, SUM(t1.float_1), SUM(t2.float_2), SUM(t3.float_3)
FROM t1,t2,t3
WHERE b1=b2 and b2=b3
GROUP BY b1,c1;
```

Without Partial GROUP BY enabled, the optimizer selects one of the following join plans:

- `[(t1 x t2) x t3]'`
- `[(t2 x t3) x t1]'`
- `[(t1 x t3) x t2]'`
where \( x \) represents the join and \([ ]\)' denotes the GROUP BY operator applied to the relation.

With Partial GROUP BY enabled, some additional join plans are considered, such as the following:

- \([([t2]' \times t3) \times t1]')\)
- \([([t1] \times [t3]') \times t2]')\)
- \([([t2]' \times [t3]') \times t1]')\)
- \([([t1] \times t3]' \times [t2]'\)
- ...

This provides the Optimizer with an opportunity to find a lower cost join plan among a larger set of possible join plans. Even though there are more join plans to be considered, the additional search effort consumes negligible CPU time and returns a significant reduction time in the execution of the request.

The Partial GROUP BY optimization reduces cardinalities as early as possible in the optimization process.

Consider the following example:

```sql
SELECT ps_partkey, SUM(l_quantity), SUM(ps_supplycost)
FROM partsupp, lineitem
WHERE l_partkey = ps_partkey
GROUP BY 1;
```

Without Partial GROUP BY enabled, the Optimizer redistributes `lineitem` on the join column `l_partkey`, performs the join with `partsupp`, and then aggregates the result of the join. With a one terabyte database, this redistributes 6 billion rows and performs the join with `partsupp` to produce a result set of 6 billion rows on which the aggregation is performed.

With Partial GROUP BY enabled, the Optimizer first separately aggregates the relations `partsupp` and `lineitem` and then joins the results of those aggregations.

Without the transformation, approximately 85 billion rows must be read and written. This is largely because the join of the 6 billion row `lineitem` relation is joined to the 800 million row `partsupp` relation to produce a 24 billion row join result spool for aggregation.

With the transformation, the `lineitem` relation is aggregated as part of the sort and redistribution operation to produce a 200 million row spool. The `partsupp` relation is locally aggregated to produce another 200 million row spool. The two spool files are then joined to produce a 200 million row result, and there is an overall reduction of about 3 times in the number of rows read and written.
DBS Control Flags to Enable or Disable Partial GROUP BY

Implementation of Partial GROUP BY is automatic and transparent. The Optimizer chooses the new plan without user intervention. Note, however, that the optimization can be turned on or off using the internal DBS Control flag SynDiagFlags as follows:

<table>
<thead>
<tr>
<th>Flag Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Partial GROUP BY is enabled with costing.</td>
</tr>
<tr>
<td></td>
<td>This is the default.</td>
</tr>
<tr>
<td>1</td>
<td>Partial GROUP BY is enabled without costing.</td>
</tr>
<tr>
<td>2</td>
<td>Partial GROUP BY is disabled.</td>
</tr>
</tbody>
</table>

Identifying Partial GROUP BY Operations in EXPLAIN Report Text

They are two ways to coalesce rows using Partial GROUP BY:

- Partial SUM steps (see “Early GROUP BY With a Partial SUM Step” on page 355)
- Sort/Group steps (see “SORT/GROUP Step” on page 357)

Early GROUP BY With a Partial SUM Step

In this section we demonstrate the new EXPLAIN phrase partial SUM.

Consider the following query:

```sql
SELECT l_partkey, SUM(l_quantity), SUM(ps_supplycost)  
FROM partsupp, lineitem  
WHERE l_partkey = ps_partkey  
GROUP BY 1;
```

Without Partial GROUP BY enabled, the join plan would be [partsupp x lineitem]'.
With Partial GROUP BY enabled, the join plan is [partsupp] x [lineitem]'.

The following EXPLAIN text compares two explanations of this query: the first with Partial GROUP BY disabled and the second with Partial GROUP BY enabled.

Explanation with Partial GROUP BY disabled:

```
Explanation
-----------------------------------------------------------
1) First, we lock a distinct TPCD50G."pseudo table" for read on a RowHash to prevent global deadlock for TPCD50G.partsupp.  
2) Next, we lock a distinct TPCD50G."pseudo table" for read on a RowHash to prevent global deadlock for TPCD50G.lineitem.  
3) We lock TPCD50G.partsupp for read, and we lock TPCD50G.lineitem for read.  
4) We do an all-AMPS RETRIEVE step from TPCD50G.lineitem by way of an all-rows scan with no residual conditions into Spool 4 (all_amps), which is redistributed by hash code to all AMPs. Then we do a SORT to order Spool 4 by row hash. The result spool file will not be cached in memory. The size of Spool 4 is estimated with high confidence to be 300,005,811 rows. The estimated time for this step is 2 hours and 51 minutes.
```
5) We do an all-AMPS JOIN step from TPCDS50G.partsupp by way of a RowHash match scan with no residual conditions, which is joined to Spool 4 (Last Use). TPCDS50G.partsupp and Spool 4 are joined using a merge join, with a join condition of ("L_PARTKEY = TPCDS50G.partsupp.PS_PARTKEY"). The input table TPCDS50G.partsupp will not be cached in memory, but it is eligible for synchronized scanning. The result goes into Spool 3 (all_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 3 is estimated with low confidence to be 1,200,023,244 rows. The estimated time for this step is 2 hours and 46 minutes.

6) We do an all-AMPS SUM step to aggregate from Spool 3 (Last Use) by way of an all-rows scan, and the grouping identifier in field 2. Aggregate Intermediate Results are computed locally, then placed in Spool 5. The aggregate spool file will not be cached in memory. The size of Spool 5 is estimated with low confidence to be 1,200,023,244 rows. The estimated time for this step is 10 hours and 6 minutes.

7) We do an all-AMPS RETRIEVE step from Spool 5 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 1 is estimated with low confidence to be 1,200,023,244 rows. The estimated time for this step is 2 hours and 28 minutes.

8) 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 on statement 1.

Explanation with Partial GROUP BY enabled:

1) First, we lock a distinct TPCDS50G."pseudo table" for read on a RowHash to prevent global deadlock for TPCDS50G.partsupp.

2) Next, we lock a distinct TPCDS50G."pseudo table" for read on a RowHash to prevent global deadlock for TPCDS50G.lineitem.

3) We lock TPCDS50G.partsupp for read, and we lock TPCDS50G.lineitem for read.

4) We do an all-AMPS partial SUM step to aggregate from TPCDS50G.partsupp by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 6. The input table will not be cached in memory, but it is eligible for synchronized scanning. The aggregate spool file will not be cached in memory. The size of Spool 6 is estimated with low confidence to be 10,000,000 rows. The estimated time for this step is 12 minutes and 36 seconds.

5) We execute the following steps in parallel.

1) We do an all-AMPS RETRIEVE step from Spool 6 (Last Use) by way of an all-rows scan into Spool 5 (all_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 5 by row hash. The result spool file will not be cached in memory. The size of Spool 5 is estimated with no confidence to be 10,000,000 rows.

2) We do an all-AMPS partial SUM step to aggregate from TPCDS50G.lineitem by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed globally, then placed in Spool 5. The input table will not be cached in memory, but it is eligible for synchronized scanning. The aggregate spool file will not be cached in memory. The size of Spool 5 is estimated with no confidence to be 10,000,000 rows. The estimated time for this step is 3 hours and 39 minutes.

6) We do an all-AMPS RETRIEVE step from Spool 10 (Last Use) by way of an all-rows scan into Spool 9 (all_amps), which is redistributed by hash code to all AMPs. Then we do a SORT to order Spool 9 by row hash. The result spool file will not be cached in memory. The size of Spool 9 is estimated with no confidence to be 10,000,000 rows.

7) We do an all-AMPS JOIN step from Spool 5 (Last Use) by way of a RowHash match scan, which is joined to Spool 9 (Last Use). Spool 5 and Spool 9 are joined using a merge join, with a join condition of ("L_PARTKEY = PS_PARTKEY"). The result goes into Spool 3 (all_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 3 is estimated with low confidence to be 993,739,248 rows. The estimated time for this step is 2 hours and 16 minutes.

8) We do an all-AMPS RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 1 is estimated with low confidence to be 993,739,248 rows. The estimated time for this step is 2 hours and 28 minutes.

9) 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 on statement 1.

With Partial GROUP BY disabled, the SUM step is performed in step 6 where the cardinalities are reduced. In contrast, when Partial GROUP BY is enabled, the partial SUM step is performed in steps 4 and 5.1.

Because the Partial GROUP BY optimization is performed long before the join, the EXPLAIN text adds the phrase partial before the SUM step for easy identification of the optimization.
Chapter 3: Join Planning and Optimization

Partial GROUP BY Block Optimization

Note that it is possible for the Optimizer to select a join plan that does not use this optimization, even though Partial GROUP BY is enabled. This might happen, for example, if the cost of aggregation makes it not worth doing because of the number of small groups.

**SORT/GROUP Step**

This section introduces the phrase **SORT/GROUP** as an identification of GROUP BY. The following query has a nested subquery, with the inner query specifying an OUTER JOIN. Both the inner and outer queries have GROUP BY clauses.

```
SELECT c_count, COUNT(*) AS custdist
FROM (SELECT c_custkey, COUNT(o_orderkey)
     FROM customer LEFT OUTER JOIN ordertbl
     ON c_custkey = o_custkey
     AND o_comment NOT LIKE '%special%requests%
     GROUP BY c_custkey) AS c_orders (c_custkey, c_count)
GROUP BY c_count
ORDER BY custdist desc, c_count DESC;
```

**Explanation with Partial GROUP BY enabled:**

```
1) First, we lock a distinct TPCD50G."pseudo table" for read on a RowHash to prevent global deadlock for TPCD50G.CUSTOMER.
2) Next, we lock a distinct TPCD50G."pseudo table" for read on a RowHash to prevent global deadlock for TPCD50G.ORDERTBL.
3) We lock TPCD50G.CUSTOMER for read, and we lock TPCD50G.ORDERTBL for read.
4) We do an all-AMPs RETRIEVE step from TPCD50G.ORDERTBL by way of an all-rows scan with a condition of ("NOT(TPCD50G.ORDERTBL.O_COMMENT LIKE '%special%requests%')") into Spool 5 (all_amps), which is redistributed by hash code to all AMPs. Then we do a SORT/GROUP to order Spool 5 by row hash and non-aggregate fields grouping duplicate rows. The result spool file will not be cached in memory. The size of Spool 5 is estimated with no confidence to be 67,500,000 rows.
5) We do an all-AMPs JOIN step from TPCD50G.CUSTOMER by way of a RowHash match scan with no residual conditions, which is joined to Spool 5 (Last Use). TPCD50G.CUSTOMER and Spool 5 are left outer joined using a merge join, with a join condition of ("TPCD50G.CUSTOMER.C_CUSTKEY = O_CUSTKEY"). The input table TPCD50G.CUSTOMER will not be cached in memory. The result goes into Spool 3 (all_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 3 is estimated with low confidence to be 74,954,952 rows. The estimated time for this step is 15 minutes and 32 seconds.
6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (all_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 1 is estimated with low confidence to be 74,954,952 rows. The estimated time for this step is 11 minutes and 6 seconds.
7) We do a SUM step to aggregate from Spool 1 (Last Use) by way of an all-rows scan, and the grouping identifier in field 3. Aggregate Intermediate Results are computed globally, then placed in Spool 10. The size of Spool 10 is estimated with no confidence to be 8,658 rows. The estimated time for this step is 1 minute and 26 seconds.
8) We do an all-AMPs RETRIEVE step from Spool 10 (Last Use) by way of an all-rows scan into Spool 8 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 8 by the sort key in spool field1. The size of Spool 8 is estimated with no confidence to be 8,658 rows. The estimated time for this step is 0.25 seconds. Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request. -> The contents of Spool 8 are sent back to the user as the result of statement 1.
```

In this explanation, the phrase **SORT/GROUP** appears in Step 4. That means the Partial GROUP BY is used early to reduce cardinalities.

When the system executes this query using the EXPLAINed plan, but with Partial GROUP BY disabled, the ratio of **Partial GROUP BY Off** to **Partial GROUP By On** is 100.
Eliminating the Last Aggregation for Better Performance

Partial GROUP BY does aggregations concurrently with joins, and GROUP BY operations might be performed several times. For several reasons, it is helpful for the Optimizer to know when these aggregations are terminated. The fundamental question is this: under what conditions can the last aggregation be eliminated?

The following are examples of join plans that do not require a final GROUP BY operation:

Assume the following query is submitted to the database for the TPC-H benchmark (see http://www.tpc.org/tpch/spec/tpch2.6.0.pdf for a complete definition of the relations used for this benchmark).

```sql
SELECT l_suppkey, SUM(l_quantity), SUM(ps_availqty)
FROM lineitem, partsupp
WHERE l_suppkey = ps_suppkey
GROUP BY 1;
```

This query uses the join plan `[lineitem]`' x `[partsupp]''.

Consider the following request:

```sql
SELECT b1, c1, SUM(t1.float_1), SUM(t2.float_2), SUM(t3.float_3)
FROM t1, t2, t3
WHERE b1 = b2
AND b2 = b3
GROUP BY 1, 2;
```

Without Partial GROUP BY, this query would use the join plan `[[t2xt3]' x t1]'', with a final GROUP BY operation. Because the columns cover join columns, the last GROUP BY can be skipped, and the join plan can be optimized to `[t2xt3]' x t1''.

The last GROUP BY operation need not be performed for the following request:

```sql
SELECT c_custkey, c_name,
       SUM(l_extendedprice*(1-l_discount)(FLOAT))(DECIMAL(18,2)) AS revenue, c_acctbal, n_name, c_address, c_phone, c_comment
FROM customer, ordertbl, lineitem, nation
WHERE c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate >= '1993-10-01'
AND o_orderdate < DATE '1993-10-01' + INTERVAL '3' MONTH
AND l_returnflag = 'R'
AND c_nationkey = n_nationkey
GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment
ORDER BY revenue DESC;
```

The time savings gained by eliminating this step is 15 percent. This query might have two possible join plans depending on the system configuration. Neither plan requires the last GROUP BY operation. The two join plans are as follows:

- `((lineitem x ordertbl)' x customer) x nation`
- `((ordertbl x customer) x lineitem)' x nation`
Suppose you submit the following request against the TPC-H performance benchmark database:

```sql
SELECT c_custkey, c_name,
       SUM(l_extendedprice*(1-l_discount)(FLOAT)) (DECIMAL(18,2)) AS revenue,
       c_acctbal, n_name, c_address, c_phone, c_comment
FROM customer, ordertbl, lineitem, nation
WHERE c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate >= '1993-10-01'
AND o_orderdate < DATE '1993-10-01' + INTERVAL '3' MONTH
AND l_returnflag = 'R'
AND c_nationkey = n_nationkey
GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address,
         c_comment
ORDER BY revenue DESC;
```

When the definition of the `nation` table is changed to define the primary index on `n_key` rather than `n_name`, the join plan without Partial GROUP BY enabled would include the last GROUP BY as follows: `(((lineitem x ordertbl)' x customer) x nation)'`. Note the 'GROUP BY operator at the end of the join plan specification.

The Partial GROUP BY optimization compensates for this change in the primary index definition by removing the last GROUP BY, resulting in the plan `((lineitem x ordertbl)' x customer) x nation`.

### Performance Analysis

To demonstrate the advantage of Partial GROUP BY, consider the following experiment using one of the queries from the TPC-H performance benchmark that was run against the `lineitem` table from the standard database schema for that benchmark (see http://www.tpc.org/tpch/spec/tpch2.6.0.pdf). The variables in the experiment were the number of duplicate grouping column values and whether Partial GROUP BY was enabled or not. The results of this experiment are as follows:

<table>
<thead>
<tr>
<th>Number of Grouping Column Duplicates</th>
<th>Execution Time With Partial GROUP BY On (seconds)</th>
<th>Execution Time With Partial GROUP BY Off (seconds)</th>
<th>Percent Difference in Execution Time With PGB Enabled or Disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>10</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>26.09</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>15</td>
<td>40.00</td>
</tr>
<tr>
<td>25</td>
<td>10</td>
<td>17</td>
<td>51.85</td>
</tr>
<tr>
<td>80</td>
<td>6</td>
<td>35</td>
<td>141.46</td>
</tr>
</tbody>
</table>

It is fairly obvious from these measurements that as the number of grouping column duplicates increases, the Partial GROUP BY Optimization eliminates significantly more rows than when Partial GROUP BY is not enabled, resulting in significant reduction in execution times.
Collecting Statistics for Partial GROUP BY

For the Optimizer to know when to apply GROUP BY operators effectively, you should collect statistics on all join and GROUP BY columns in your typical requests.

For the following query, for example, you should collect statistics on `l_orderkey`, `o_orderkey`, `o_orderdate`, `c_custkey`, `c_nationkey`, and `n_nationkey`.

```sql
SELECT c_custkey, c_name, SUM(l_extendedprice*(1-l_discount)(FLOAT))(DECIMAL(18,2)) AS revenue, c_acctbal, n_name, c_address, c_phone, c_comment
FROM customer, ordertbl, lineitem, nation
WHERE c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate >= '1993-10-01'
AND o_orderdate < DATE '1993-10-01' + INTERVAL '3' MONTH
AND l_returnflag = 'R'
AND c_nationkey = n_nationkey
GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment
ORDER BY revenue DESC;
```

Using another example request, you should collect statistics on `l_partkey` and `ps_partkey` for the following query:

```sql
SELECT l_partkey, SUM(l_quantity), SUM(ps_supplycost)
FROM partsupp, lineitem
WHERE l_partkey = ps_partkey
GROUP BY 1;
```
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 SQL Data Definition Language.

- To obtain an accurate description of the processing that will be performed for a particular join, always submit an EXPLAIN request modifier 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 request modifier, see SQL Data Manipulation Language.

  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:

<table>
<thead>
<tr>
<th>Join Method</th>
<th>Important Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>• Always selected by the Optimizer for WHERE clause inequality conditions.</td>
</tr>
<tr>
<td></td>
<td>• High cost because of the number of comparisons required.</td>
</tr>
<tr>
<td>Merge or Hash</td>
<td>• When done on matching primary indexes, do not require any data to be redistributed.</td>
</tr>
<tr>
<td></td>
<td>• Hash joins are often better performers and are used whenever possible.</td>
</tr>
<tr>
<td></td>
<td>• They can be used for equijoins only.</td>
</tr>
<tr>
<td>Nested</td>
<td>• Only join expression that generally does not require all AMPs.</td>
</tr>
<tr>
<td></td>
<td>• Preferred join expression for OLTP applications.</td>
</tr>
</tbody>
</table>
Strategies for Joining PPI Relations With NoPI Relations

The individual join methods each have their own specific processes for joining PPI relations with NoPI relations. This topic provides a very general, high-level look at how such joins are made.

NoPI relation joins with PPI relations fall roughly into the groups in the bulleted list at the bottom of this topic. If the NoPI relation is spooled, it is treated exactly like a primary-indexed relation that is spooled, so the regular non-NoPI join strategies apply.

The general strategy for dealing with joining a NoPI relation with a spooled PPI relation is to simply spool the relation and then apply an appropriate Product or Merge Join method.

The specific explanations of the various join methods that apply to joining PPI relations with NoPI relations are described in the following topics:

- “Product Join When One of the Relations Is a NoPI Table” on page 364
- “Product Join With Dynamic Partition Elimination” on page 366
- “Merge Join Between a Primary-Indexed Table and a NoPI Table” on page 374
- “Supported Join Methods for Rowkey-Based Merge Join” on page 383
- “Sliding Window Merge Join Between a Primary-Indexed Table and a NoPI Table” on page 393
- “Dynamic Partition Elimination for a Sliding Window Merge Join Between a Primary-Indexed Table and a NoPI Table” on page 394
- “Nested Join Between a Primary-Indexed Relation and a NoPI Relation” on page 405
Product Join

Definition

The Product Join compares every qualifying row from one relation to every qualifying row from the other relation and saves the rows that match the WHERE predicate filter. Because all rows of the left relation in the join must be compared with all rows of the right relation, the system always duplicates the smaller relation on all AMPs, and if the entire spool does not fit into available memory, the system is required to read the same data blocks more than one time. Reading the same data block multiple times is a very costly operation.

This operation is called a Product Join because the number of comparisons needed is the algebraic product of the number of qualifying rows in the two relations.

Any of the following conditions can cause the Optimizer to specify a Product Join over other join methods:

- No WHERE clause is specified in the query.
- The join is on an inequality condition.
  If you specify a connecting, or bind term between the relations, then the system does not specify a Product Join. Bind terms are those that are bound together by the equality operator.
- There are ORed join conditions.
- A referenced relation is not specified in any join condition.
- The Product Join is the least costly join method available in the situation.

Depending on the estimated cost, and whether the condition on which the join is being made is an equality or an inequality, the Optimizer might substitute a form of Hash Join (see “Hash Join” on page 397 and “Dynamic Hash Join” on page 404) in place of an equality Product Join.

8. Hash Joins are used only for equality conditions.
Product Join Types

There are seven families of product join methods:

- Inner Product Join
- Left Outer Product Join
- Right Outer Product Join
- Full Outer Product Join
- Piggybacked Product Join
- Inclusion Product Join
  - Used only for a Product Join made on an IN term.
    - Inner Inclusion Product Join
    - Outer Inclusion Product Join
- Exclusion Product Join
  - Used only for a Product Join made on a NOT IN term.
    - Inner Exclusion Product Join
    - Outer Exclusion Product Join

Product Join When One of the Relations Is a NoPI Table

A Product Join is always valid for joining a primary-indexed relation with a NoPI relation, with either relation in the join being eligible for being duplicated or remaining local.

How a Product Join is Processed

The following list outlines the Product Join process:

1. Cache the left relation rows.
2. Join each row of the right relation with each row from the cached left relation.
3. End of process.

An overall join condition is a WHERE constraint that links relations 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:

![Generic Product Join Diagram](image1)

The following graphic illustrates a concrete Product Join:

![Concrete Product Join Diagram](image2)
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 any of 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 join method available in terms of system resources, and is used only when there is not a 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.

Product Join With Dynamic Partition Elimination

The system performs Dynamic Partition Elimination, or DPE, for Product Joins in the AMPs after a query has already been optimized, so in this case, the partition elimination undertaken is dependent on the actual data on disk, not a statistical estimate of that data. “Product Join Types” on page 364

The term Direct Product Join describes a join in which the table or join index of interest is not spooled in preparation for a product join, but is instead joined directly. The Optimizer might choose a Direct Product Join when all the partitioning columns of one or more partitioning expressions are specified in equality join terms.

As is the case for Static Partition Elimination (see “Static Partition Elimination” on page 300), the system applies the Product Join DPE optimization for single-level partitioning independently for each partitioning expression.

Partition elimination methods can be mixed within the same query. For example, Static Partition Elimination can be used for some partition levels, while DPE can be used for other levels, and some levels might not evoke any partition elimination. Some partition levels might even benefit from multiple forms of partition elimination.

The system combines the result of the partition elimination process to determine which internal partitions need to be accessed. Combining can occur for Static Partition Elimination as part of generating the query plan.

DPE occurs within the AMPs as row sets are processed, and is combined with any Static Partition Elimination from the query plan at that point.

The same restrictions that apply globally for single-level partitioning apply to multilevel partitioning at each level. The system does not support other forms of DPE, such as those based on conditions against subqueries.
When performing join steps, the qualifying partitions of the PPI table are determined dynamically based on the values of rows selected from the other relation in the join. Instead of a Product Join against all the rows in the PPI table, the system does a Product Join only for each set of rows of the other relation of a binary join that match with a single partition. Note that the system does not necessarily choose to use this optimization for this type of query because other join plans might have a better estimated cost depending on the demographics of the data and the form of partitioning that is used.

This extends to multilevel partitioning for cases where the partitioning column for one or more levels is in an equality condition with a column in the other relation.

For single-level partitioning, DPE for Product Joins can occur when there is an equality constraint between the partitioning column of one table and a column of another relation in the join. This is useful when the system uses a Product Join and must look up rows in one table and match those rows to rows in corresponding partitions instead of performing a Product Join to the entire table. In this case, only those partitions that are required to answer the request take part in the join.

For a Product Join with DPE, the left relation is sorted by RowKey using the same partitioning expression as the right relation.

Rows from the left relation are loaded into the cache one partition at a time. This partition is then set as the static partition for reading the right PPI table. In other words, the file system returns rows belonging to only this partition from the right table. Once the EOF marker for the right table is reached, the system reloads the left cache with the subsequent partition, and the process is repeated.

The AMP software does not always select this optimization for this type of query because other join plans might have a lower estimated cost, depending on the demographics of the data and the form of partitioning specified by the table definition.

DPE can occur when there is an equality constraint between the partitioning column of one table and a column of the other table in the join. This is useful when looking up a row in one table and matching those rows to corresponding partitions using a Product Join instead of executing a Product Join on the entire table. When partition elimination is invoked, only the partitions required to answer the request are involved in the join. While the AMP software is performing join steps, the qualifying partitions of the PPI table are dynamically determined based on the values of rows from the other table. Instead of a Product Join against all the rows in the PPI table, the system does a Product Join only for a row of the other table against a single partition.

For example, if there are 100 partitions in the PPI table and only 5 of them are needed to answer the join request, the system does not join the other 95 partitions, providing a 95 percent resource saving for the operation.

Be sure to collect statistics on the following column sets from each table being joined:

- The primary indexes
- The system-derived PARTITION columns
- The partitioning column of the PPI table
- The column in the other table being equated with the PPI table partitioning column
Teradata Database can also apply DPE to Inner Merge Joins (see “Slow Path Inner Merge Join” on page 374 and “Fast Path Inner Merge Join” on page 375) for single-level partitioning if the right relation in the join has a PPI with a direct geography. The right relation must also be partitioned by a single column with either of the following characteristics:

- A partitioning expression based on a RANGE_N function and a single range with its granularity defined by an EACH clause.
- The standalone partitioning column.

For example, assuming the partitioning column of the PPI table is \( x \), the following are all acceptable partitioning expressions:

- \( \text{PARTITION BY } x \)
- \( \text{PARTITION BY } \text{RANGE}_N(x \text{ BETWEEN 1 AND 100 EACH 5}) \)

With the same assumption for the partitioning column, the following are some unacceptable partitioning expressions:

- \( \text{PARTITION BY } (x \mod 65535) + 1 \)
- \( \text{PARTITION BY } \text{CASE}_N(x < 10, x \leq 50, \text{NO CASE OR UNKNOWN}) \)
- \( \text{PARTITION BY } \text{RANGE}_N(x \text{ BETWEEN } *, 1, 3, 6 \text{ AND } 10, 11 \text{ AND } *) \)

The join conditions must be conjunctive (ANDed) and there must be at least one binding on the right partitioning column of the right relation such as one of the following:

<table>
<thead>
<tr>
<th>Binding Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>Join on a single partition</td>
</tr>
<tr>
<td>Inequality</td>
<td>Join on an open partition range</td>
</tr>
<tr>
<td>Range</td>
<td>Join on a closed partition range</td>
</tr>
</tbody>
</table>

A bind term is one that connects two tables on an equality condition. For special partitions such as NO RANGE and UNKNOWN, the Sliding Window Join algorithm is used, and the system does not apply DPE.

The Optimizer can consider a Product Join with Dynamic Partition Elimination between a PPI table and a NoPI table when a join condition is specified on the partitioning expression of the PPI table. In this case, the NoPI table is built based on the RowKey of its join condition column, and it is duplicated across the AMPs. The Product Join is then made within the corresponding partitions for the join.

---

9. This usually means there are equality join conditions on the primary index.
Costing a Product Join With Dynamic Partition Elimination

A Product Join with dynamic partition elimination can be considered as a single partition-to-single partition Product Join. The cost model uses a reduction ratio in the amount of I/O performed for the right relation.

Following the general cost model for a Product Join, the total I/O cost of a Product Join with dynamic partition elimination is expressed by the following equation:

$$\text{Cost}_{\text{PJ with DPE}} = \text{LeftRelCost} + (\text{RightRelCost} \times \text{Reduction} \times \text{NumCache})$$

where:

<table>
<thead>
<tr>
<th>Equation element ...</th>
<th>Specifies the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeftRelCost</td>
<td>cost of reading the left relation.</td>
</tr>
<tr>
<td>RightRelCost</td>
<td>cost of reading the right relation.</td>
</tr>
<tr>
<td>Reduction</td>
<td>amount of I/O performed for the right relation, calculated as follows:</td>
</tr>
<tr>
<td></td>
<td>( \frac{\text{NumBlocks}}{\text{NonEmptyPartNum}} \cdot \frac{\text{Reduced \text{NumBlocks}}}{\text{NumBlocks}} )</td>
</tr>
</tbody>
</table>

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 relation directly. The row is copied into a spool file, which is then replicated on every AMP on which the charges relation is stored.

On each of these AMPs, the resident rows of the charges relation are searched one at a time for the qualifying value in the proj_id column.
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 relation may have to be read several times if all of the rows of a Product Join cannot be contained in the available 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 relation definitions:

<table>
<thead>
<tr>
<th>Employee</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ENum</td>
<td>EName</td>
<td>Dept</td>
</tr>
<tr>
<td>PK</td>
<td></td>
<td>FK</td>
</tr>
<tr>
<td>UPI</td>
<td>Brown</td>
<td>200</td>
</tr>
<tr>
<td>1</td>
<td>Smith</td>
<td>310</td>
</tr>
<tr>
<td>2</td>
<td>Jones</td>
<td>310</td>
</tr>
<tr>
<td>3</td>
<td>Clay</td>
<td>400</td>
</tr>
<tr>
<td>4</td>
<td>Peters</td>
<td>150</td>
</tr>
<tr>
<td>5</td>
<td>Foster</td>
<td>400</td>
</tr>
<tr>
<td>6</td>
<td>Gray</td>
<td>310</td>
</tr>
<tr>
<td>7</td>
<td>Baker</td>
<td>310</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department</th>
<th>Dept</th>
<th>DeptName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept</td>
<td>UPI</td>
<td></td>
</tr>
<tr>
<td>PK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPI</td>
<td>400</td>
<td>Education</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>Payroll</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>Finance</td>
</tr>
<tr>
<td></td>
<td>310</td>
<td>Manufacturing</td>
</tr>
</tbody>
</table>

Assume that a join request such as the following is executed against these relations:

```sql
SELECT *
FROM employee, department
WHERE employee.dept > department.dept;
```
The rows are redistributed as shown in the following graphic:

**DEPARTMENT ROWS HASH-DISTRIBUTED ON DEPARTMENT.DEPT(UPI):**

```
150 PAYROLL
310 MFG
200 FINANCE
400 EDUCATION
```

**SPOOL FILES AFTER DUPLICATING THE DEPARTMENT ROWS:**

```
310 MFG
400 EDUCATION
200 FINANCE
150 PAYROLL
310 MFG
400 EDUCATION
200 FINANCE
150 PAYROLL
310 MFG
400 EDUCATION
200 FINANCE
150 PAYROLL
310 MFG
400 EDUCATION
200 FINANCE
150 PAYROLL
```

**EMPLOYEE ROWS HASH-DISTRIBUTED ON EMPLOYEE.ENUM (UPI):**

```
6 FOSTER 400
8 BAKER 310
4 CLAY 400
3 JONES 310
1 BROWN 200
7 GRAY 310
5 PETERS 150
2 SMITH 310
```

Notice that the rows of the smaller relation, `department`, are duplicated on all AMPS.
The smaller relation is determined by multiplying the number of required column bytes by
the number of rows.


**Merge Join**

**Definition**

The 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. Merge Join is generally more efficient than a Product Join (see “Product Join” on page 363) because it requires fewer comparisons and because blocks from both tables are read only once.

Two different general Merge Join algorithms are available:

- **Slow Path** (see “Slow Path Inner Merge Join” on page 374)
  - 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** (see “Fast Path Inner Merge Join” on page 375)
  - The fast path is used when the left table is accessed using the all-row scan reading mode.

Each of these can also be applied to the eight various Merge Join methods:

- **Fast Path Inner Merge Join** (see “Fast Path Inner Merge Join” on page 375)
  - Fast Path Inner Inclusion Merge Join (see “Inclusion Merge Join” on page 424)

- **Slow Path Inner Merge Join** (see “Slow Path Inner Merge Join” on page 374)
  - Slow Path Inner Inclusion Merge Join (see “Inclusion Merge Join” on page 424)

- **Exclusion Merge Join** (see “Exclusion Merge Join” on page 419)

- **Fast Path Left Outer Join**
  - Fast Path Left Outer Inclusion Merge Join (see “Inclusion Merge Join” on page 424)

- **Slow Path Left Outer Join**
  - Slow Path Left Outer Inclusion Merge Join (see “Inclusion Merge Join” on page 424)

- **Fast Path Right Outer Merge Join**

- **Slow Path Right Outer Merge Join**

- **Full Outer Merge Join**
Another class of merge join methods is used only with PPI tables:

- Direct Merge Join (see “Direct PPI Merge Join” on page 381)
- Rowkey-based Merge Join (see “Rowkey-Based Merge Join” on page 383)
- Single Window Merge Join (see “Single Window Merge Join” on page 386)
- Sliding Window Merge Join (see “Sliding Window Merge Join” on page 389)

Note that Dynamic Partition Elimination (see “Product Join With Dynamic Partition Elimination” on page 366 and Database Design) is not supported for Merge Joins of MLPPI tables.

The Optimizer has three general approaches when joining a PPI table to an NPPI table or when joining two PPI tables with different partitioning expressions:

- Spool the PPI table (or both PPI tables) into an NPPI spool file in preparation for a traditional Merge Join.
- Spool the NPPI table (or one of the two PPI tables) into a PPI spool file, with identical partitioning to the remaining table, in preparation for a Rowkey-Based Merge Join (see “Rowkey-Based Merge Join” on page 383).
  This option is not always available.
- Use the Sliding Window Merge Join of the tables without spooling either one (see “Sliding Window Merge Join” on page 389).

In all cases, the Optimizer considers all reasonable join strategies and selects the one with the least estimated cost.

Depending on the relative costs, the Optimizer might substitute a form of Hash Join (see “Hash Join” on page 397 and “Dynamic Hash Join” on page 404) in place of a Merge Join when there is no skew in the data, depending on the relative costs of the two methods.

**Generic Merge Join Strategy**

The high level process applied by the Merge Join algorithm is the following:

1. Identify the smaller relation of the pair to be joined.
2. The Optimizer pursues the following steps only if it is necessary to place qualified rows into a spool file:
   a. Place the qualifying rows from one or both relations into a spool file.
   b. Relocate the qualified spool rows to their target AMPs based on the hash of the join column set.
   c. Sort the qualified spool rows on their join column row hash values.
3. Compare the relocated row set with matching join column row hash values in the other relation.
Merge Join Costing

To cost the possible binary combinations of merge join strategies, the following combinations of relations \( R_1 \) and \( R_2 \) are analyzed:

- \( R_1 \) as the left relation, \( R_2 \) as the right relation using the fast path method.
- \( R_1 \) as the left relation, \( R_2 \) as the right relation using the slow path method.
  
  This is only costed if there is an access path for \( R_1 \).
- \( R_2 \) as the left relation, \( R_1 \) as the right relation using the fast path method.
- \( R_2 \) as the left relation, \( R_1 \) as the right relation using the slow path method.
  
  This is only costed if there is an access path for \( R_2 \).

The Optimizer selects the least costly combination to be used for further comparisons.

Merge Join Between a Primary-Indexed Table and a NoPI Table

For most cases, the rows of a NoPI table require a change in their demographics before they can be Merge-Joined with a primary-indexed table. The only time it is possible to avoid relocating NoPI table rows across the AMPs is the case where the other table is duplicated. In all cases documented in the following table, the assumption is made that the hashes needed for making the join are preserved when the demographics for the rows change.

<table>
<thead>
<tr>
<th>Merge Join Demographics</th>
<th>Merge Join Demographics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NoPI Table Direct</td>
</tr>
<tr>
<td>PI Table Direct</td>
<td>Not valid</td>
</tr>
<tr>
<td>PI Table Local</td>
<td>Not valid</td>
</tr>
<tr>
<td>PI Table Duplicated</td>
<td>Not valid</td>
</tr>
<tr>
<td>PI Table Redistributed</td>
<td>Not valid</td>
</tr>
</tbody>
</table>

Slow Path Inner Merge Join

The process applied by the slow path Merge Join algorithm is the following:

1. Read each row from the left table.
2. Join each left table row with the right table rows having the same hash value.
3. End of process.
The following graphic illustrates the generic slow path Merge Join process:

Fast Path Inner Merge Join

The process applied by the fast path Merge Join algorithm is:

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 \( \geq \) to that of the left table row.

<table>
<thead>
<tr>
<th>IF the row hash values are ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>join the two rows.</td>
</tr>
<tr>
<td>not equal</td>
<td>use the larger row hash value to read the row from the other table.</td>
</tr>
</tbody>
</table>

3. End of process.

The following graphic illustrates the generic fast path Merge Join process:
Merge Join Distribution Strategies

Merge Joins use one of the following distribution strategies:

<table>
<thead>
<tr>
<th>Strategy Number</th>
<th>Strategy Name</th>
<th>Stage</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hash Redistribution</td>
<td>1</td>
<td>Hash redistribute one or both sides (depending on the primary indexes used in the join).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Sort the rows into join column row hash sequence.</td>
</tr>
<tr>
<td>2</td>
<td>Table Duplication</td>
<td>1</td>
<td>Duplicate the smaller side on all AMPs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Sort the rows into the row hash sequence of the join column.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Locally copy the other side and sort the rows into row hash sequence of the join column.</td>
</tr>
<tr>
<td>3</td>
<td>Index Matching</td>
<td>1</td>
<td>No redistribution is required if the primary indexes are the join columns and if they match.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FOR a representation of this strategy ...</th>
<th>See this illustration ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash redistribution</td>
<td>“Merge Join Row Distribution (Hash Redistribution)” on page 378.</td>
</tr>
<tr>
<td>Table duplication</td>
<td>“Merge Join Row Distribution (Duplicate Table)” on page 379.</td>
</tr>
<tr>
<td>Index matching</td>
<td>“Merge Join Row Distribution (Matching Indexes)” on page 380.</td>
</tr>
</tbody>
</table>

Example 1

The following SELECT request determines who works in what location:

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

The following list presents the stages of processing this Merge Join:

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 are 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.

8. End of process.

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:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>DeptName</td>
</tr>
<tr>
<td>PK</td>
<td>PK</td>
</tr>
<tr>
<td>UPI</td>
<td>UPI</td>
</tr>
<tr>
<td>1</td>
<td>Brown</td>
</tr>
<tr>
<td>2</td>
<td>Smith</td>
</tr>
<tr>
<td>3</td>
<td>Jones</td>
</tr>
<tr>
<td>4</td>
<td>Clay</td>
</tr>
<tr>
<td>5</td>
<td>Peters</td>
</tr>
<tr>
<td>6</td>
<td>Foster</td>
</tr>
<tr>
<td>7</td>
<td>Gray</td>
</tr>
<tr>
<td>8</td>
<td>Baker</td>
</tr>
</tbody>
</table>

One of the Merge Join row redistribution methods (see “Merge Join Row Distribution (Hash Redistribution)” on page 378 or “Merge Join Row Distribution (Duplicate Table)” on page 379) is used if you perform the following SELECT request against these tables:

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

Merge Join Row Distribution (Hash Redistribution)

The following graphic shows a Merge Join row distribution using hash redistribution:

- EMPLOYEE ROWS HASH-DISTRIBUTED ON EMPLOYEE.ENUM (UPI):
  - 6 FOSTER 400
  - 8 BAKER 310
  - 4 CLAY 400
  - 3 JONES 310
  - 1 BROWN 200
  - 7 GRAY 310
  - 5 PETERS 150
  - 2 SMITH 310

- SPOOL FILE AFTER REDISTRIBUTION ON EMPLOYEE.DEPT ROW HASH:
  - 5 PETERS 150
  - 7 GRAY 310
  - 3 JONES 310
  - 8 BAKER 310
  - 2 SMITH 310
  - 1 BROWN 200
  - 6 FOSTER 400
  - 4 CLAY 400

- DEPARTMENT ROWS HASH-DISTRIBUTED ON DEPARTMENT.DEPT (UPI):
  - 150 PAYROLL
  - 310 MFG
  - 200 FINANCE
  - 400 EDUCATION

FF04A005
**Merge Join Row Distribution (Duplicate Table)**

The following graphic shows a Merge Join row distribution by duplicating a table:

**Example**

This example uses the following `Employee` and `Employee_phone` table definitions.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Employee_Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>Ename</td>
</tr>
<tr>
<td>PK</td>
<td>Brown</td>
</tr>
<tr>
<td>UPI</td>
<td>Smith</td>
</tr>
<tr>
<td></td>
<td>Jones</td>
</tr>
<tr>
<td></td>
<td>Clay</td>
</tr>
<tr>
<td></td>
<td>Peters</td>
</tr>
<tr>
<td></td>
<td>Foster</td>
</tr>
<tr>
<td></td>
<td>Gray</td>
</tr>
<tr>
<td></td>
<td>Baker</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The matching indexes row distribution strategy shown in “Merge Join Row Distribution (Matching Indexes)” on page 380 is used if you perform the following SELECT statement against these tables:

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

### Merge Join Row Distribution (Matching Indexes)

The following graphic shows a Merge Join row distribution by matching indexes:

**EMPLOYEE ROWS HASH-DISTRIBUTED ON ENUM (UPI):**

- 6 FOSTER 400
- 8 BAKER 310
- 4 CLAY 400
- 3 JONES 310
- 1 BROWN 200
- 7 GRAY 310
- 5 PETERS 150
- 2 SMITH 310

**EMPLOYEE_PHONE ROWS HASH-DISTRIBUTED ON ENUM (NUPI):**

- 6 203 8373461
- 8 301 2641616
- 8 301 6675888
- 4 415 6347180
- 3 408 3628822
- 1 213 3241576
- 1 213 4950703
- 5 312 7463513

FF04A007
Direct PPI Merge Join

Introduction

The term Direct Merge Join describes a join method in which the table or join index of interest is not spooled in preparation for a merge join, but instead is done directly. The Optimizer might choose a Direct Merge Join when at minimum all columns of the primary index are specified in equality join terms.

To qualify for a Direct PPI Merge Join, there must be equality conditions on all the primary index columns of the two relations. There are several forms of this optimization. The particular form selected by the Optimizer depends on factors such as the following:

- Any additional conditions in the query
- The total number of partitions
- The number of populated partitions

Example: PPI-To-PPI Direct Merge Join

In the following example, the system can do a Direct Merge Join of markets and market_penetration instead of redistributing both tables to spool, sorting the spool in hash order of the primary index, and then doing a Row Hash Merge Join.

First consider the table definitions used for this example:

```
CREATE TABLE markets (
    productid INTEGER NOT NULL,
    region BYTEINT NOT NULL,
    activity_date DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    revenue_code BYTEINT NOT NULL,
    business_sector BYTEINT NOT NULL,
    note VARCHAR(256))
PRIMARY INDEX (productid, region)
PARTITION BY (
    RANGE N(region BETWEEN 1 AND 9 EACH 3),
    RANGE N(business_sector BETWEEN 0 AND 49 EACH 10),
    RANGE N(revenue_code BETWEEN 1 AND 34 EACH 2),
    RANGE N(activity_date BETWEEN DATE '1986-01-01' AND DATE '2007-05-31' EACH INTERVAL '1 MONTH'));
```
CREATE TABLE market_penetration (  
  productid INTEGER NOT NULL,  
  region BYTEINT NOT NULL,  
  activity_date DATE FORMAT 'yyyy-mm-dd' NOT NULL,  
  revenue_code BYTEINT NOT NULL,  
  business_sector BYTEINT NOT NULL,  
  saturation FLOAT)  
PRIMARY INDEX (productid, region)  
PARTITION BY (  
  RANGE_N(region BETWEEN 1  
      AND 9  
      EACH 3),  
  RANGE_N(business_sector BETWEEN 0  
      AND 49  
      EACH 10),  
  RANGE_N(revenue_code BETWEEN 1  
      AND 34  
      EACH 2),  
  RANGE_N(activity_date BETWEEN DATE '1986-01-01'  
      AND DATE '2007-05-31'  
      EACH INTERVAL '1' MONTH));

The following query joins markets and market_penetration. Because of the specified conditions, the Optimizer is able to select a Direct PPI Merge Join to join the relations.

SELECT a.*, b.saturation  
FROM markets AS a, market_penetration AS b  
WHERE a.productid = b.productid  
AND a.region = b.region  
AND a.business_sector = b.business_sector  
AND a.revenue_code = b.revenue_code  
AND a.activity_code = b.activity_code;
Rowkey-Based Merge Join

Introduction

As for single-level partitioning, a Rowkey-Based Merge Join for multilevel partitioning requires equality conditions on all the primary index columns and partitioning columns of the two relations.

To be eligible for a Rowkey-Based Merge Join, both relations must also have the same partitioning. Otherwise, one of the relations must be spooled and partitioned to impose equal partitioning between the two.

Supported Join Methods for Rowkey-Based Merge Join

The following join types are supported for a Rowkey-Based Merge Join, where fast path designates row hash match scan within left and right matching partitions and slow path designates index access on one relation followed by a lookup in the other table for all rows with matching row hashes within the corresponding partition:

- Fast path Inner, Left and Full Outer Merge Join.
- Slow path Inner and Left Outer Merge Join.
- Fast path Inner and Outer Inclusion/Exclusion Merge Join.
- Slow path Inner and Outer Inclusion/Exclusion Merge Join.
- Fast path Correlated Inclusion/Exclusion Merge Join.
- Slow path Correlated Inclusion/Exclusion Merge Join.

Note that this type of join can only occur between a table and a spool, and not directly between two tables.

Right Outer Merge Join is not supported for a Rowkey-Based Merge Join. Note that the Optimizer can switch the relations so that the join type is a Left Outer Merge Join.

If the internal partition mapping is not the same for both relations, a Rowkey-Based Merge Join is not eligible for synchronized scanning. If the internal partition mapping is the same for both relations, a Rowkey-Based Merge Join is eligible for synchronized scanning only for the following join types:

- Slow path Left Outer Merge Join.
- Slow path Inner Merge Join.

A Rowkey-Based Merge Join can be considered for a join between a primary-indexed table and a NoPI table. To make such a join, the system builds the Rowkey for the NoPI table in spool based on the partitioning expression of the PPI table on the other side of the join.
Example: PPI-To-Spool Join

In the following example, the system can select rows from the dimension tables and join the intermediate results set to form a spool that is sorted by a rowkey matching that of the PPI table. Then the system can do a RowKey Merge Join from the PPI table to the spool.

This method replaces having to redistribute the spool and then sorting it on the join columns, with the other tables joined and spool also sorted on the join columns, and then joining the two spools using a Rowhash Merge Join or some other join plan. Ultimately, the Optimizer selects the plan that it estimates to be the least costly of its available options.

Definition DDL statement text for the `markets` table:

```sql
CREATE TABLE markets (
    productid INTEGER NOT NULL,
    region BYTEINT NOT NULL,
    activity_date DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    revenue_code BYTEINT NOT NULL,
    business_sector BYTEINT NOT NULL,
    note VARCHAR(256)
) PRIMARY INDEX (productid, region)
PARTITION BY (
    RANGE_N(region) BETWEEN 1 AND 9 EACH 3),
    RANGE_N(business_sector) BETWEEN 0 AND 49 EACH 10),
    RANGE_N(revenue_code) BETWEEN 1 AND 34 EACH 2),
    RANGE_N(activity_date) BETWEEN DATE '1986-01-01' AND DATE '2007-05-31' EACH INTERVAL '1' MONTH)
);
```

Definition DDL statement text for the `products` table:

```sql
CREATE TABLE products (
    productid INTEGER NOT NULL,
    product_name CHARACTER(30),
    description VARCHAR(256)
) PRIMARY INDEX (productid);
```

Definition DDL statement text for the `regions` table:

```sql
CREATE TABLE regions (
    region INTEGER NOT NULL,
    region_name CHARACTER(30),
    description VARCHAR(256)
) PRIMARY INDEX (region_name);
```

Definition DDL statement text for the `business_sectors` table:

```sql
CREATE TABLE business_sectors (
    productid INTEGER NOT NULL,
    business_sector_name CHARACTER(30),
    description VARCHAR(256)
) PRIMARY INDEX (business_sector_name);
```
Definition DDL statement text for the *revenue_codes* table:

```sql
CREATE TABLE revenue_codes (
    revenue_code INTEGER NOT NULL,
    revenue_code_name CHARACTER(30),
    description VARCHAR(256)
) PRIMARY INDEX (revenue_code_name);
```

Definition DDL statement text for the *activity_calendar* table:

```sql
CREATE TABLE activity_calendar (
    quarter CHARACTER(6),
    activity_date DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    description VARCHAR(256)
) PRIMARY INDEX (quarter);
```

Example query that joins all these tables:

```sql
SELECT p.product_name, r.region_name, b.business_sector_name,
       rc.revenue_code_name, a.quarter, m.activity_date, m.note
FROM markets AS m, products AS p, regions AS r,
     business_sectors AS b, revenue_codes AS rc,
     activity_calendar AS a
WHERE p.productid BETWEEN 4000 AND 4999
    AND r.region_name = 'West'
    AND b.business_sector_name IN ('Cosmetics', 'Snack Food', 'Hats')
    AND rc.revenue_code_name IN ('Web', 'Catalog')
    AND a.quarter IN ('2006Q1', '2005Q1')
    AND m.productid = p.productid
    AND m.region = r.region
    AND m.business_sector = b.business_sector
    AND m.revenue_code = rc.revenue_code
    AND m.activity_code = a.activity_code;
```
Single Window Merge Join

Introduction

A Single Window Merge Join is very similar to a Sliding Window Merge Join (see “Sliding Window Merge Join” on page 389) except that the number of populated partitions after static or delayed partition elimination has been applied for each table in the join is small enough to be able to handle all the partitions at once. This could be either a join of a PPI table to a PPI table or a PPI table to a spool.

The two relations must be joined with equality conditions on the primary index. There must be a limited number of participating partitions. The number of participating partitions is estimated based on the estimated number of populated partitions after any Static Partition Elimination. The maximum number of combined partitions processed in as a single set is determined in the same way as for single-level partitioning. This calculation is based on the setting of the DBS Control flag PPICacheThrP.

The Optimizer estimates whether the set will be small enough; if the estimate differs from the actual data, a Sliding Window Merge Join might or might not be used. For example, the estimate might indicate a Single Window Merge Join is the best plan but a Sliding Window Merge Join may be done if many populated partitions exist after Static or Delayed Partition Elimination has been applied. The Optimizer might also estimate that the cost of a Single Window Merge Join would exceed that of some other join method, and so would choose to use that method instead.

Example: Single Window Merge Join

In the following example, the system joins two tables on their primary indexes.

The WHERE clause conditions eliminate all but two partitions for level 1, and three partitions for level 2 of orders.

The WHERE clause conditions also eliminate all but one partition for level 1, and seven partitions for level 2 of lineitem.

After the system applies these conditions, it must join six partitions of the combined expression for orders to seven partitions of the combined partitioning expression of lineitem, making a total of thirteen partitions.

Because of this situation, the system is able to join the two tables using a Direct Merge Join, assuming the Optimizer estimates it to be the most cost effective join method. When joining the two sets of partitions, the Direct Merge Join operation handles the row sets logically as if they are in hash order by maintaining a memory-resident data block for each populated partition.
The definition DDL statement text for the two tables to be joined in this example query is as follows:

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

CREATE TABLE lineitem (
    l_orderkey INTEGER NOT NULL,
    l_partkey INTEGER NOT NULL,
    l_suppkey INTEGER,
    l_linenumber INTEGER,
    l_quantity INTEGER NOT NULL,
    l_extendedprice DECIMAL(13,2) NOT NULL,
    l_discount DECIMAL(13,2),
    l_tax DECIMAL(13,2),
    l_returnflag CHARACTER(1),
    l_linestatus CHARACTER(1),
    l_shipdate DATE FORMAT 'yyyy-mm-dd',
    l_commitdate DATE FORMAT 'yyyy-mm-dd',
    l_receiptdate DATE FORMAT 'yyyy-mm-dd',
    l_shipinstruct VARCHAR(25),
    l_shipmode VARCHAR(10),
    l_comment VARCHAR(44)
) PRIMARY INDEX (l_orderkey)
PARTITION BY ( 
    RANGE_N(l_suppkey BETWEEN 0 AND 4999 EACH 10),
    RANGE_N(l_shipdate BETWEEN DATE '2000-01-01' AND DATE '2006-12-31' EACH INTERVAL '1' MONTH)
);
```
The system selects a Single Window Merge Join to join the \textit{orders} and \textit{lineitem} tables when creating a join plan for the following query:

\begin{verbatim}
SELECT *
FROM orders, lineitem
WHERE o_orderkey = l_orderkey
    AND o_orderdate BETWEEN DATE '2005-04-01'
    AND DATE '2005-06-30'
    AND o_custkey IN (618, 973)
    AND l_shipdate BETWEEN DATE '2005-04-01'
    AND DATE '2005-10-31'
    AND l_suppkey = 4131;
\end{verbatim}
Sliding Window Merge Join

Introduction

A Direct Merge Join cannot be used when one table is partitioned and the other is not, or when both tables are partitioned, but not partitioned identically, because the rows of the two tables are not ordered in the same way. The Sliding Window Merge Join, which is PPI-aware, can be invoked by the Optimizer to cover situations that the standard Direct Merge Join cannot. Sliding window joins can be slower than a Merge Join or Rowkey-Based Merge Join when there are many noneliminated combined partitions; however, a Sliding Window Merge Join can provide roughly similar elapsed time performance (but with greater CPU utilization and memory consumption) when the number of noneliminated combined partitions is small.

Sliding Window Merge Joins follow the general principle of structuring each of the left and right relations into windows of appropriate sizes. The join is done as a Product Join between each of these left and right window pairs. The operation uses the identical algorithm to that used for a regular Merge Join within each window pair with the exception that the rows are not necessarily in row hash order across the multiple partitions within a window.

The obvious way to join an NPPI table to a PPI table would be to make one pass over the NPPI table for each noneliminated partition of the PPI table, executing the join as a series of subjoins. It turns out that this is an inefficient way to make the join, especially for a large NPPI table. To escape the inefficiency of this method, the Sliding Window Merge Join uses a similar concept, but minimizes the number of disk reads.

1. The system reads the first data block for the NPPI table, and then reads the first data block of each noneliminated combined partition of the PPI table into memory.

2. The rows from the NPPI data block are compared to the rows of each PPI data block. The join routines present the rows in row hash order, but you can think of the process as visiting the data block for each combined partition in turn.

3. As the rows of a data block are exhausted, the system reads the next data block for that combined partition.

   This results in each data block of each table being read only once.11 There is some additional overhead to manage the pool of data blocks, but join performance is not badly degraded when the window covers all noneliminated combined partitions.

4. End of process.

If a nontrivial fraction of the combined partitions can be eliminated because of query conditions, overall performance can be improved, perhaps dramatically, over a traditional Merge Join, depending on the percentage of combined partitions that can be eliminated.

10. Appropriate in terms of the number of partitions required as determined by the Optimizer based on the available memory for making the join.

11. This is because of partitioning. Merge Joins usually have to read some number of rows in one of the tables multiple times, either from disk or from cache.
A limiting factor for the Sliding Window Merge Join algorithm is the number of data blocks that can be contained in memory at the same time. The file system cache memory provides memory for the data blocks. The DBS Control performance flag PPICachThrP controls memory usage for this purpose. The value is a number expressed in tenths of a percent of the file system cache available for each query step that needs a sliding window.

The default value is 10, which represents one percent. A higher value may improve multipartition operations, of which joins are the most obvious example, if there are a large number of noneliminated combined partitions to be processed. A higher value can also result in other applications having less memory available to them, resulting in fewer cache hits, so higher values for PPICachThrP are not necessarily better for overall system performance. See Utilities and Database Design for further details.

A significant degradation of join performance can occur when there are more noneliminated combined partitions than data block buffers. Assume enough memory has been allocated for 20 data blocks and a table with 100 partitions. In this case, the sliding window method is appropriate. The first 20 combined partitions are processed as they should be, and then the system reads the NPPI table again as the join window slides down to combined partitions 21 through 40. A total of five passes through the NPPI table are required, and, assuming the NPPI table is roughly the same size as the PPI table, the join can conceptually take five times longer than a join for which the window covers the entire table. The actual performance degradation is not as bad as a factor of five, because the output spool file has exactly the same number of rows in either case, and each smaller window is more sparse with respect to the NPPI table than a larger window would be. There can be an offsetting performance gain from cache usage by the NPPI table, especially when it is fairly small.

An even more expensive situation occurs in the following cases:

- Both tables are partitioned, but have different partitioning expressions
- There are no join conditions specified on the partitioning columns

In both cases, there can be a sliding window advancing through both tables. The EXPLAIN text for such a query neither indicates that a sliding window join is being used, nor indicates the number of contexts used for each table.

The number of logical disk reads required to process such joins are expressed by the following equations:

\[
\text{LDR}_{\text{mp}} = d_1 + d_2
\]
\[
\text{LDR}_{\text{tp}} = \left( \frac{p_2}{k_2} \times d_1 \right) + d_2
\]
\[
\text{LDR}_{\text{mp}} = \left( \frac{p_2}{k_2} \times d_1 \right) + \left( \frac{p_1}{k_1} \times d_2 \right)
\]
where:

<table>
<thead>
<tr>
<th>Equation element</th>
<th>Specifies the number of</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR_{nip}</td>
<td>logical data reads when neither table is partitioned.</td>
</tr>
<tr>
<td>LDR_{t2p}</td>
<td>logical data reads when the second table is partitioned.</td>
</tr>
<tr>
<td>LDR_{btp}</td>
<td>logical data reads when both tables are partitioned.</td>
</tr>
<tr>
<td>d_{1}</td>
<td>data blocks for the first table.</td>
</tr>
<tr>
<td>d_{2}</td>
<td>data blocks for the second table.</td>
</tr>
<tr>
<td>p_{1}</td>
<td>noneliminated combined partitions in the first table.</td>
</tr>
<tr>
<td>p_{2}</td>
<td>noneliminated combined partitions in the second table.</td>
</tr>
<tr>
<td>k_{1}</td>
<td>data blocks that can be contained in memory for the first table.</td>
</tr>
<tr>
<td>k_{2}</td>
<td>data blocks that can be contained in memory for the second table.</td>
</tr>
</tbody>
</table>

If $\frac{p_{1}}{k_{1}} > 1$, some available memory is not needed, and the system substitutes a value of 1 for $\frac{p_{1}}{k_{1}}$. For one partitioned table, $\frac{p_{1}}{k_{1}}$ must be small (at most 4) for performance to be attractive, unless $p_{1}$ is small compared to the total number of partitions in the table. Without addressing every possible join costing consideration, reading the NPPI table twice and reading 40 percent of the PPI table after partition elimination might be a less expensive operation than reading each table once, as is done in the traditional Merge Join situation.

For two partitioned tables, $\frac{p_{1}}{k_{2}}$ and $\frac{p_{2}}{k_{2}}$ must both be small, with a maximum value of 2 or 3, for performance to be attractive, unless at least one of $p_{1}$ and $p_{2}$ is small compared to the total number of combined partitions in the table.

Given the importance of the term $\frac{p}{k}$ in the preceding formulas, it is important that the Optimizer have a realistic estimate of the number of noneliminated combined partitions. This is why the earlier examples that partitioned on `division_number` indicated that the RANGE_N example was better than the example that used the column directly. In the RANGE_N example, the Optimizer knows that there are a maximum of four partitions with rows, and so uses 4 or less as the value of $p$. When the column is used directly, the maximum number of partitions is 65,535, and the Optimizer might use a value much larger than 4 when estimating $p$, especially if statistics have not been collected on the `division_number` column.
The PPICacheThrP setting directly influences the $k_1$ and $k_2$ values in the preceding formulas. Each table which needs a sliding window is allocated enough memory for the duration of the step to hold some number of data blocks.\footnote{The estimated average data block size is used in this calculation.}

<table>
<thead>
<tr>
<th>This allocation, in units of data blocks ...</th>
<th>Is this value ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimum</td>
<td>8 or less, to match the number of noneliminated combined partitions, even if PPICacheThrP is set to 0.</td>
</tr>
<tr>
<td>maximum</td>
<td>the smallest of the following:</td>
</tr>
<tr>
<td></td>
<td>• 256</td>
</tr>
<tr>
<td></td>
<td>• the largest number which does not exceed the percentage specified in the PPICacheThrP setting</td>
</tr>
<tr>
<td></td>
<td>• the number of noneliminated combined partitions</td>
</tr>
<tr>
<td></td>
<td>The maximum allocation can never be less than the minimum allocation.</td>
</tr>
</tbody>
</table>

A larger number in PPICacheThrP allocates relatively more memory, when needed, to PPI windowing steps, at the expense of other steps that might use cache for other purposes.

The default PPICacheThrP setting is low enough that steps unrelated to PPI windowing are not likely to be short on cache. In theory, there could be a maximum of 50 concurrent sliding window joins on a given AMP, each operating on tables with many combined partitions, which could consume up to about 50 percent of the cache memory allocated for windowing. Even in that worst case scenario, the few NPPI steps running on the AMP would have the remaining half of the cache.

If you consider it likely that there will never be more than 10 concurrent PPI windowing operations, the default PPICacheThrP setting allows up to 10 percent of the file system cache to be used for PPI windowing steps, leaving at least 90 percent for other uses. With this assumption, it would be reasonable to consider a setting higher than the default, especially if PPI steps are considered to have fairly high priority. In a situation in which there is already heavy memory paging, it would be reasonable to consider a setting for PPICacheThrP that is lower than the default, especially if the PPI steps are considered to have a fairly low priority.

As is true for single-level partitioning, a direct PPI Merge Join under multilevel partitioning requires equality conditions on all the primary index columns of the two relations.

A Sliding Window Merge Join processes a set of populated partitions against each of multiple sets of populated partitions in the other relation. This stage of the process is followed by processing the next set of populated partitions against each of multiple sets of populated partitions in the other relation. This process is repeated until all sets of populated partitions have been joined with the other relation.
When joining two sets together, a Sliding Window Merge Join operation handles the row sets logically as if they are in hash order. Only those partitions retained after the partition elimination process are considered for join processing, and the number of combined partitions processed is determined in the same manner as is used for the single-partition case. The calculation is based on the setting of the PPICacheThrP flag in the DBS Control record for your system (see Utilities, Database Design, or SQL Data Definition Language for details).

The sliding windows are based on the partitions defined by the combined partitioning expression. For a Sliding Window Merge Join to be cost effective, only a limited number of populated partitions can participate in the join; however, multilevel partitioning usually defines many partitions for the combined partitioning expressions. This means that the Optimizer does not often have the option of choosing to use a Sliding Window Merge Join in a multilevel partitioning situation.

The final cost of a PPI Sliding Window Join is the Merge Join cost of a window pair multiplied by the number of window pairs involved in the join. The number of window pairs involved is a function of the number of partitions in the PPI relation set and the window size.

The Optimizer always estimates the cost of the join plans it evaluates to process a query; however, if an estimate differs from the actual data on the AMPs as it is revealed during processing, the system might then choose to substitute a Sliding Window Merge Join. Alternatively, the query plan developed by the Optimizer might specify a Sliding Window Merge Join, but in the final result, the AMP database software might instead dynamically reoptimize the request and use a Single Window Merge Join if there are few populated partitions (see “Product Join With Dynamic Partition Elimination” on page 366).

In cases where there are conditions on a partitioning column that permit partition elimination, the Optimizer uses the number of active partitions rather than the total number of partitions.

In cases where there is a range constraint between the partitioning column of a PPI relation and the other table that can be used to generate a partition-level constraint, the AMP software applies Dynamic Partition Elimination to the operation.

**Sliding Window Merge Join Between a Primary-Indexed Table and a NoPI Table**

Teradata Database uses the following process to make a Sliding Window Merge Join between a primary-indexed table and a NoPI table:

1. Teradata Database reads the first data block for the NoPI table.
2. Teradata Database reads the first data block of each nonexcluded partition of the PPI table into memory.
Teradata Database compares the rows from the NoPI data block to the rows of each PPI data block.

The join routines present the rows in row hash order, but the process is conceptually similar to visiting the data block for each partition in turn.

As the rows of a data block are exhausted, Teradata Database reads the next data block for that partition. This results in each data block of each table being read only once because of partitioning.

Dynamic Partition Elimination for a Sliding Window Merge Join Between a Primary-Indexed Table and a NoPI Table

The Optimizer cannot use dynamic partition elimination for a Sliding Window Merge Join between a primary-indexed table and a NoPI table.

Example: Sliding Window Merge Join

In the following example, the two tables are joined on their primary indexes.

The WHERE conditions eliminate all but two partitions for level 1, and fifteen partitions for level 2 of orders.

The WHERE conditions eliminate all but one partition for level 1, and nineteen partitions for level 2 of lineitem.

After the system applies these conditions, it must join 30 partitions of the combined partitioning expression for orders to 19 partitions of the combined partitioning expression of lineitem, making a total of 49 partitions.

Assume the following properties for this specific query:

- The remaining partitions are all populated
- Only 18 partitions can be joined at a time based on the PPICacheThrP setting for your system

Because of this situation, the system is able to join the two tables using a Sliding Window Merge Join, assuming the Optimizer estimates it to be the most cost effective join method.

Assume that the Optimizer decides that 8 partitions from orders are to be joined to 10 partitions from lineitem at a time.

Also assume that the Optimizer clusters the partitions from the respective tables into the following sets of windows for making the join:

- The Optimizer divides the partitions of orders into 4 sets of 8, 8, 8, and 6 partitions.
- The Optimizer divides the partitions of lineitem into 2 sets of partitions of 10 and 9 partitions.

The system directly merge joins each set of partitions from orders to each set of partitions from lineitem, making a total of 8 Single Window Merge Joins.
The definition DDL statement text for the two tables to be joined in this example query is as follows:

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

```
CREATE TABLE lineitem (  
l_orderkey INTEGER NOT NULL,  
l_partkey INTEGER NOT NULL,  
l_suppkey INTEGER,  
l_linenumber INTEGER,  
l_quantity INTEGER NOT NULL,  
l_extendedprice DECIMAL(13,2) NOT NULL,  
l_discount DECIMAL(13,2),  
l_tax DECIMAL(13,2),  
l_returnflag CHARACTER(1),  
l_linestatus CHARACTER(1),  
l_shipdate DATE FORMAT 'yyyy-mm-dd',  
l_commitdate DATE FORMAT 'yyyy-mm-dd',  
l_receiptdate DATE FORMAT 'yyyy-mm-dd',  
l_shipinstruct VARCHAR(25),  
l_shipmode VARCHAR(10),  
l_comment VARCHAR(44))  
PRIMARY INDEX (l_orderkey)  
PARTITION BY (  
RANGE_N(l_suppkey BETWEEN 0  
AND 4999  
EACH 10),  
RANGE_N(l_shipdate BETWEEN DATE '2000-01-01'  
AND DATE '2006-12-31'  
EACH INTERVAL '1' MONTH));
```
Given these table definitions, the Optimizer selects the Direct Sliding Window Merge join method for the join plan to join the 49 qualifying partitions when it processes the following query:

```sql
SELECT *
FROM orders, lineitem
WHERE o_orderkey = l_orderkey
AND o_orderdate BETWEEN DATE '2005-04-01'
  AND DATE '2006-06-30'
AND o_custkey IN (618, 973)
AND l_shipdate BETWEEN DATE '2005-04-01'
  AND DATE '2006-10-31'
AND l_suppkey = 4131;
```
Hash Join

Introduction

Hash Join is a method that performs better than Merge Join (see “Merge Join” on page 372) under certain conditions. Hash Join is applicable only to equijoins. The performance enhancements gained with the hybrid Hash Join comes mainly from eliminating the need to sort the tables to be joined before performing the actual join operation.

Depending on whether the large table in the join must be spooled, the Optimizer might substitute a Dynamic Hash Join (see “Dynamic Hash Join” on page 404) in place of a standard Hash Join. If join conditions are on a column set that is not the primary index, then some relocation of rows must be performed prior to the sort operation.

Hash joins, like other join methods, perform optimally when the statistics on the join columns are current. This is particularly important for hash join costing to assist the Optimizer in detecting skew in the data (see “Effects of Data Skew on Hash Join Processing” on page 401 for details about the negative effect of skewed data on hash join performance).

Hash Join Terminology

Description of the Hash Join method requires several new terms, which are defined in the following table:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Table</td>
<td>The smaller of the two join tables.</td>
</tr>
<tr>
<td></td>
<td>So named because it is used to build the hash table.</td>
</tr>
<tr>
<td>Fanout</td>
<td>The maximum number of partitions to be created at each partitioning level.</td>
</tr>
<tr>
<td>Hash Join</td>
<td>A join algorithm in which a hash table is built for the smaller of the two tables being joined based on the join column set. The larger table is then used to probe the hash table to perform the join.</td>
</tr>
<tr>
<td>Hash Table</td>
<td>A memory-resident table composed of several hash buckets, each of which has an ordered linked list of rows belonging to a particular hash range.</td>
</tr>
<tr>
<td>Probe Table</td>
<td>The larger of the two join tables in the hash join.</td>
</tr>
<tr>
<td></td>
<td>Rows from this table are used to probe rows in the hash table for a hash match before the join conditions are evaluated.</td>
</tr>
</tbody>
</table>
Chapter 3: Join Planning and Optimization
Hash Join

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition</td>
<td>A segment of a table in a hash join.</td>
</tr>
<tr>
<td></td>
<td>Tables are segmented into a number of partitions using the equation described in “Assigning Rows to a Hash Join Partition” on page 400.</td>
</tr>
<tr>
<td></td>
<td>Partitions can be memory-resident, in which case they also constitute the hash table, or a spool file.</td>
</tr>
<tr>
<td></td>
<td>The limit on the number of partitions for a hash join operation is 50.</td>
</tr>
<tr>
<td></td>
<td>Note that Hash Join partitions have no relationship to the partitions of a Partitioned Primary Index (see SQL Data Definition Language and Database Design). They are entirely different, unrelated things.</td>
</tr>
</tbody>
</table>

Classic Hash Join

Classic Hash Join is applicable when the entire build table fits into available memory. The system reads rows from the build relation directly into a memory-resident hash table. The process then reads rows from the probe table and uses them to individually probe the hash table for a match. A result row is returned for each hash match that satisfies all the join conditions. No partitioning, and thus no partitioning-related I/O, is incurred in this case. Consequently, it is the fastest form of hash join.

Teradata Database uses a variant of Classic Hash Join commonly referred to as Hybrid Hash Join (see Yu and Meng, 1998, for a concise description of the differences between the Classic Hash Join and the Hybrid Hash Join) as well as a variant of the Hybrid Hash Join referred to as Dynamic Hash Join (see “Dynamic Hash Join” on page 404). The method deals with build tables that are too large to fit into available memory by dividing them into chunks called partitions that are small enough to fit into memory (see “Partitioning the Smaller Table of a Hash Join Pair” on page 399 for details about how the system does this).

The process applied by the hybrid Hash Join algorithm is provided in the following table:

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.
4. End of process.
The following graphic illustrates the Hybrid Hash Join process:

**Partitioning the Smaller Table of a Hash Join Pair**

A hash table is derived from the smaller table in a hash join pair. It is a single-partition, memory-resident data structure that contains a hash array as well as the rows in the larger table that the hash array points to. The system configures the smaller table in a hash join operation as an ordered linked list of rows that belong to a particular range of hash values. Depending on its size, this table can be decomposed into several smaller partitions.

When the smaller table is too large to fit into the memory available for Hash Join processing, the system splits it into several smaller, range-bounded partitions. Each partition is small enough to fit into the available space. Partition size is controlled by the settings of several flags in the DBS Control record (see “Hash Join Control Variables” on page 401). The default partition size for a nonduplicated (larger) table is roughly 51 Kbytes, while the default partition size for a duplicated (smaller) table is roughly 204 Kbytes. A table can be divided into a maximum of 50 partitions. Partitioning avoids the maintenance overhead that would otherwise be required for virtual memory management whenever a hash bucket overflow condition occurs.

The system segments the smaller table using an algorithm that uses the join columns to hash rows into the number of partitions required to make the join (see “Assigning Rows to a Hash Join Partition” on page 400).
For example, suppose six partitions are needed to make the join. That is, the number of qualifying rows for the smaller table is six times larger than the largest single partition that can fit into the available memory. The system then hashes each row into one of the six partitions.

The system spools and partitions the larger table in the same way. Although the partitions for the large table are also larger, they need not fit into memory. When the system makes the join, it brings a partition of the smaller table, which is copied to a spool file, into memory. Rows from the matching partition in the other table, which is also in a spool file, can then be matched to the rows in memory.

Note that a row from one partition cannot match a row in the other table that is in a different partition because their respective hash values are different.

Each left table partition is then hash-joined in turn with its corresponding right table partition. The graphic in the section on “Classic Hash Join” shows partition 2 of the triple-partitioned left table being hash-joined with partition 2 of the triple-partitioned 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, which is also illustrated in the graphic in the section on “Classic Hash Join.” The process of creating the partitions is referred to as fanning out in EXPLAIN reports (see “Hash Join Example” on page 403, where the phrases that describe the hash join and the partitioning of the hash join tables are highlighted in boldface).

**Assigning Rows to a Hash Join Partition**

When the number of qualifying rows in a Hash Join table is too large to fit into the available memory, the system assigns groups of its rows to smaller table partitions using the following equation:

$$\text{partition_number} = \text{row_hash_value}(\text{MOD(fanout)})$$

where:

<table>
<thead>
<tr>
<th>Equation element</th>
<th>Specifies the …</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition_number</td>
<td>number of the memory-resident Hash Join partition to which a row from the table is assigned.</td>
</tr>
<tr>
<td>row_hash_value</td>
<td>row hash value for the row in question. See Database Design for details.</td>
</tr>
<tr>
<td>MOD</td>
<td>modulo function.</td>
</tr>
<tr>
<td>fanout</td>
<td>maximum number of partitions to be created at each partitioning level in the Hash Join operation.</td>
</tr>
</tbody>
</table>
Effects of Data Skew on Hash Join Processing

Data skew in the build table can seriously degrade the performance of Hash Join. One of the premises of Hash Join is that the hashing algorithm is good enough to ensure that the build relation can be reduced into relatively equivalent-sized partitions. When there is a large quantity of duplicate row values in the build table, the hash partitioning algorithm might not partition it optimally. Skew in the probe table can also degrade performance if it results in the probe table being smaller than the build table.

To make allowances for possible skew in either table in a hash join operation, you can use the DBS Control utility to size build table partitions proportionately to their parent hash table (see “SkewAllowance” on page 402).

If the specified skew allowance is insufficient to correct for data skew, and hash table bucket overflow occurs, then the system matches rows from the corresponding probe table against build table rows that are already loaded into the memory-resident hash table. After all the probe partition rows have been processed, the system clears the hash table and moves more rows from the oversized build table partition into memory. The system then re-reads rows from the probe partition and matches them against the freshly loaded build table rows. This procedure repeats until the entire build partition has been processed.

Controlling the Size of a Hash Table

You can control the size of the hash table using the HTMemAlloc and HTMemAllocBase flags of the DBS Control record (see “HTMemAlloc” on page 402 and “HTMemAllocBase” on page 402). If you specify a value of 0, the system cannot build a hash table. This effectively turns off Hash Join, and the Optimizer does not consider the method when it is doing its join plan evaluations.

Hash Join Control Variables

You can access the Hash Join-related flags HTMemAlloc and SkewAllowance using the DBS Control utility (see Utilities). Use them to optimize the performance of your Hash Joins. HTMemAllocBase is an internal DBS Control flag that can only be accessed by Teradata support personnel. Contact your Teradata technical support team if you suspect the value of this flag needs to be changed.
Hash Join Costing

The cost for both regular and dynamic hash joins is determined from the sum of the following six components:

- Preparation cost for the left relation
- Preparation cost for the right relation
- Read cost for the left relation
- Read cost for the right relation
- Build cost for the hash array
- Probe cost for the hash array
The cost of a regular hash join is computed as follows:

\[
\text{Cost of regular hash join} = \text{LeftPrepCost} + \text{RightPrepCost} + \text{LeftRelCost} + \text{RightRelCost} + \text{HTBuildCost} + \text{HTProbeCost}
\]

The cost of a dynamic hash join is computed as follows:

\[
\text{Cost of dynamic hash join} = \text{LeftPrepCost} + \text{LeftRelCost} + \text{RightRelCost} + \text{HTBuildCost} + \text{HTProbeCost}
\]

where:

<table>
<thead>
<tr>
<th>Costing term ...</th>
<th>Specifies the cost of ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeftPrepCost</td>
<td>preparing the left relation for the hash join.</td>
</tr>
<tr>
<td>RightPrepCost</td>
<td>preparing the right relation for the hash join.</td>
</tr>
<tr>
<td>LeftRelCost</td>
<td>reading the left relation.</td>
</tr>
<tr>
<td>RightRelCost</td>
<td>reading the right relation.</td>
</tr>
<tr>
<td>HTBuildCost</td>
<td>building the hash array.</td>
</tr>
<tr>
<td>HTProbeCost</td>
<td>probing the hash array.</td>
</tr>
</tbody>
</table>

The only difference between the two costs is the absence of the RightPrepCost term from the costing equation for a Dynamic Hash Join.

**Hash Join Example**

The Optimizer decides to hash join the Employee and Department tables on the equality condition Employee.Location = Department.Location in this query. The EXPLAIN text indicates that the hash tables in Spool 2 (step 4) and Spool 3 (step 5) are segmented (fanned out) into 22 hash join partitions (see “Partitioning the Smaller Table of a Hash Join Pair” on page 399).

Hash table memory allocation is set at 5 percent and skew allowance is set at 75 percent (see “Effects of Data Skew on Hash Join Processing” on page 401).

```
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.***
Chapter 3: Join Planning and Optimization

Hash Join

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 40 minutes.

DBS Control Record - Performance Fields:
HTMemAlloc = 5%
SkewAllowance = 75%

Dynamic Hash Join

Dynamic Hash Join\(^{13}\) provides the ability to do an equality join directly between a small table and a large table on nonprimary index columns without placing the large table into a spool file. For Dynamic Hash Join to be used, the left table must be small enough to fit in a single partition.

Dynamic Hash Join can be used only when two tables are joined based on nonprimary index columns, and one table, referred to as the left table, is very small compared to the other.

The process is as follows:

1. Duplicate the smaller table.
2. Place the smaller table in a hash array
3. Read the right table
4. Compute the row hash code
5. Do a hash join lookup

This is faster than duplicating the small table, putting it into the hash array, reading the large table, building the row hash code, writing the data out to spool, and then reading the table in again to do the hash join.

\(^{13}\) Sometimes called Hash Join On the Fly. This terminology is deprecated.
Nested Join

Definition

Nested Join is a join for which the WHERE conditions specify an equijoin with a constant value for a unique index in one table and those conditions also match some column of that single row to a primary or secondary index of the second table (see “Cost Effectiveness”). The Nested Join is one of the most performant joins available because it is the only join type that need not touch all AMPs in order to join the relations.

Types of Nested Join

There are two types of Nested Join: local and remote.

Local Nested Join is more commonly used than remote nested joins. Local Nested Join is described in “Local Nested Join” on page 407.

Remote Nested Join is described in “Remote Nested Join” on page 412.

Process

Teradata Database uses the following general process to perform a nested join:

1. Retrieve the single row that satisfies the WHERE conditions from the first relation.
2. Use that row to locate the AMP having the matching rows on which the join is to be made.
3. End of process.

Nested Join Between a Primary-Indexed Relation and a NoPI Relation

There are two valid cases for nested joins when one of the relations is a NoPI table:

- The NoPI table is the first relation specified, it has a USI, and the USI is used to access the primary table row. The join column is then hashed and used to probe the index of the second table, which can be either unique or nonunique.
- The NoPI table is the second relation specified. The row IDs from the secondary index of the NoPI table must point to its base table rows.
**Costing a Nested Join**

Nested joins are costed as follows:

\[
\text{Cost}_{\text{Nested Join}} = \text{Cost}_{\text{Accessing Left Relation}} + \text{Cost}_{\text{Accessing Right Relation by a Multivalued Index Access Path}}
\]

The cost of accessing the right relation by means of a multivalued access path is determined by the sum of the costs of the following component operations:

- Cost of accessing the index subtable with multiple key values.
- Cost of spooling and, optionally, sorting row ID values from the index access.
- Cost of accessing the base table by means of the row ID spool.

**Cost Effectiveness**

Nested Join is very cost-effective because it is the only join type that does not always use all AMPs. Because of this, Nested Join is 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.

- 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 434). 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 408)
- Fast path (see “Fast Path Local Nested Join” on page 411)

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 434.

<table>
<thead>
<tr>
<th>IF the equality condition is on this index type ...</th>
<th>THEN the left table is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>USI</td>
<td>1 Hash-distributed based on the joined column.</td>
</tr>
<tr>
<td></td>
<td>2 Nested Joined with the right table.</td>
</tr>
<tr>
<td></td>
<td>3 The resulting rows are redistributed by row hashing the rowID of the right table rows.</td>
</tr>
<tr>
<td></td>
<td>4 The rowID is used to retrieve the data rows from the right table to complete the join.</td>
</tr>
<tr>
<td></td>
<td>5 End of process.</td>
</tr>
<tr>
<td>NUSI</td>
<td>1 Duplicated on all AMPs.</td>
</tr>
<tr>
<td></td>
<td>2 Nested Joined with the right table.</td>
</tr>
<tr>
<td></td>
<td>3 The resulting rows are redistributed by row hashing the rowID of the right table rows.</td>
</tr>
<tr>
<td></td>
<td>4 The rowID is used to retrieve the data rows from the right table to complete the join.</td>
</tr>
<tr>
<td></td>
<td>5 End of process.</td>
</tr>
</tbody>
</table>
**Slow Path Local Nested Join**

**Slow Path Local Nested Join Process**

The following list documents the process applied by the slow path Nested Join algorithm:

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.
8. End of process.

The following graphics illustrate the generic slow path local Nested Join process:

**Step 1**

![Diagram: Left Table -> Right Table Index]

**Step 2**

![Diagram: Left Table with Right Table Row IDs -> Right Table]
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:

```sql
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 410.
Example of a Local Nested Join Operation

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;
```

<table>
<thead>
<tr>
<th>Enum</th>
<th>Ename</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK</td>
<td>Brown</td>
<td>200</td>
</tr>
<tr>
<td>UPI</td>
<td>Smith</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>Jones</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>Clay</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>Peters</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Foster</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>Gray</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>Baker</td>
<td>310</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th>DeptName</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>Education</td>
</tr>
<tr>
<td>150</td>
<td>Payroll</td>
</tr>
<tr>
<td>200</td>
<td>Finance</td>
</tr>
<tr>
<td>310</td>
<td>Mfg</td>
</tr>
</tbody>
</table>
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.

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 $\geq$ to that of the left base table row.

<table>
<thead>
<tr>
<th>IF the row hash values are ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>join the two rows.</td>
</tr>
<tr>
<td>not equal</td>
<td>use the larger row hash value to read the row from the other table.</td>
</tr>
</tbody>
</table>

3. End of process.

Example

The following SELECT request is an example of a query that is processed using a very simple fast path local Nested Join:

```sql
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 Join is 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:

```sql
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 as follows:

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.
5. End of process.
Remote Nested Join is 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)
- Nonindexed 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 following process occurs:

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.
3. End of process.
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

\[(\text{table\_1.UPI = constant OR table\_1.USI = constant}) \land (\text{table\_2.UPI = constant OR table\_2.USI = constant})\]

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

Example 2

\[(\text{table\_1.UPI = constant OR table\_1.USI = constant}) \land ((\text{table\_2.NUPI = table\_1.field}) \lor (\text{table\_2.USI = table\_1.field}))\]

Example 3

\[(\text{table\_1.NUPI = constant}) \land (\text{table\_2.UPI = table\_1.field}) \land \text{few rows returned for the table\_1.NUPI = constant}\]
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 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>
<thead>
<tr>
<th>Table Name</th>
<th>Column 1 Name</th>
<th>Column 2 Name</th>
<th>Primary Index Name</th>
<th>Unique Secondary Index Name</th>
<th>Number of Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_1</td>
<td>NUPI</td>
<td>y_1</td>
<td>NUPI</td>
<td>none defined</td>
<td>2</td>
</tr>
<tr>
<td>table_2</td>
<td>NUPI</td>
<td>USI_2</td>
<td>NUPI</td>
<td>USI_2</td>
<td>1,000,000</td>
</tr>
</tbody>
</table>

Test Query

The following query is tested against this database, both using and without using Nested Join:

```sql
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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 1: Product Join</td>
<td>duped table_1, direct table_2</td>
<td>1 hour 15 minutes</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 3:Nested Join</td>
<td>(Hashed table_1, index table_2)</td>
<td>0.24 seconds</td>
</tr>
<tr>
<td>Spool 1:rowID Join</td>
<td>(Hashed spool_3, direct table_2)</td>
<td>0.22 seconds</td>
</tr>
</tbody>
</table>

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

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, Exclusion Join finds rows in the first table that do not have a matching row in the second table.

Exclusion Join is an implicit form of the outer join.

Also see “Inclusion and Exclusion Product Joins With Dynamic Partition Elimination” on page 425.

SQL Operators That Often Cause 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 Join (see “Exclusion Merge Join” on page 419)
- Exclusion Product Join (see “Exclusion Product Join” on page 422)
Exclusion Merge Join

Exclusion Merge Join Process

The process applied by the Exclusion Join algorithm is as follows:

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.
4. End of process.

Example 1

The following SELECT statement is an example of an Exclusion Merge Join:

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

The following stages document a concrete example of the Exclusion Join 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.
6. End of process.

When the last AMP has completed its portion of the join, the contents of all result spools are sent to the user by means of a BYNET merge.
The processing stages of an Exclusion Merge Join are like those used in the Exclusion Product Join (see “Exclusion Join” on page 418), 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 Join.

**Example 2**

Consider the following *Employee* and *Customer* tables:

<table>
<thead>
<tr>
<th>Enum</th>
<th>Name</th>
<th>Job_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK</td>
<td>Brown</td>
<td>512101</td>
</tr>
<tr>
<td>PK</td>
<td>Smith</td>
<td>412101</td>
</tr>
<tr>
<td>PK</td>
<td>Jones</td>
<td>512101</td>
</tr>
<tr>
<td>PK</td>
<td>Clay</td>
<td>412101</td>
</tr>
<tr>
<td>PK</td>
<td>Peters</td>
<td>512101</td>
</tr>
<tr>
<td>PK</td>
<td>Foster</td>
<td>512101</td>
</tr>
<tr>
<td>PK</td>
<td>Gray</td>
<td>413201</td>
</tr>
<tr>
<td>PK</td>
<td>Baker</td>
<td>512101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cust</th>
<th>Sales_enum</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPI</td>
<td></td>
</tr>
<tr>
<td>PK</td>
<td>23</td>
</tr>
<tr>
<td>PK</td>
<td>24</td>
</tr>
<tr>
<td>PK</td>
<td>25</td>
</tr>
<tr>
<td>PK</td>
<td>26</td>
</tr>
<tr>
<td>PK</td>
<td>27</td>
</tr>
<tr>
<td>PK</td>
<td>28</td>
</tr>
<tr>
<td>PK</td>
<td>29</td>
</tr>
<tr>
<td>PK</td>
<td>30</td>
</tr>
</tbody>
</table>

The graphic “Exclusion Merge Join Row Distribution” on page 421 illustrates the row redistribution caused by the following SELECT request:

```sql
SELECT name
FROM employee
WHERE job_code = 512101
AND enum NOT IN (SELECT sales_enum
FROM customer);
```
Exclusion Merge Join Row Distribution

CUSTOMER ROWS HASH DISTRIBUTED ON CUST_NUM (UPI):

- 30 6
  - 24 3
- 23 6
  - 29 1
- 28 8
  - 27 6
- 25 8
  - 26 1

CUSTOMER.SALES_ENUM (AFTER HASHING AND DUPLICATE ELIMINATION):

- 6
  - 8
- 3
- 1

QUALIFYING EMPLOYEE ROWS STILL DISTRIBUTED ON ENUM (UPI):

- 6 FOSTER 512101
  - 8 BAKER 512101
- 3 JONES 512101
- 1 BROWN 512101
- 5 PETERS 512101
Exclusion Product Join

Exclusion Product Join Process

The process applied by the Exclusion Product Join algorithm is as follows:

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.
3. End of process.

Example: Exclusion Product Join

The following request returns names of employees who do not work in Chicago:

```sql
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:

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.
6. End of process.

The graphic “Exclusion Product Join Operation” on page 423 illustrates this process.
Chapter 3: Join Planning and Optimization

Exclusion Product Join Operation

Exclusion Product Join Operation

SQL Request and Transaction Processing
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 Join
- Inclusion Product Join

Also see “Inclusion and Exclusion Product Joins With Dynamic Partition Elimination” on page 425.

Inclusion Merge Join

The process applied by the Inclusion Merge Join Algorithm is as follows:

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.
3. End of process.

Inclusion Product Join

The process applied by the Inclusion Product Join algorithm is as follows:

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.
3. End of process.
Inclusion and Exclusion Product Joins With Dynamic Partition Elimination

Introduction

When the Optimizer determines that the least costly method for making a join between a PPI table and another relation on either an IN or NOT IN condition, it can choose to use either an Inclusion or Exclusion Product Join with DPE (see “Inclusion Product Join With Dynamic Partition Elimination” on page 429 and “Exclusion Product Join With Dynamic Partition Elimination” on page 432). Whether it uses an Inclusion or Exclusion Product Join with DPE depends on the condition on which the relations are joined.

<table>
<thead>
<tr>
<th>IF the join condition is based on this term ...</th>
<th>THEN the Optimizer chooses this type of Product Join ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Inclusive with DPE.</td>
</tr>
<tr>
<td>NOT IN</td>
<td>Exclusive with DPE.</td>
</tr>
</tbody>
</table>

The performance enhancement seen for Inclusion and Exclusion DPE Product Joins over standard Inclusion and Exclusion Product Joins occurs for two reasons:

- The number of comparisons between left relation and right relation rows is reduced, which in turn reduces the I/O on the left and right relations and the CPU costs to perform the join.
- The following is true for the case when one relation in a join is a table and the other is a spool file.

  When the number of values in the spool file is small, resulting in fewer populated spool file partitions than populated PPI table partitions, then those table partitions that are not in spool need not be either read or joined, reducing both the I/O and CPU costs of the join.

Because the performance improvement gained from DPE joins is highly dependent on the number of rows in each partition (and so the number of row comparisons that must be made), the Optimizer does not choose a DPE join unless single-column PARTITION statistics have been collected on the outer (PPI) table in the join. PARTITION statistics improve the ability of the Optimizer to estimate the number of comparisons that must be made. The Optimizer does not choose a DPE join when there is a no confidence estimate on the cardinality of the subquery spool.
Chapter 3: Join Planning and Optimization
Inclusion and Exclusion Product Joins With Dynamic Partition Elimination

For example, suppose you create the following tables and submit the given SELECT request against those tables:

```
CREATE SET TABLE MWS.t1, FALLBACK, NO BEFORE JOURNAL,
    NO AFTER JOURNAL, CHECKSUM = DEFAULT (
    a INTEGER,
    b INTEGER)
PRIMARY INDEX (a);

CREATE SET TABLE MWS.t2, FALLBACK, NO BEFORE JOURNAL,
    NO AFTER JOURNAL, CHECKSUM = DEFAULT (
    a INTEGER,
    b INTEGER,
    c INTEGER,
    d INTEGER)
PRIMARY INDEX (a)
PARTITION BY (RANGE_N(b BETWEEN 1 /* Partitioning level one. */
    AND 100
    EACH 7,
    NO RANGE OR UNKNOWN),
    RANGE_N(c BETWEEN 1 /* Partitioning level two. */
    AND 100
    EACH 10,
    NO RANGE OR UNKNOWN),
    RANGE_N(d BETWEEN 1 /* Partitioning level three. */
    AND 100
    EACH 20,
    NO RANGE OR UNKNOWN));

SELECT *
FROM t2
WHERE (b,c) IN (SELECT a,b
    FROM t1);
```

For this request to be eligible for an Inclusion or Exclusion Product Join with DPE, you must collect the following statistics:

```
COLLECT STATISTICS t1 COLUMN(a);
COLLECT STATISTICS t2 COLUMN(PARTITION);
```

The following rules apply to collecting these statistics:

<table>
<thead>
<tr>
<th>To enable this type of join...</th>
<th>You must collect statistics on the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product (all kinds) with DPE</td>
<td>primary index of the table that must be duplicated to enable an Inclusion or Exclusion Product Join with DPE.</td>
</tr>
<tr>
<td>Inclusion or Exclusion Product with DPE</td>
<td>system-derived PARTITION column set of the PPI table.</td>
</tr>
</tbody>
</table>

Another name for the type of join being done with Inclusion and Exclusion Product Joins is *semijoin*. As a result, the Inclusion and Exclusion Product Joins with DPE can also be referred to by the name Product Semijoin with DPE.
When a Product Semijoin with DPE is selected by the Optimizer, the EXPLAIN text for the query indicates enhanced by dynamic partition elimination for the corresponding AMP step. In the following example, this text is highlighted in boldface type:

```
EXPLAIN SELECT COUNT(*)
    FROM t55
    WHERE (b,c) NOT IN (SELECT 1, 1);
```

**Explanation**

1) First, we lock a distinct MWS."pseudo table" for read on a RowHash to prevent global deadlock for MWS.t55.
2) Next, we lock MWS.t55 for read.
3) We do an INSERT into Spool 5.
4) We do an all-AMPs RETRIEVE step from Spool 5 (Last Use) by way of an all-rows scan into Spool 4 (group_amps), which is redistributed by the hash code of (1, 1) to all AMPs. The size of Spool 4 is estimated with high confidence to be 1 row (22 bytes). The estimated time for this step is 0.03 seconds.
5) We do a group-AMP RETRIEVE step from Spool 4 (Last Use) by way of an all-rows scan into Spool 7 (all_amps), which is duplicated on all AMPs. Then we do a SORT to partition by rowkey. The size of Spool 7 is estimated with high confidence to be 2 rows (30 bytes). The estimated time for this step is 0.03 seconds.
6) We do an all-AMPs JOIN step from MWS.t55 by way of an all-rows scan with no residual conditions, which is joined to Spool 7 (Last Use) by way of an all-rows scan. MWS.t55 and Spool 7 are joined using an exclusion product join with a join condition of "(MWS.t55.b - Field 2) AND (MWS.t55.c - Field 3)" enhanced by dynamic partition elimination. The input table MWS.t55 will not be cached in memory, but it is eligible for synchronized scanning. The result goes into Spool 3 (all_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 3 is estimated with index join confidence to be 2,370,668 rows (35,560,020 bytes). The estimated time for this step is 39.38 seconds.
7) We do an all-AMPs SUM step to aggregate from Spool 3 (Last Use) by way of an all-rows scan. Aggregate Intermediate Results are computed globally, then placed in Spool 8. The size of Spool 8 is estimated with high confidence to be 1 row (23 bytes). The estimated time for this step is 3.06 seconds.
8) We do an all-AMPs RETRIEVE step from Spool 8 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.02 seconds.
9) 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 Optimizer selects an Inclusion or Exclusion Product Join with DPE when there are equality join terms between the partitioning columns at one or more partitioning levels of a PPI table, which is the outer table of a semijoin, and another relation. Inclusion and Exclusion Product Joins with DPE enhance the performance of such queries for those cases where the PPI table is large and a costly sort in preparation for a regular Inclusion or Exclusion Merge Join can be avoided. Such queries further benefit when the number of rows for the join column selected by the subquery is small because in that case, the number of values in the spool file is small, resulting in fewer populated spool file partitions than populated PPI table partitions. As a result, the table partitions that are not in spool need neither need to be read nor joined, reducing both the I/O and CPU costs of the join.
For example, assume you have the following table definitions and cardinalities:

```sql
CREATE SET TABLE t1 (  
a INTEGER,  
b INTEGER,  
c INTEGER)  
PRIMARY INDEX (a);

CREATE SET TABLE t8 (  
a INTEGER,  
b INTEGER,  
c INTEGER)  
PRIMARY INDEX (a)  
PARTITION BY (RANGE_N(c BETWEEN 1 AND 1200 EACH 30, NO RANGE OR UNKNOWN),  
RANGE_N(b BETWEEN 1 AND 11000 EACH 7, NO RANGE OR UNKNOWN));
```

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Table Cardinality (rows)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>1,000</td>
</tr>
<tr>
<td>t8</td>
<td>9,000,000</td>
</tr>
</tbody>
</table>

For the following query, the number of rows returned by the subquery is small, so it is possible to eliminate many partitions in `t8` dynamically, making a Product Join cost effective. Additionally, it is not necessary to sort the `t8` rows to make the join.

```sql
SELECT COUNT(*)  
FROM t8  
WHERE (b,c) IN (SELECT a,b  
FROM t1  
WHERE c = 1);
```

```plaintext
*** Query completed. One row found. One column returned.  
*** Total elapsed time was 1 second.

Count(*)  
--------
65
```

On the other hand, if the Optimizer were to use a Merge Join instead of an Inclusion Product Join with DPE, then the `t8` rows would have to be sorted, which takes the majority of the processing time, and the query takes 57 seconds to complete rather than 1 second.
Inclusion Product Join With Dynamic Partition Elimination

Inclusion Product Join with Dynamic Partition Elimination (DPE), which is designed for use when the outer relation in a join based on an IN term is a PPI table, differ from Inclusion Product Joins without DPE in that the DPE versions of the join are driven from the inner table instead of the outer, or PPI, table, while the non-DPE versions are driven from the outer table.

For example, suppose you have the following table definitions and query against those tables:

```
SELECT COUNT(*)
FROM t8
WHERE (b,c) IN (SELECT a,b
    FROM t1
    WHERE c = 1);
```

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

**Count(*)**

---------

| 65 |

Be aware that the performance enhancement achieved with Inclusion and Exclusion Product Joins with DPE is not always this great, and the degree of the enhancement depends heavily on two factors:

- The number of rows per partition of the PPI table
- The number of columns projected by the subquery

Costing takes into account those cases for which the PPI table has a highly variable number of rows per partition, and avoids selecting an Inclusion or Exclusion Product Join with DPE for those tables.
CREATE SET TABLE MWS.t1, FALLBACK, NO BEFORE JOURNAL, 
NO AFTER JOURNAL, CHECKSUM = DEFAULT ( 
a INTEGER, 
b INTEGER) 
PRIMARY INDEX (a); 

CREATE SET TABLE MWS.t2, FALLBACK, NO BEFORE JOURNAL, 
NO AFTER JOURNAL, CHECKSUM = DEFAULT ( 
a INTEGER, 
b INTEGER) 
PRIMARY INDEX (a) 
PARTITION BY (RANGE_N(a BETWEEN 1 /* Partitioning level one. */ AND 60000 EACH 60000, 
NO RANGE OR UNKNOWN), 
RANGE_N(b BETWEEN -3 /* Partitioning level two. */ AND 31580 EACH 1, 
NO RANGE, UNKNOWN) ); 

SELECT * 
FROM t2 
WHERE (b) IN (SELECT a 
 FROM t1); 

If the Optimizer were to choose a standard Inclusion Product Join to execute this request, the system would perform the join as follows:

1. Sort t1 to remove duplicate values of column a.
2. Duplicate the sorted values of t1 in a spool file.
3. Read the first row in t2.
   - If there are no more rows to read, go to Stage 6.
4. Search for a match based on the connecting term t2.b = t1.a until a match is found or all rows in spool have been read.
5. Go to Stage 3 and read the next row in t2.
6. End of process.

The system reads all rows in t2 sequentially and joins them with the spool file in this manner.

For an Inclusion Product Join with DPE, the system does not read t2 sequentially, nor does it compare each row in t2 with all rows in the spool file.

When a bind term is specified on all, or on a subset of, the partitioning levels of a PPI table, the system can use the information to build a spool file having the same partitioning as the PPI table. It can then join only the equivalent partitioning ranges between the base table partitions and the spool file partitions.

The setup for the Inclusion Product Join with DPE is the same as that for a standard Inclusion Product Join, except that when the spool is duplicated, it is built with the same partitioning as is defined for the base table, and it is then sorted by partition number. In the previous example, the spool would be partitioned using the partitioning expression that is defined for PPI table t2:
PARTITION BY (RANGE_N(a BETWEEN -3 AND 31580 EACH 1, NO RANGE, UNKNOWN));

Note that the partitioning is done on column \( a \) of the \( t1 \) spool because the bind term \( t2.b = t1.a \) links partitioning on \( t2.b \) with \( t1.a \).

The system performs the Inclusion Product Join with DPE as follows:

1. Sort \( t1 \) to remove duplicate values of column \( a \).
2. Duplicate the sorted values of \( t1 \) and partition them in a spool file by the partitioning ranges defined for \( t1 \).
3. Read the first row in the first partition of the spool.
4. Build the list of partitions dynamically in the PPI table that the current spool partition is to join.
5. Read the first row in the first DPE partition of the PPI table.
6. Compare the row with all rows in the current spool partition until either a match is found or there are no more rows in the spool partition.

<table>
<thead>
<tr>
<th>IF a match is ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>found</td>
<td>return the PPI table row.</td>
</tr>
<tr>
<td>not found</td>
<td>go to stage 7.</td>
</tr>
</tbody>
</table>

7. Read the next PPI table row in the participating DPE partitions.

<table>
<thead>
<tr>
<th>IF there are ...</th>
<th>THEN go to stage ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>no more rows</td>
<td>6.</td>
</tr>
<tr>
<td>more rows</td>
<td>4.</td>
</tr>
</tbody>
</table>

8. Read the first row in the spool with a partition number greater than the current spool partition.

<table>
<thead>
<tr>
<th>IF there ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>are no such rows</td>
<td>the join is complete. Go to stage 9.</td>
</tr>
<tr>
<td>is such a row</td>
<td>Go to stage 2.</td>
</tr>
</tbody>
</table>

9. End of process.
Exclusion Product Join With Dynamic Partition Elimination

The algorithm for an Exclusion Product Join with DPE is similar the algorithm for an Inclusion Product Join with DPE, but with a restriction and with some other differences.

The restriction is that Exclusion Product Join with DPE is only enabled with PPI tables where the partitioning expression at each partitioning level consists solely of a RANGE_N function, and the test expression is a simple column. The reason for this restriction is that rows with null partitioning column values at any bound partitioning level must be joined with all partitions at that level. Because that might eliminate much of the performance improvement gained from DPE, Exclusion Product Join with DPE avoids reading the extra PPI table rows by placing all spool rows with null partitioning column values into the error partition.14 Because expressions on columns containing nulls might produce nonnulls, there is no simple method of grouping all rows with nulls into one partition other than using the RANGE_N function with a simple column test expression.

To ensure that rows with null partitioning columns are placed in the error partition, the system modifies qualifying RANGE_N expressions to remove both the UNKNOWN and NO RANGE partitions from their definition when they are grouped together as NO RANGE OR UNKNOWN. The join algorithm then joins rows in the error partition with all PPI table rows.

All connecting terms must reference partitioning columns for the Optimizer to choose an Exclusion Product Join with DPE. If additional nonpartitioning columns are components of the connecting conditions, then the Optimizer does not select the Exclusion Product Join with DPE algorithm.

The system performs the Exclusion Product Join with DPE as follows:

1 Build a list of partitions that includes all noneliminated partitions in the PPI table.
   This is the unvisited partition list, which must be joined with the error partition after all DPE partitions have been joined.
2 Read the first row in the first partition of the spool.
3 Dynamically build the list of partitions in t2 that this current spool partition is to join.
   Remove this set of partitions from the unvisited partition list.
4 Read the first row in the first DPE partition of the PPI table.
5 Compare the row with all rows in the current spool partition and with all rows in the error partition.

<table>
<thead>
<tr>
<th>IF there is ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a match</td>
<td>go to stage 6.</td>
</tr>
<tr>
<td>no match</td>
<td>return the PPI table row.</td>
</tr>
</tbody>
</table>

14. Meaning the NO RANGE partition in this case.
6. Read the next PPI table row in the participating DPE partitions.

<table>
<thead>
<tr>
<th>IF there are ...</th>
<th>THEN go to stage ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>no more rows</td>
<td>7.</td>
</tr>
<tr>
<td>more rows</td>
<td>5.</td>
</tr>
</tbody>
</table>

7. Read the first row in the spool with a partition number greater than the current spool partition.

<table>
<thead>
<tr>
<th>IF there ...</th>
<th>THEN go to stage ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>are no such rows</td>
<td>8.</td>
</tr>
<tr>
<td>is such a row</td>
<td>5.</td>
</tr>
</tbody>
</table>

8. Read the first row in the PPI table not eliminated by the unvisited partition list.

9. 

<table>
<thead>
<tr>
<th>IF ...</th>
<th>THEN ...</th>
</tr>
</thead>
</table>
| all of the connecting term columns are null | do not return the row.  
|                         | Go to stage 12.      |
| some, but not all, of the connecting term columns are null | join the row with the rows in the spool.  
|                         | Go to stage 11.      |
| none of the connecting term columns is null | Go to stage 10.      |

10. Join this row with all rows in the error partition of the spool.
11. If no match is found, return the row.
12. Read the next row in the PPI table that was not eliminated by the unvisited partition list.

<table>
<thead>
<tr>
<th>IF there is ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>such a row</td>
<td>go to stage 9.</td>
</tr>
<tr>
<td>no such row</td>
<td>go to stage 13.</td>
</tr>
</tbody>
</table>

13. End of process.
**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 equijointed 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:

```sql
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 as follows:

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.
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.
7. End of process.

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 435.
RowID Join

The following graphic demonstrates a RowID Join:

```
SELECT *
FROM table_1, table_2
WHERE table_1.column_1 = 10
AND table_1.column_3 = table_2.column_5;
```

Example

Assume the following SELECT request 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

Correlated Join constitutes a class of join methods developed to process correlated subqueries. Some types of Correlated Join are extensions of the following more general join types:

- Inclusion Merge Join (see “Inclusion Merge Join” on page 424)
- Exclusion Merge Join (see “Exclusion Merge Join” on page 419)
- Inclusion Product Join (see “Inclusion Product Join” on page 424)
- Exclusion Product Join (see “Exclusion Product Join” on page 422)

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 424) 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.
  This join comes in two forms:
  - Correlated inclusion fast path merge join
  - Correlated inclusion slow path merge join

- **Correlated Exclusion Merge Join**
  Correlated version of standard Exclusion Merge Join. See “Exclusion Merge Join” on page 419.
  This join comes in two forms:
  - Correlated exclusion fast path merge join
  - Correlated exclusion slow path merge join

- **Correlated Inclusion Product Join**
  Correlated version of standard Inclusion Product Join. See “Inclusion Product Join” on page 424.

- **Correlated Exclusion Product Join**
  Correlated version of standard Exclusion Product Join. See “Exclusion Product Join” on page 422.

- **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 as follows:

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.

<table>
<thead>
<tr>
<th>Left Table</th>
<th>Right Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Returns (2, 5, 10, 15)

4. End of process.
# Relational Join Optimization References

The following references are helpful for acquiring a basic understanding of how relational join operations are optimized:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic</td>
<td>Reference</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
Proves that join optimization errors due to out of date statistics propagate at an exponential rate as a function of the number of joins in a query. |
CHAPTER 4 Join Optimizations

This chapter describes several join optimizations.

The performance-oriented aspects of the topics described in this chapter refers only to how the 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 other perspectives can be found in the following books:

• *Database Design*
• *Performance Management*
Chapter 4: Join Optimizations
Star and Snowflake Join 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 illustration on the next page.
As you can see, the graphic representation of such a join resembles a crude drawing of a star:

![Star Join Diagram](image.png)

The concept of star schemas in physical database design is described in *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 *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:

The concept of snowflake schemas in physical database design is described in *Database Design*.

**Primary Target of a Star Join**

Star joins are a fundamental component of a dimensional database model (see *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.

When one or more IN conditions are specified on the large table, the Optimizer might choose to combine the IN lists with the small tables first. The query plan would then join the resulting join relation with the large table. The result is a star join plan with more dimension tables than the number of dimension tables explicitly specified in the query (see stage 6 in “Evaluating Join Orders” on page 344).

**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 $1 \times 10^6$. The next join is between this joined relation and another of the small tables. Its relative cost is $1 \times 10^5$. Finally, this relation is joined with the last small table at a relative cost of $1 \times 10^3$. 

![Diagram of nonoptimal star join plan]

**Diagram Note:**

- **10M**: 10 million
- **1M**: 1 million
- **100K**: 100 thousand
- **1K**: 1 thousand
- **100**: 100
- **ff07D418**: Identifier or key value
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 $1 \times 10^2$, the second a cost of $1 \times 10^4$, and the final join between the large table and its joined small tables relation has a cost of $1 \times 10^3$.

Cost of nonoptimized star join = $1 \times 10^6 + 1 \times 10^5 + 1 \times 10^3 = 1,101,000$

Cost of optimized star join = $1 \times 10^2 + 1 \times 10^4 + 1 \times 10^3 = 11,100$

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 448.

- Unindexed Joins, LT/ST-J2
  For more detailed information on LT/ST-J2 joins, see “LT-ST-J2 Unindexed Joins” on page 449.
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 might 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:
Chapter 4: Join Optimizations
Selecting Indexes for Star Joins

For example, if the star join between Widgets and \((\text{period, state, country})\) is used more often than the star join between Widgets and \((\text{color, shape, size})\), the primary index should be created on \((\text{period, state, country})\).

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 \text{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.

<table>
<thead>
<tr>
<th>IF LT/ST joins are ...</th>
<th>THEN the primary index should be created using the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>used with the same frequency</td>
<td>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.</td>
</tr>
<tr>
<td>not used with the same frequency</td>
<td>collection of join columns associated with the most often used star join</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Table name</th>
<th>Columns</th>
<th>Primary index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>code, description</td>
<td>description</td>
</tr>
<tr>
<td>Size</td>
<td>code, description</td>
<td>description</td>
</tr>
<tr>
<td>Options</td>
<td>code, description</td>
<td>description</td>
</tr>
</tbody>
</table>

Large Table Definition

<table>
<thead>
<tr>
<th>Table name</th>
<th>Columns</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widgets</td>
<td>color, size, options, units, period</td>
<td>1,000,000 rows</td>
</tr>
</tbody>
</table>
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” on page 458
- “Reasonable Indexed Join Plan With Star Join Optimization: Large Table Primary Index Joined to Small Tables” on page 460
- “Reasonable Indexed Join Plan With Star Join Optimization: Large Table USI Joined to Small Tables” on page 462
- “Reasonable Indexed Join Plan With Star Join Optimization: Large Table NUSI Joined to Small Tables” on page 464
- “Join Plan With Star Join Optimization: Large Table Subquery Join” on page 465
- “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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>2</td>
</tr>
<tr>
<td>size</td>
<td>10</td>
</tr>
<tr>
<td>options</td>
<td>10</td>
</tr>
</tbody>
</table>

Large Table Uniqueness Statistics

The following statistics information is given for the large table:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Number of Unique Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>10</td>
</tr>
<tr>
<td>size</td>
<td>100</td>
</tr>
<tr>
<td>options</td>
<td>1000</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 4: Product Join</td>
<td>duped options, direct size</td>
<td>2.67</td>
</tr>
<tr>
<td>Spool 5: Product Join</td>
<td>duped color, direct widgets</td>
<td>6 660.00^a</td>
</tr>
<tr>
<td>Spool 1: Merge Join</td>
<td>duped 4, local 5</td>
<td>7.43</td>
</tr>
</tbody>
</table>

^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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 4: Product Join</td>
<td>duped options, direct size</td>
<td>2.67</td>
</tr>
<tr>
<td>Spool 5: Product Join</td>
<td>duped color, direct widgets</td>
<td>6 660.00^a</td>
</tr>
<tr>
<td>Spool 1: Merge Join</td>
<td>duped 4, local 5</td>
<td>7.43</td>
</tr>
</tbody>
</table>

^a. 1 hour, 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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 3: Product Join</td>
<td>duped color, direct options</td>
<td>0.31</td>
</tr>
<tr>
<td>Spool 5: Product Join</td>
<td>duped size, direct 3</td>
<td>1.62</td>
</tr>
<tr>
<td>Spool 1: Merge Join</td>
<td>hashed 5, direct widgets</td>
<td>4.09</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 3: Product Join</td>
<td>duped color, direct options</td>
<td>0.31</td>
</tr>
<tr>
<td>Spool 5: Product Join</td>
<td>duped size, direct 3</td>
<td>1.62</td>
</tr>
<tr>
<td>Spool 6: Nested Join</td>
<td>hashed 5, index widgets</td>
<td>2.71</td>
</tr>
<tr>
<td>Spool 1: rowID Join</td>
<td>hashed 6, index widgets</td>
<td>5.65</td>
</tr>
</tbody>
</table>

**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:

```
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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 3: Product Join</td>
<td>duped color, direct options</td>
<td>0.31</td>
</tr>
<tr>
<td>Spool 5: Product Join</td>
<td>duped size, direct 3</td>
<td>4.43</td>
</tr>
<tr>
<td>Spool 1: Nested Join</td>
<td>duped 5, index widgets</td>
<td>22.73</td>
</tr>
</tbody>
</table>

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.
Chapter 4: Join Optimizations

Join Plan With Star Join Optimization: Large Table Subquery Join

Example Query

The following query is used for this example:

```sql
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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 4: Product Join</td>
<td>duped color, direct size</td>
<td>0.31</td>
</tr>
<tr>
<td>Spool 6: Product Join</td>
<td>local 4, duped options</td>
<td>4.46</td>
</tr>
<tr>
<td>Spool 1: Nested Join</td>
<td>duped 6, index widgets</td>
<td>22.73</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Cardinality (rows)</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>3</td>
</tr>
<tr>
<td>size</td>
<td>10</td>
</tr>
<tr>
<td>options</td>
<td>10</td>
</tr>
</tbody>
</table>

Large Table Uniqueness Statistics

The following statistics information is given for the large table:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Number of Unique Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>6</td>
</tr>
<tr>
<td>size</td>
<td>30</td>
</tr>
<tr>
<td>options</td>
<td>300</td>
</tr>
</tbody>
</table>

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:

```sql
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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 3: Merge Join</td>
<td>direct options, direct size</td>
<td>0.46</td>
</tr>
<tr>
<td>Spool 4: Product Join</td>
<td>duped color, direct widgets</td>
<td>12 491.00a</td>
</tr>
<tr>
<td>Spool 1: Merge Join</td>
<td>duped 3, local 4</td>
<td>21.05</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Joined Tables</th>
<th>Total Processing Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool 2: Merge Join</td>
<td>direct options, direct size</td>
<td>0.44</td>
</tr>
<tr>
<td>Spool 3: Product Join</td>
<td>direct color, duped 2</td>
<td>1.24</td>
</tr>
<tr>
<td>Spool 1: Merge Join</td>
<td>local widgets, duped 3</td>
<td>7 761.00^a</td>
</tr>
</tbody>
</table>

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 *SQL Data Definition Language*.

Example 1: Simple Join Query
The following is an example of a simple join query:

```sql
EXPLAIN
SELECT o_orderdate, o_custkey, l_partkey, l_quantity, l_extendedprice
FROM Lineitem, Ordertbl
WHERE l_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 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:

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

Explanation
---
1) First, we lock a distinct LOUISB."pseudo table" for read on a 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.OrderJoinLineLine with a range constraint 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 (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:

```sql
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:

```sql
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 1_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:

```sql
CREATE JOIN INDEX CustOrderJoinLine
AS SELECT (l_orderkey, o_orderdate, c_nationkey, o_totalprice),
(l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM (Lineitem
LEFT OUTER JOIN Ordertbl ON l_orderkey=o_orderkey)
INNER JOIN Customer ON o_custkey=c_custkey
PRIMARY INDEX (l_orderkey);

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

EXPLAIN
SELECT (l_orderkey, o_orderdate, c_nationkey, o_totalprice),
(l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM Lineitem, Ordertbl, Customer
WHERE l_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.

<table>
<thead>
<tr>
<th>FOR this category of database operation</th>
<th>Join indexes are maintained in this way</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>The corresponding rows in the join index are located and then deleted with a Merge Delete step.</td>
</tr>
<tr>
<td>INSERT</td>
<td>Newly formed join result rows are added to the join index with a Merge step.</td>
</tr>
<tr>
<td>UPDATE</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

The system 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:

```sql
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:

<table>
<thead>
<tr>
<th>This step ...</th>
<th>Does this ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Reproduces the affected portion of the join index rows.</td>
</tr>
<tr>
<td>6.1</td>
<td>Deletes the corresponding rows from the join index.</td>
</tr>
<tr>
<td>8</td>
<td>Inserts new non-matching outer join rows into the join index.</td>
</tr>
</tbody>
</table>

```
EXPLAIN
DELETE FROM Ordertbl
WHERE o_custkey = 1001;

*** Help information returned. 37 rows.
*** Total elapsed time was 2 seconds.
```

---

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.
Chapter 4: Join Optimizations

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:

<table>
<thead>
<tr>
<th>This step ...</th>
<th>Does this ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Produces the new join result rows for the join index.</td>
</tr>
<tr>
<td>11</td>
<td>Deletes any formerly unmatched outer join rows in the join index.</td>
</tr>
<tr>
<td>12</td>
<td>Inserts the new join result rows into the join index.</td>
</tr>
</tbody>
</table>

EXPLAIN
INSERT Order_tbl
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:

<table>
<thead>
<tr>
<th>This step ...</th>
<th>Does this ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>reproduces the affected portion of the join index rows.</td>
</tr>
<tr>
<td>6.1</td>
<td>deletes the corresponding rows from the join index.</td>
</tr>
<tr>
<td>6.2</td>
<td>reproduces the affected portion of the join index rows and makes the necessary modifications to form the new rows.</td>
</tr>
<tr>
<td>7.1</td>
<td>inserts the newly modified rows into the join index.</td>
</tr>
</tbody>
</table>

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 \texttt{join\_index\_field\_2} is defined, no modifications to \texttt{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:

```sql
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 5 Interpreting the Output of the EXPLAIN Request Modifier

This chapter describes several aspects of the EXPLAIN request modifier. See SQL Data Manipulation Language for the syntax, usage notes, and other operational aspects of the EXPLAIN request modifier.
EXPLAIN Request Modifier

EXPLAIN Report Overview

An EXPLAIN request modifier reports a summary of the query plan generated by the SQL query optimizer to process any valid SQL request: the steps the system would perform to resolve a request. The request itself is not processed.

The EXPLAIN modifier is an extremely useful utility for query designers because it provides an English language summary of the access and join plans generated by the Optimizer for the query you explain with it. The report details which indexes would be used to process the request, identifies any intermediate spool files that would be generated, indicates the types of join to be performed, shows whether the requests in a transaction would be dispatched in parallel, and so on.

When you perform an EXPLAIN against any SQL request, that request is parsed and optimized, and the parse tree (access and join plans) generated by the Optimizer is returned to the requestor in the form of a text file that explains the steps taken in the optimization of the request as well as the relative time it would take to complete the request given the statistics the Optimizer had to work with. The EXPLAIN report reproduces the execution strategy of the Optimizer, but does not explain why it makes the choices it does.

EXPLAIN helps you to evaluate complex queries and to develop alternative, more efficient, processing strategies.

References to bit mapping might appear when complex conditional expressions involving nonunique secondary indexes are applied to a very large table.

Such expressions can be resolved by mapping each subtable row ID to a number in the range 0 - 32,767. This number is used as an index into a bit map in which the bit for each qualifying data row is set equal to 1.

The Optimizer is better able to determine whether the table is a candidate for bit mapping when up-to-date statistics exist for the important columns of that table. See SQL Data Definition Language.

Only the first 255 characters of a conditional expression are displayed in an EXPLAIN. The entire conditional expression is enclosed in quotation marks.

Although the times reported in the EXPLAIN output are presented in units of seconds, they are actually arbitrary units of time. These numbers are valuable because they permit you to compare alternate coding formulations of the same query with respect to their relative performance.

Keep EXPLAIN results with your system documentation because they can be of value when you reevaluate your database design.
EXPLAIN Processes SQL Requests Only

You can modify any valid Teradata SQL statement with EXPLAIN. The definition of what is not a statement (or request) is important here. For example, you cannot EXPLAIN a USING request modifier (see SQL Data Manipulation Language), a WITH RECURSIVE modifier (see SQL Data Manipulation Language), or another EXPLAIN request modifier. Nor can you explain individual functions, stored procedures, or methods.

Effect of Request Cache Peeking on EXPLAIN Reports

If you specify USING data, or if you specify a DATE or CURRENT_DATE built-in function in a request, or both, the system can invoke Request Cache peeking (see “Peeking at the Request Cache” on page 36). When this occurs, the EXPLAIN text reports the peeked literal values (see “Resolving the DATE or CURRENT_DATE Value for Optimization” on page 46 and “Usage Considerations” on page 48 for examples).

If you do not specify USING data, or if the USING variables, DATE or CURRENT_DATE values, or both are not peeked, then there is no impact on either the generated plan or the generated EXPLAIN text.

Note that parameterized requests specified without a USING request modifier, but using either CLIv2 data parcel flavor 3 (Data) or CLIv2 parcel flavor 71 (DataInfo), cannot be explained using any of the following request modifiers or statements:

- EXPLAIN
- DUMP EXPLAIN
- INSERT EXPLAIN


EXPLAIN and Teradata Workload Manager

Some of the information produced by EXPLAIN reports is generated to give the Teradata Workload Manager software hooks to enable it to control various aspects of system workloads. For example, the Optimizer does not need to cost steps that cannot be performed in more than one way, but EXPLAIN text reports\(^1\) generate costs for such information for the benefit of Teradata Workload Manager, which can use its Query Estimator tool to estimate the resources required to process a given query or query workload.

Note that this information is carried in the white tree for Type 2 cost profiles only (see Chapter 2: “Query Rewrite and Optimization,” for information about cost profiles).

---

1. The non-Optimizer-costed “costs” are calculated outside the Optimizer software and then added to the white tree at the time the query optimization process concludes.
Among the steps the Optimizer does not need to cost, but which are calculated and carried by its white tree and reported in EXPLAIN text, are the following:

- Delete
- Insert
- Merge
- Merge Delete
- Merge Row Multiple
- Merge Update
- Update

For more information about Teradata Workload Manager, see *Teradata Dynamic Workload Manager User Guide*.

**Teradata Visual Explain**

The Teradata Visual Explain tool provides a graphic display of Optimizer plans and also permits you to compare the plans for queries that return the same result. This feature can simplify the interpretation of EXPLAIN reports.

For more information about Teradata Visual Explain, see *Teradata Visual Explain User Guide*. Related information appears elsewhere in this chapter and in Chapter 2: “Query Rewrite and Optimization.”
EXPLAIN Confidence Levels

Introduction

When the Optimizer estimates relation and join cardinalities, it does so with an expressed level of confidence in the accuracy of the estimate.

Unlike the probabilistic confidence levels used with the computation of a statistic (such as a t-test or an ANOVA or multiple regression F-test) as a measure of how replicable the result is, Optimizer confidence levels are based on various heuristics. Optimizer confidence levels express a qualitative level of confidence that a given cardinality estimate is accurate given certain knowledge about the available statistics for the tables being analyzed in the process of optimizing a query.

Confidence levels are one of the factors the Optimizer employs to determine which of its available strategies is best at each step of creating a query plan. The lower the confidence estimate, the more conservative the strategy employed, particularly for join planning, because the errors in a query plan are cumulative (and in the case of join planning, errors are multiplicative). Because of this, the Optimizer chooses to pursue a less aggressive query plan, particularly when it comes to join planning, whenever it suspects the accuracy of the data it is using to plan a query step is not high, or is not as reliable as it might be if complete and accurate statistics were available.

Be aware that a cardinality estimate that is based on stale statistics can be inaccurate, causing the Optimizer to generate a less optimal query plan than it otherwise would. It is even possible for the partial statistics collected from a random single-AMP sample to produce a better plan than complete, but stale, statistics, depending on how poorly the stale statistics reflect the demographics of the current set of values making up the population of a column set or index (see SQL Request and Transaction Processing for details).

Optimizer Confidence Levels

An EXPLAIN can report any or all of the following confidence levels for a cardinality estimate:

- No confidence
- Low confidence
- High confidence
- Index Join confidence

These confidence levels are based heavily on the presence or absence of statistics for the column and index sets specified as predicates in the SQL request being reported.

---

2. The system reports Index Join confidence only for join operations.

3. The only exception to this is the case where the query conditions are so complex that statistics cannot be used. In such cases, an EXPLAIN reports no confidence.
Note that even when a join operation has No confidence, the Optimizer still uses any statistics that are available for the condition to enhance the likelihood of producing a better query plan than would otherwise be developed.\(^4\)

Keep in mind that a high confidence level is not a guarantee of an accurate cardinality estimate. For example, suppose the Optimizer locates a query predicate value in one of the statistical histograms for a column or index set. In this case, confidence is assumed to be High. But suppose the available statistics are stale. The Optimizer, by pursuing the assumptions that accrue to a High confidence level, can then produce a bad plan as a result.

The following sections are meant to provide a high-level overview of confidence levels only. They are in no way meant to be comprehensive, and do not take into account any special cases.

In general, confidence levels are assigned to the cardinality estimates for only two types of operations:

- Single table retrievals
- Joins

Note that the confidence levels the Optimizer reports and the confidence levels recorded in the Level attribute of the QCD StatsRec table are not related in any way.

**Confidence Levels For Single Table Retrieval Operations**

The following table lists the meaning of each confidence level in context and some of the reasons why each confidence level is assigned for single table retrieval operations:

<table>
<thead>
<tr>
<th>Confidence Level</th>
<th>Meaning</th>
<th>Reason</th>
</tr>
</thead>
</table>
| No               | The system has neither Low nor High confidence in the cardinality and distinct value estimates for the relation. The Optimizer pursues conservative strategies to optimize the relevant steps. | Any of the following situations exists:  
- There are no statistics on the column or index sets specified in the predicate.  
- The predicate contains complex expressions for which statistics cannot be collected.  
For example, statistics cannot be collected for either of the following two expressions:  
- \( \text{SUBSTR}(\text{col1}, 5, 10) \)  
- \( \text{CASE} \quad \begin{array} {l} \text{WHEN } x=10 \text{ THEN } x+10 \\ \text{ELSE } x-10 \end{array} \)  
- For an aggregate estimation, there are no statistics on the grouping columns. |

\(^4\) A join operation where there is No confidence on one of the relations, but Low, Index Join, or High on the other, has an overall confidence level of No confidence, even though there are statistics on the join columns of one of the relations. This is because the Optimizer always assumes a confidence level that is equal to the lower confidence level assigned to one of the relations in the join.
## EXPLAIN Confidence Levels

<table>
<thead>
<tr>
<th>Confidence Level</th>
<th>Meaning</th>
<th>Reason</th>
</tr>
</thead>
</table>
| Low              | The system is moderately certain that the estimated cardinality and distinct value estimates for the relation are accurate. The Optimizer pursues more aggressive strategies to optimize the relevant steps. | One of the following states exists for the relation:  
- There are conditions in the query on an index set that has no statistics. Cardinality estimates can be made based on sampling the index set.  
- There are conditions in the query on an index or column set with statistics that is ANDed with conditions on non-indexed columns.  
- There are conditions in the query on an index or column set with statistics that is ORed with other conditions.  
- For an aggregate estimation, there are statistics on single columns of the grouping column set or statistics on some, but not all, of the grouping columns.  

The confidence for single-AMP random AMP statistical samples is always Low. EXPLAIN reports always express No confidence in estimates where there are no statistics, but the system always samples randomly from at least one AMP in such cases when the query is actually executed. |
EXPLAIN Confidence Levels

<table>
<thead>
<tr>
<th>Confidence Level</th>
<th>Meaning</th>
<th>Reason</th>
</tr>
</thead>
</table>
| High             | The system is fairly certain that the estimated cardinality and distinct value estimates for the relation are accurate. The Optimizer pursues more aggressive strategies to optimize the relevant steps. | • Retrieval from a single relation with no predicates:  
  • There are conditions in the query on the primary index and there are statistics on the primary index.  
  • There are conditions in the query on the primary index, but there are no statistics on the primary index.  
  The confidence is high under any of the following situations:  
  • 5 or more AMPs are sampled.  
  • Rows per value are sampled.  
  • No skew is detected.  
  • Retrieval from a single relation with predicates:  
  • There are statistics on the predicate columns or indexes and there is no skew.  
  • There are multiple equality predicates with covering multitable statistics.  
  • For an aggregate estimation, the confidence is high under any of the following situations:  
  • The grouping columns are constants.  
  • The grouping columns have equality predicates.  
  • The grouping columns are all covered by a single multicolumn statistics set.  
  • There is only one grouping column and it has statistics. |

For a retrieval operation from a spool file, the confidence level is the same as the confidence level for the step that generated the spool.

**Confidence Levels For Join Operations**

In the case of join operations, the Optimizer needs to know approximately how many rows will result from each step in each join operation required to perform an SQL request. It uses this information to select an optimal plan for joining the relations. Among other things, the join plan determines the best order for joining the relations. Because you can join as many as 128 tables and single-table views per join clause, it is critical to minimize join cardinality estimation errors to ensure that an optimal query plan is generated.

Keep in mind that errors in join processing are cumulative, so it is critical to minimize the possibilities for errors to occur in join planning. The only way to ensure optimal join processing is to keep fresh statistics on all your indexes and non-index join columns.

5. Such as join method and join geography.
Join cardinality and rows per value estimates nearly always have a lower confidence level than is seen for a single table retrieval under analogous conditions for several reasons, including the following:

- Join cardinality estimates only rate High confidence when there is only a single row in both the left and right relation in the join.
- The confidence level for a join operation never exceeds that of its input relations and assumes the confidence for the relation having the lower confidence.

The following table lists the meaning of each confidence level in context and some of the reasons why each confidence level is assigned for join operations:

<table>
<thead>
<tr>
<th>Confidence Level</th>
<th>Meaning</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>The system has neither Low nor High nor Index Join confidence in the estimated join cardinality.</td>
<td>One (or both) relation in the join does not have statistics on the join columns.</td>
</tr>
</tbody>
</table>
| Low              | The system is moderately certain that the estimated join cardinality is accurate. | • There are statistics on the join columns of both the left and right relations.  
• One relation in the join has Low confidence and the other has any of the following three confidence levels:  
  • Low  
  • High  
  • Index Join |
| High             | The system is fairly certain that the estimated join cardinality is accurate.         | One relation in the join has High confidence and the other has either of the following two confidence levels:  
  • High  
  • Index Join |
| Index Join       | The system is fairly certain that the estimated join cardinality is accurate because of a uniqueness constraint on the join columns. | • There is a unique index on the join columns.  
• There is a foreign key relationship between the two relations in the join.  
Note that because of the way Teradata implements PRIMARY KEY and UNIQUE INDEX constraints, these are essentially two ways of saying the same thing. |

6. All relational joins are binary operations: no more than two relations are ever joined in a single operation. Instead, joins on multiple relations are chained together such that the result of an earlier join operation is spooled and then joined to the next relation in the sequence the Optimizer determines for its join plan. See SQL Request and Transaction Processing for more information about join strategies.
Effect of Random AMP Sampling On Reported Confidence Levels

The analysis of skew is based on the distribution of rows from each of the AMPs, and is a contributing factor to the confidence level expressed by the Optimizer.

Skew analysis computes the expected number of rows per AMP, and if that number is less than 5 percent of the expected number of rows per AMP, the AMP is moved to the skewed AMP list. If the total number of AMPs in the skewed list is less than or equal to 5 percent of the total number of AMPs sampled, then the confidence level is set to Low, otherwise it is set to High.

When statistics are sampled from only one randomly selected AMP, the confidence level is always set to Low.
Terms used in EXPLAIN phrases are described in the following list:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n) partitions of …</td>
<td>Only (n) of the partitions are accessed. (n) is greater than one.</td>
</tr>
<tr>
<td>a single partition of …</td>
<td>Only a single partition is accessed.</td>
</tr>
<tr>
<td>(all_amps)</td>
<td>The operation takes place on all AMPs.</td>
</tr>
<tr>
<td>all-AMPs JOIN step by way of an all-rows scan …</td>
<td>On each AMP on which they reside, spooled rows and/or primary table rows are searched row by row; rows that satisfy the join condition are joined.</td>
</tr>
</tbody>
</table>
| all-AMPs JOIN step by way of a RowHash match scan … | 1 The first row is retrieved from the first table; the hash code for that row is used to locate a row from the second table.  
2 Each row-hash match is located and processed as follows:  
   a The row-hashes are compared. If not equal, the larger row-hash is used to read rows from the other table until a row-hash match is found, or until the table is exhausted.  
   b If match is found, each pair of rows with the same hash code is accessed (one at a time) from the two tables. For each such pair, if the join condition is satisfied, a join result row is produced.  
   c After all rows with the same row-hash are processed from both tables, one more row is read from each table. The row-hashes from these two rows are compared, restarting the compare process. |
### All-AMPS RETRIEVE step by way of an all-rows scan ...

All rows of a table are selected row by row on all AMPs on which the table is stored.

BMSMS (bit map set manipulation step);
intersects the following row id bit maps:

1) The bit map built for ... by way of index # \( n \)...
2) The bit map built for ... by way of index # \( n \)...
...

The resulting bit map is placed in Spool \( n \)...  
BMSMS...

Indicates that two or more bit maps are intersected by ANDing them to form one large bit map.

index # \( n \)...

Identifies, in the order in which they are ANDed, each nonunique secondary index used in the intersection.

resulting bit map is placed in Spool \( n \)...

Identifies the temporary file in which the large bit map produced by the BMSMS is stored to make it available for use in producing the final result.

### All partitions of ...

All partitions are accessed for a primary index access.

### A rowkey-based ...

The join is hash-based by partition (that is, by the rowkey). In this case, there are equality constraints on the partitioning columns and primary index columns. This allows for a faster join since each non-eliminated partition needs to be joined with at most only one other partition.

When the phrase is not given, the join is hash based. That is, there are equality constraints on the primary index columns from which the hash is derived. For a partitioned table, there is some additional overhead in processing the table in hash order.

Note that with either method, the join conditions must still be validated.

### <BEGIN ROW TRIGGER LOOP>

Processing of the trigger action statements defined in the row trigger starts from the current step, step \( n \), of the EXPLAIN text.

All steps from the current step through the step in which the phrase END ROW TRIGGER LOOP for step \( n \) appears constitute the row trigger loop.

### By skipping global aggregation

Teradata Database avoids the cost of redistributing rows and aggregating them globally by skipping global aggregation when you collect multicolumn statistics and any column in the multicolumn set is unique.
<table>
<thead>
<tr>
<th>Phrase</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>by skipping local aggregation</td>
<td>Teradata Database avoids the cost of first writing to spool and then sorting the spooled rows by skipping local aggregation when you collect statistics.</td>
</tr>
<tr>
<td>by the hash code of ([database_name].table_name.column_name).</td>
<td>The spool sort is done on the specified column in row hash order.</td>
</tr>
<tr>
<td>by using cache during local aggregation</td>
<td>Teradata Database avoids the cost of sorting large numbers of input rows by caching output rows during local aggregation and coalescing any duplicate rows in the cache when you collect statistics.</td>
</tr>
<tr>
<td>by way of index # n and the bit map in Spool n ...</td>
<td>The data row associated with the row ID is accessed only if the associated bit is turned on in the bit map (see Usage Notes).</td>
</tr>
<tr>
<td>by way of the sort key in spool field1 ...</td>
<td>Field1 is created to allow a tag sort.</td>
</tr>
<tr>
<td>(compressed columns allowed)</td>
<td>The target spool file can have compressed values.</td>
</tr>
<tr>
<td>computed globally ...</td>
<td>The computation involves all the intermediate spool file data.</td>
</tr>
<tr>
<td>condition ...</td>
<td>An intermediate condition that joins table rows (as compared with the overall join condition).</td>
</tr>
<tr>
<td>duplicated on all AMPs ...</td>
<td>A spool file containing intermediate data that is used to produce a result is copied to all AMPs containing data with which the intermediate data is compared.</td>
</tr>
<tr>
<td>eliminating duplicate rows ...</td>
<td>Duplicate rows can exist in spool files, either as a result of selection of nonunique columns from any table or of selection from a MULTISET table. This is a DISTINCT operation.</td>
</tr>
</tbody>
</table>
### EXPLAIN Request Modifier Terminology

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;END ROW TRIGGER LOOP for step ( n ).&gt;</td>
<td>Delimits the running of the last step in the row trigger loop. Control moves to the next step outside the trigger.</td>
</tr>
<tr>
<td>END TRANSACTION step ...</td>
<td>This indicates that processing is complete and that any locks on the data may be released. Changes made by the transaction are committed.</td>
</tr>
<tr>
<td>enhanced by dynamic partition elimination...</td>
<td>This indicates a join condition where dynamic partition elimination has been used.</td>
</tr>
<tr>
<td>estimated size ...</td>
<td>This value, based on any statistics collected for a table, is used to estimate the size of the spool file needed to accommodate spooled data. If statistics have not been collected for a table, this estimate may be grossly incorrect (see <em>SQL Data Definition Language</em>).</td>
</tr>
<tr>
<td>estimated time ...</td>
<td>This approximate time is based on average times for the suboperations that comprise the overall operation, and the likely number of rows involved in the operation. The accuracy of the time estimate is also affected by the accuracy of the estimated size.</td>
</tr>
<tr>
<td>execute the following steps in parallel ...</td>
<td>This identifies a set of steps that are processed concurrently. The explanatory text immediately following the list describes the execution of each step.</td>
</tr>
<tr>
<td>(group_amps)</td>
<td>the operation is a group AMP operation, meaning that it occurs on more than 1, but fewer than all, AMPs in the system.</td>
</tr>
<tr>
<td>grouping by fieldn ([database_name].table_name.column_expression).</td>
<td>the specified grouping column expression is based on column ( n ) in the specified table.</td>
</tr>
<tr>
<td>in view view_name</td>
<td>the specified [database_name].table_name is accessed by means of the view view_name.</td>
</tr>
</tbody>
</table>
### Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

#### EXPLAIN Request Modifier Terminology

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| join condition ... | The overall constraint that governs the join. In the following request, `employee.empno = department.mgrno` is the overall constraint governing the join.  
   ```sql
   SELECT deptname, name  
   FROM employee, department  
   WHERE employee.empno = department.mgrno;
   ```
   Join conditions are also specified in the ON clause of an outer join or an ANSI SQL standard-formatted inner join. |

| (Last Use) | This term identifies the last reference to a spool file that contains intermediate data that produces a statement’s final result. The file is released following this step. |

| locally on the AMPs ... | That portion of spooled intermediate or result data for which an AMP is responsible is stored on the AMP; it is not duplicated on or redistributed to other AMPs that are processing the same request. |

| lock ... | The Lock Manager places an ACCESS, READ, WRITE, or EXCLUSIVE lock on the database object that is to be accessed by a request. |

| merge join ... | One of the methods of join processing performed by Teradata Database. |

| merge with matched updates and unmatched inserts ... | An AMP step that can do any of the three following operations in a single step:  
   - Both update and insert operations  
   - Insert operations only  
   - Update operations only  
   The step assumes that the source table is always distributed on the join column of the source table, which is specified in the ON clause as an equality constraint with the primary index of the target table and sorted on the RowKey.  
   The step performs a RowKey-based Merge Join internally, identifying source rows that qualify for updating target rows and source rows that qualify for inserts, after which it performs those updates and inserts.  
   This step is very similar to the APPLY phase of MultiLoad because it guarantees that the target table data block is read and written only once during the MERGE operation. |

| nested join ... | One of the methods of join processing performed by Teradata Database. |
### EXPLAIN Request Modifier Terminology

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(no lock required)</td>
<td>The lock required for the operation was acquired in a previous step. This phrase is not always reported for steps where a lock was acquired earlier.</td>
</tr>
<tr>
<td>no residual conditions ...</td>
<td>Rows are selected in their entirety; there are no specific search conditions. All applicable conditions have been applied to the rows.</td>
</tr>
<tr>
<td>of n partitions ...</td>
<td>The optimizer was able to determine that all rows in each of n partitions may be completely deleted. n is greater than one. In some cases, this allows for faster deletion of entire partitions.</td>
</tr>
<tr>
<td>of a single partition ...</td>
<td>The optimizer was able to determine that all rows in a single partition may be completely deleted. In some cases, this allows for faster deletion of the entire partition.</td>
</tr>
<tr>
<td>partial SUM</td>
<td>A partial GROUP BY optimization has been done on the SUM step.</td>
</tr>
<tr>
<td>(partial-covering)</td>
<td>The NUSI in question partially covers the query using a constraint scan. For example, the system can probe a NUSI subtable to get the RowID values for base table rows that might qualify for the query, but it must then also access the underlying base table for the NUSI to obtain the rows that definitely qualify.</td>
</tr>
<tr>
<td>product join ...</td>
<td>One of the methods of join processing performed by Teradata Database.</td>
</tr>
<tr>
<td>redistributed by hash code to all AMPs ...</td>
<td>Given the values of an indexed or nonindexed column, rows are sent to the AMPs that are responsible for storing the rows that use these values as a primary index.</td>
</tr>
<tr>
<td>single-AMP JOIN step by way of the unique primary index ...</td>
<td>A row is selected on a single AMP using the unique primary index for the table. Using a value in the row that hashes to a unique index value in a second table, the first row is joined with a second row located on another AMP.</td>
</tr>
<tr>
<td>single-AMP RETRIEVE by way of unique index # n ...</td>
<td>A single row of a table is selected using a unique secondary index value that hashes to the AMP on which the table row is stored.</td>
</tr>
</tbody>
</table>
### Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

#### EXPLAIN Request Modifier Terminology

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>single-AMP RETRIEVE step by way of the unique primary index ...</strong></td>
<td>A single row of a table is selected using a unique primary index value that hashes to the single AMP on which the data row is stored. Although not explicitly stated in the LOCK portion of the Explain text, a rowhash lock is required because the table is accessed via the unique primary index.</td>
</tr>
</tbody>
</table>

#### SORT/GROUP

- Partial GROUP BY was used to reduce cardinality by simultaneously sorting and grouping.

#### SORT to order Spool n by row hash ...

- Rows in the spool file are sorted by hash code to put them the same order as rows in the primary table, or in another spool file on the same AMP, with which they are to be matched.

#### SORT to order Spool n by row hash and the sort key in spool field1 eliminating duplicate rows ...

- Rows in the spool file are sorted by hash code using a uniqueness sort to eliminate duplicate rows. Uniqueness is based on the data in field1.
  - The contents of field1 depend on the query and may comprise any of the following:
    - A concatenation of all the fields in the spool row (used for queries with SELECT DISTINCT or that involve a UNION, INTERSECT, EXCEPT, or MINUS operation).
    - A concatenation of the row IDs that identify the data rows from which the spool row was formed (used for complex queries involving subqueries).
  - Some other value that uniquely defines the spool row (used for complex queries involving aggregates and subqueries).

#### SORT to order Spool 1 by the sort key in spool field1 ...

- Rows in the spool file are sorted by the value in `field1`. The contents of `field1` are determined by the column set defined in the ORDER BY or WITH...BY clause of the request being processed.
  - For example, “…SORT to partition by rowkey…” or “…SORT to order Spool 1 by the sort key in spool field1 (THU.JI1.b1)…”

#### SORT to partition by rowkey.

- This indicates that a join spool file is joined using a rowkey-based join and sorted to the appropriate partitions.

#### SORT to string_1 by string_2 ...

- Rows are sorted to `string_1` by their `string_2` sort key.
  - For example, “…SORT to partition by rowkey…” or “…SORT to order Spool 1 by the sort key in spool field1 (THU.JI1.b1)…”
SORT to partition Spool \( n \) by rowkey ...

The optimizer determined that a spool file is to be partitioned based on the same partitioning expression as a table to which the spool file is be joined. That is, the spool is to be sorted by rowkey (partition and hash). Partitioning the spool file in this way allows for a faster join with the partitioned table. \( n \) is the spool number.

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>spool ( n ) ...</td>
<td>Identifies a spool file, which is a temporary file used to contain data during an operation.</td>
</tr>
<tr>
<td></td>
<td>• Spool 1 is normally used to hold a result before it is returned to the user.</td>
</tr>
<tr>
<td></td>
<td>• Spools 2, 3, etc., contain intermediate data that produces a result.</td>
</tr>
<tr>
<td>spool_n.Field ...</td>
<td>Identifies the field of the spooled rows that is being used in a join constraint or comparison operation.</td>
</tr>
<tr>
<td></td>
<td>For example:</td>
</tr>
</tbody>
</table>
|                         | \[
|                         | \texttt{personnel.employee.empno = spool\_2.mgrno}                                                                                       |
|                         | \texttt{Statement 1...}                                                                                                                 |

This term refers to the initiating request.

The estimated time for this step is \( nn \) seconds.

The estimated time for the reported Delete, Insert, Join, Merge, Merge Delete, Merge Update, Retrieval, Sum, or Update step in seconds.

Note that the time shown is not clock time, though it should be fairly close to it; you should consider it to be an arbitrary unit of measure that you can use to compare different operations.

the estimated time is \( nn \) seconds.

The estimated time to completion for the entire request in seconds.

The time shown is not clock time, though it should be fairly close to it; you should consider it to be an arbitrary unit of measure that you can use to compare different operations.

The size of Spool \( x \) is estimated with low confidence to be \( y \) rows (\( z \) bytes).

The estimated size of spool \( n \) (which is always reported with Low Confidence - see "Optimizer Confidence Levels" on page 487), where \( m \) is the estimated cardinality of the spool and \( o \) is the estimated number of bytes in the spool.

The spool size in bytes is calculated as follows:

\[
\text{Estimated spool size (bytes)} = \text{Estimated spool cardinality} \times \text{Row size}
\]
### EXPLAIN Request Modifier Terminology

**This query is optimized using type 2 profile m, profileid n.**

The cost profile for this query is a Type 2 profile (see Chapter 2: “Query Rewrite and Optimization” for more information about cost profiles).

- **m** specifies the name of the Type 2 cost profile.
- **n** specifies the name of the cost profile ID.

**two-AMP RETRIEVE step by way of unique index #n …**

A row of a table is selected based on a USI value:

- A single row in the USI subtable is selected using the index value that hashes to the AMP on which the subtable row is stored.
- The hash value in the index row ID determines the AMP on which the data row is stored.

**we do a BMSMS... (bit map set manipulation step)**

BMSMS is a method for handling weakly selective secondary indexes that have been ANDed; NUSI bit mapping.

**we do an ABORT test ...**

An ABORT or ROLLBACK statement was detected.

**we do a SMS (set manipulation step) ...**

Combine rows under control of a UNION, EXCEPT, MINUS, or INTERSECT operator.

**we lock a distinct [database_name]."pseudo table" for [locking_severity] on a [locking_level] for deadlock prevention**

The Lock Manager sets a pseudo table lock for the indicated locking severity at the indicated locking level. See “Pseudo Table Locks” on page 750 for details.

**we lock a distinct [database_name]."pseudo table" for exclusive use on a Row Hash to prevent global deadlock for [database_name.table_name]**

The Lock Manager sets a pseudo table lock for EXCLUSIVE locking severity at the RowHash locking level to prevent global deadlock on database_name.table_name.

**we lock database_name.table_name for exclusive use**

The Lock Manager sets an EXCLUSIVE lock on database_name.table_name.

**we lock database_name.table_name for [locking_severity] on a [locking_level] for deadlock prevention**

The Lock Manager sets a lock for the indicated locking severity at the indicated locking level on database_name.table_name.

**which is duplicated on all AMPs ...**

Relocating data in preparation for a join.
### EXPLAIN Request Modifier Terminology

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>which is redistributed by hash code to all AMPs ...</td>
<td>Relocating data by hash code in preparation for a join.</td>
</tr>
<tr>
<td>which is redistributed by the hash code of ([database_name].table_name.column_expression)</td>
<td>Relocating data by the hash code of the specified column expression based on a column set from table_name in preparation for a join.</td>
</tr>
</tbody>
</table>
EXPLAIN Modifier in Greater Depth

This section examines the usefulness of the EXPLAIN modifier in different situations.

You should always use EXPLAINs to analyze any new queries under development. Subtle differences in the way a query is structured can produce enormous differences in its resource impact and performance while at the same time producing the identical end result.

EXPLAIN modifier terms are defined in “EXPLAIN Request Modifier” on page 484.

The following topics are described here.

• “EXPLAIN: Examples of Complex Queries” on page 504
• “EXPLAIN Request Modifier and Join Processing” on page 512
• “EXPLAIN and Standard Indexed Access” on page 517
• “EXPLAIN and Parallel Steps” on page 520
• “EXPLAIN Request Modifier and Partitioned Primary Index Access” on page 522
• “EXPLAIN Request Modifier and MERGE Conditional Steps” on page 529
• “EXPLAIN and UPDATE (Upsert Form) Conditional Steps” on page 534
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier
EXPLAIN: Examples of Complex Queries

Introduction

The personnel.employee table has a unique primary index defined on the empno column and a nonunique secondary index defined on the name column.

Example 1: SELECT

The EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
SELECT name, deptno
FROM employee
WHERE empno = 10009;
```

Explanation

1) First, we do a single-AMP RETRIEVE step from Personnel.Employee by way of the unique primary index "PERSONNEL.Employee.EmpNo = 10009" with no residual conditions. The estimated time for this step is 0.04 seconds.

   -> The row is sent directly back to the user as the result of statement 1. The total estimated time is 0.04 seconds.

Example 2: SELECT With WHERE on Nonunique Index

The WHERE condition in this example is based on a column that is defined as a nonunique index. Note that the system places a READ lock on the table.

The EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
SELECT empno, deptno
FROM employee
WHERE name = 'Smith T';
```

Explanation

1) First, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to prevent global deadlock for PERSONNEL.employee.
2) Next, we lock PERSONNEL.employee for read.
3) We do an all-AMPS RETRIEVE step from PERSONNEL.employee by way of index # 4 "PERSONNEL.employee.Name = 'Smith T '" with no residual conditions into Spool 1 (group_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with low confidence to be 33,563 rows (1,443,209 bytes). The estimated time for this step is 0.34 seconds.
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 0.34 seconds.

Example 3: SELECT With WHERE on Unique Secondary Index

Assume that the Employee table has another column (socsecno), where socsecno is defined as a unique secondary index.
If the WHERE condition is based on this column, then the EXPLAIN modifier generates the following response for this request.

EXPLAIN
SELECT name, empno
FROM employee
WHERE socsecno = '123456789';

Explanation
-----------------------------------------------------
1) First, we do a two-AMP RETRIEVE step from PERSONNEL.Employee by way of unique index # 20 "PERSONNEL.Employee.socSecNo = 123456789" with no residual conditions. The estimated time for this step is 0.09 seconds.

Example 4: SELECT With WHERE Based On a Join

In this example, the WHERE clause defines an equality constraint that governs a join.

The rows of the Department table are copied to a spool file for use in the join operation.

The EXPLAIN modifier generates the following response for this request.

EXPLAIN
SELECT DeptName, Name
FROM Employee, Department
WHERE Employee.EmpNo = Department.MgrNo;

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.department 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 8 rows. The estimated time for this step is 0.11 seconds.
5) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of a RowHash match scan, which is joined to PERSONNEL.employee. Spool 2 and PERSONNEL.employee are joined using a merge join, with a join condition of ("PERSONNEL.employee.EmpNo = Spool 2.EmpNo"). The result goes into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated to be 8 rows. The estimated time for this step is 0.07 seconds.

Example 5: SELECT With WHERE Based on Subquery

In this example, the constraint that governs the join is defined by a subquery predicate and the ORDER BY clause specifies a sorted result.

The EXPLAIN modifier generates the following response for this request.

EXPLAIN
SELECT Name, EmpNo
FROM Employee
WHERE EmpNo IN
(SELECT EmpNo
 FROM Charges)
ORDER BY Name;

Explanation
--------------------------------------------------------------------
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN: Examples of Complex Queries

Example 6: Recursive Query

The following example demonstrates a recursive query:

```sql
EXPLAIN WITH RECURSIVE temp_table (employee_number, depth) AS
  ( SELECT root.employee_number, 0 as depth
    FROM Employee root
    WHERE root.manager_employee_number = 801 
    UNION ALL
    SELECT indirect.employee_number, seed.depth+1 as depth
    FROM temp_table seed, Employee indirect
    WHERE seed.employee_number = indirect.manager_employee_number
    AND depth <= 20
  )
SELECT employee_number, depth FROM temp_table;
```

EXPLAIN generates the following report for this request:

Explanation
---------------------------------------------------------------------------
1) First, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to prevent global deadlock for PERSONNEL.root.
2) Next, we lock PERSONNEL.root for read.
3) We do an all-AMPs RETRIEVE step from PERSONNEL.root by way of an all-rows scan with a condition of ("PERSONNEL.root.manager_employee_number = 801") into Spool 3 (all_amps), which is built locally on the AMPs. The size of Spool 3 is estimated with no confidence to be 1 row. The estimated time for this step is 0.06 seconds.
4) We do an all-AMPs RETRIEVE step from Spool 3 by way of an all-rows scan into Spool 2 (all_amps), which is built locally on the AMPs. The size of Spool 2 is estimated with no confidence to be 1 row. The estimated time for this step is 0.07 seconds.
5) We execute the following steps in parallel.
   1) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan with a condition of (("DEPTH <= 20) AND (NOT (EMPLOYEE_NUMBER IS NULL ))") into Spool 4 (all_amps), which is duplicated on all AMPs. The size of Spool 4 is estimated with no confidence to be 8 rows. The estimated time for this step is 0.03 seconds.
   2) We do an all-AMPs RETRIEVE step from PERSONNEL.indirect by way of an all-rows scan with a condition of ("NOT (PERSONNEL.indirect.manager_employee_number IS NULL)"") into Spool 5 (all_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with no confidence to be 8 rows. The estimated time for this step is 0.01 seconds.
6) 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) by way of an all-rows scan. Spool 4 and Spool 5 are joined using a single partition hash join, with a join condition of ("EMPLOYEE_NUMBER = manager_employee_number"). The result goes into Spool 6 (all_amps), which is built locally on the AMPs. The size of Spool 6 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.07 seconds.

The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0 hours and 0.23 seconds.
time for this step is 0.08 seconds.
7) We do an all-AMPs RETRIEVE step from Spool 6 (Last Use) by way of an all-rows scan into Spool 3 (all_amps), which is built locally on the AMPs. The size of Spool 3 is estimated with no confidence to be 4 rows. The estimated time for this step is 0.07 seconds. If one or more rows are inserted into spool 3, then go to step 4.

8) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of an all-rows scan into Spool 7 (all_amps), which is built locally on the AMPs. The size of Spool 7 is estimated with no confidence to be 142 rows. The estimated time for this step is 0.07 seconds.

9) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

-> The contents of Spool 7 are sent back to the user as the result of statement 1. The total estimated time is 0.38 seconds.

Step 3 indicates the processing of the seed statement inside the recursive query and produces the initial temporary result set.

Steps 4 through 7 correspond to the processing of the recursive statement inside the recursive query and repeat until no new rows are inserted into the temporary result set. Although steps 4 through 7 indicate a static plan, each iteration can produce spools with varying cardinalities; thus, the level of confidence for the spool size in these steps is set to no confidence.

Step 8 indicates the processing of the final result that is sent back to the user.

---

**Example 7: Large Table SELECT With More Complex Condition**

Assume that a table named Main is very large and that its columns named *numa*, *numb*, *kind*, and *event* are each defined as nonunique secondary indexes. The request in this example uses these indexes to apply a complex conditional expression. The EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
SELECT COUNT(*)
FROM main
WHERE numa = '101'
AND numb = '02'
AND kind = 'B'
AND event = '001';
```

The response indicates that bit mapping would be used.

---

**Explanation**

1) First, we lock TESTING.Main for read.
2) Next, we do a BMSMS (bit map set manipulation) step that intersects the following row id bit maps:
   1) The bit map built for TESTING.Main by way of index # 12 "TESTING.Main.Kind = 'B'".
   2) The bit map build for TESTING.Main by way of index # 8 "TESTING.Main.NumB = '02'".
   3) The bit map built for TESTING.Main by way of index # 16 "TESTING.Main.Event = '001'".
   The resulting bit map is placed in Spool 3. The estimated time for this step is 17.77 seconds.
3) We do a SUM step to aggregate from TESTING.Main by way of index # 4 "TESTING.Main.NumA = '101'" and the bit map in Spool 3 (Last Use) with a residual condition of ("(TESTING.Main.NumB = '02') and ((TESTING.Main.Kind = 'B') and (TESTING.Main.Event = '001'))"), Aggregate Intermediate Results are computed globally, then placed in Spool 2.
4) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of an all-rows scan into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated to be 142 rows. The estimated time for this step is 0.11 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 a result of statement 1.
Example 8: Implicit Multistatement INSERT Transaction

In the following BTEQ multistatement request, which is treated as an implicit transaction, the statements are processed concurrently.

In Teradata session mode, the EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
INSERT Charges (30001, 'AP2-0004', 890825, 45.0)
; INSERT Charges (30002, 'AP2-0004', 890721, 12.0)
; INSERT Charges (30003, 'AP2-0004', 890831, 37.5)
; INSERT Charges (30004, 'AP2-0004', 890825, 11.0)
; INSERT Charges (30005, 'AP2-0004', 890831, 24.5)
; INSERT Charges (30006, 'AP2-0004', 890825, 40.5)
; INSERT Charges (30007, 'AP2-0004', 890721, 32.0)
; INSERT Charges (30008, 'AP2-0004', 890825, 41.5)
; INSERT Charges (30009, 'AP2-0004', 890721, 22.0) ;
```

Explanation

1) First, we execute the following steps in parallel.
   1) We do an INSERT into PERSONNEL.charges.
   2) We do an INSERT into PERSONNEL.charges.
   3) We do an INSERT into PERSONNEL.charges.
   4) We do an INSERT into PERSONNEL.charges.
   5) We do an INSERT into PERSONNEL.charges.
   6) We do an INSERT into PERSONNEL.charges.
   7) We do an INSERT into PERSONNEL.charges.
   8) We do an INSERT into PERSONNEL.charges.
   9) We do an INSERT into PERSONNEL.charges.
10) We do an INSERT into PERSONNEL.charges.

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.
   No rows are returned to the user as the result of statement 2.
   No rows are returned to the user as the result of statement 3.
   No rows are returned to the user as the result of statement 4.
   No rows are returned to the user as the result of statement 5.
   No rows are returned to the user as the result of statement 6.
   No rows are returned to the user as the result of statement 7.
   No rows are returned to the user as the result of statement 8.
   No rows are returned to the user as the result of statement 9.
   No rows are returned to the user as the result of statement 10.

Example 9: ANSI Versus Teradata Session Mode

This example shows the EXPLAIN differences between running the session in ANSI versus Teradata session modes:

```
EXPLAIN
UPDATE Employee
SET deptno = 650
WHERE deptno = 640;
```
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN: Examples of Complex Queries

In ANSI mode, EXPLAIN generates the following response for this request:

Explaination
--------------------------------------------------------------------
1) First, we lock a distinct PERSONNEL."pseudo table" for write on a RowHash to prevent global deadlock for PERSONNEL.employee. 
2) Next, we lock PERSONNEL.employee for write. 
3) We do an all-AMPs UPDATE from PERSONNEL.employee by way of an all-rows scan with a condition of ("PERSONNEL.employee.DeptNo = 640").
   -> No rows are returned to the user as the result of statement 1.

In Teradata session mode, EXPLAIN generates this response for the same request.

Explaination
--------------------------------------------------------------------
1) First, we lock a distinct PERSONNEL."pseudo table" for write on a RowHash to prevent global deadlock for PERSONNEL.employee.
2) Next, we lock PERSONNEL.employee for write. 
3) We do an all-AMPs UPDATE from PERSONNEL.employee by way of an all-rows scan with a condition of ("PERSONNEL.employee.DeptNo = 640").
4) 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.

In ANSI session mode the transaction is not committed, therefore it is not ended, whereas in Teradata session mode, no COMMIT is required to end the transaction.

Example 10: Row Trigger With Looping Trigger Action

In this example three tables t1, t2 and t3, and an AFTER row trigger with t1 as the subject table, are created. The trigger action modifies tables t2 and t3.

The EXPLAIN text for the INSERT operation, which is part of the trigger action, specifically marks the beginning and ending of the row trigger loop. The relevant phrases in the EXPLAIN report are highlighted in boldface type:

The DDL statements for creating the tables and the trigger are as follows:

CREATE TABLE t1(i INTEGER, j INTEGER);
CREATE TABLE t2(i INTEGER, j INTEGER);
CREATE TABLE t3(i INTEGER, j INTEGER);

CREATE TRIGGER g1 AFTER INSERT ON t1
FOR EACH ROW
(
    UPDATE t2 SET j = j+1;
    DELETE t2;
    DELETE t3;
);

The EXPLAIN text reports the steps used to process the following INSERT … SELECT statement:

EXPLAIN INSERT t1 SELECT * FROM t3;

*** Help information returned. 50 rows.
*** Total elapsed time was 1 second.
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN: Examples of Complex Queries

Explanation
----------------------------------------------------------------
1) First, we lock a distinct EXP_TST1."pseudo table" for write on a RowHash to prevent
global deadlock for EXP_TST1.t3.
2) Next, we lock a distinct EXP_TST1."pseudo table" for write on a RowHash to prevent
global deadlock for EXP_TST1.t2.
3) We lock a distinct EXP_TST1."pseudo table" for write on a RowHash to prevent global
deadlock for EXP_TST1.t1.
4) We lock EXP_TST1.t3 for write, we lock EXP_TST1.t2 for write, and we lock
EXP_TST1.t1 for write.
5) We do an all-AMPs RETRIEVE step from EXP_TST1.t1 by way of an all-rows scan with no
residual conditions into Spool 3 (all_amps), which is redistributed by hash code to
all AMPs. Then we do a SORT to order Spool 3 by the sort key in spool field
eliminating duplicate rows. The size of Spool 3 is estimated with low confidence to
be 2 rows. The estimated time for this step is 0.03 seconds.
6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan
into Spool 2 (all_amps), which is duplicated on all AMPs. The size of Spool 2 is
estimated with no confidence to be 4 rows.
7) We do an all-AMPs JOIN step from EXP_TST1.t3 by way of an all-rows scan with no
residual conditions, which is joined to Spool 2 (Last Use). EXP_TST1.t3 and Spool 2
are joined using an exclusion product join, with a join condition of "((j =
EXP_TST1.t3.j) OR (j IS NULL)) AND (((j - EXP_TST1.t3.j) OR (EXP_TST1.t3.j IS
NULL)) AND (((i - EXP_TST1.t3.i) OR (EXP_TST1.t3.i IS NULL)) AND (((i -
EXP_TST1.t3.i) OR (i IS NULL))))))" where unknown comparison will be ignored. The
result goes into Spool 1 (all_amps), which is built locally on the AMPs. The size of
Spool 1 is estimated with index join confidence to be 2 rows. The estimated time for
this step is 0.03 seconds.
8) We do an all-AMPs RETRIEVE step from Spool 1 (Last Use) by way of an all-rows scan
into Spool 4 (all_amps), which is redistributed by hash code to all AMPs. Then we do
a SORT to order Spool 4 by row hash. The size of Spool 4 is estimated with index
join confidence to be 2 rows. The estimated time for this step is 0.04 seconds.
9) We do an all-AMPs MERGE into EXP_TST1.t1 from Spool 4 (Last Use).
10) <BEGIN ROW TRIGGER LOOP>
    we do an all-AMPs UPDATE from EXP_TST1.t2 by way of an all-rows scan with no
    residual conditions.
11) We do an all-AMPs DELETE from EXP_TST1.t2 by way of an all-rows scan with no
    residual conditions.
12) We do an all-AMPs DELETE from EXP_TST1.t3 by way of an all-rows scan with no
    residual conditions.
<br /> <END ROW TRIGGER LOOP> for step 10.
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.
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN Request Modifier and Join Processing

Introduction

The following descriptions are generated by EXPLAIN when applied to sample join requests.

Each explanation is preceded by the request syntax and is followed by a listing of any new terminology found in the display.

Example 1: Product Join

This request has a WHERE condition based on the value of a unique primary index, which produces efficient processing even though a product join is used (compare with the merge join used to process the request in “Example 2: Merge Join” on page 513).

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

This request returns the following EXPLAIN report:

```
Explanation
--------------------------------------------------------
1) First, we lock PERSONNEL.Charges for read.
2) Next, we do a single-AMP RETRIEVE step from PERSONNEL.Project by way of the unique primary index "PERSONNEL.Project.Proj_Id = 'ENG-0003'" 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 0.07 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 PERSONNEL.Charges with a condition of ("PERSONNEL.Charges.Proj_Id = 'ENG-0003'"). Spool 2 and PERSONNEL.Charges are joined using a product join, with a join condition of ("PERSONNEL.Charges.WkEnd > Spool_2.DueDate"). The result goes into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated to be 6 rows. The estimated time for this step is 0.13 seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
-> The contents of Spool 1 are back to the user as the result of statement 1. The total estimated time is 0.20 seconds.
```

Terminology

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-AMP RETRIEVE step from ... by way of the unique primary index</td>
<td>A single row of a table is selected using a unique primary index that hashes to the single AMP on which the row is stored.</td>
</tr>
<tr>
<td>duplicated on all AMPs</td>
<td>The contents of a spool file, selected from the first table involved in the join, is replicated on all the AMPs that contain another table involved in the join.</td>
</tr>
</tbody>
</table>
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN Request Modifier and Join Processing

Example 2: Merge Join

This request returns the following EXPLAIN report:

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

Explanation

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-AMPs JOIN step ... by way of an all-rows scan</td>
<td>Table rows are searched row by row on each AMP on which they reside. Rows that satisfy the join condition are joined to the spooled row or rows.</td>
</tr>
<tr>
<td>condition of ...</td>
<td>An intermediate condition used to qualify the joining of selected rows with spooled rows (as compared with an overall join condition).</td>
</tr>
<tr>
<td>product join</td>
<td>One of the types of join processing performed by Teradata Database.</td>
</tr>
</tbody>
</table>

Terminology

New terminology in this explanation is defined as follows.

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge join</td>
<td>One of the types of join processing performed by the Teradata Database.</td>
</tr>
</tbody>
</table>
Example 3: Hash Join

This request returns the following EXPLAIN report:

```
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.

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%

Terminology

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash join</td>
<td>One of the types of join processing performed by Teradata Database.</td>
</tr>
</tbody>
</table>

Example 4: Nested Join

This request returns the following EXPLAIN report:

```
EXPLAIN
SELECT DeptName, Name, YrsExp
FROM Employee, Department
WHERE Employee.EmpNo = Department.MgrNo
AND Department.DeptNo = 100;
```
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN Request Modifier and Join Processing

Explanation
-------------------------------------------------------
1) First, we do a single AMP JOIN step from PERSONNEL.Department by way of the unique primary index "PERSONNEL.Department.DeptNo = 100" with no residual condition which is joined to PERSONNEL.Employee by way of the unique primary index "PERSONNEL.Employee.EmpNo = PERSONNEL.Department.MgrNo". PERSONNEL.Department and PERSONNEL.Employee are joined using a nested join. The result goes into Spool 1, which is built locally on that AMP. The size of Spool 1 is estimated to be 1 rows. The estimated time for this step is 0.10 seconds.

> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.10 seconds.

Terminology

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-AMP JOIN step from …</td>
<td>A single row on one AMP is selected using the value of a unique index</td>
</tr>
<tr>
<td>by way of the unique primary index</td>
<td>defined for its table. Using the value in that row which hashes to a</td>
</tr>
<tr>
<td></td>
<td>unique index value in a row of a second table on another AMP, the first</td>
</tr>
<tr>
<td></td>
<td>row is joined with the second row. (Note that when a table is accessed</td>
</tr>
<tr>
<td></td>
<td>by its unique primary index, the need for a rowhash lock on the table is</td>
</tr>
<tr>
<td></td>
<td>implied, even though it is not explicitly stated in the Explain text.)</td>
</tr>
<tr>
<td>nested join</td>
<td>One of the types of join processing performed by the Teradata Database.</td>
</tr>
</tbody>
</table>

Example 5: Exclusion Merge Join

The request in this example selects columns only from the primary table.

If an additional column was selected from the table being joined via the embedded select (for example, if the request was SELECT Name, DeptNo, Loc FROM Employee, Department), the result would be a Cartesian product.

EXPLAIN
SELECT Name, DeptNo
FROM Employee
WHERE DeptNo NOT IN
(SELECT DeptNo
 FROM Department
 WHERE Loc = 'CHI')
ORDER BY Name ;
This request returns the following EXPLAIN report:

Explanation

1) First, we lock PERSONNEL.Department for read, and we lock PERSONNEL.Employee for read.
2) Next, we execute the following steps in parallel.
   1) We do an all-AMPs RETRIEVE step from PERSONNEL.Department by way of an all-rows scan with a condition of (“PERSONNEL.Department.Loc = ’CHI’”) into Spool 2, which is redistributed by hash code to all AMPs. Then we do a SORT to order Spool 2 by row hash and the sort key in spool field1 eliminating duplicate rows. The size of Spool 2 is estimated to be 4 rows. The estimated time for this step is 0.07 seconds.
   2) We do an all-AMPs RETRIEVE step from PERSONNEL.Employee by way of an all-rows scan with no residual conditions 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 8 rows. The estimated time for this step is 0.10 seconds.
   3) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an all-rows scan, which is joined to Spool 2 (Last Use). Spool 3 and Spool 2 are joined using an exclusion merge join, with a join condition of (“Spool_3.DeptNo = Spool_2.DeptNo”). The result goes into Spool 1, which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1. The size of Spool 1 is estimated to be 8 rows. The estimated time for this step is 0.13 seconds.
   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 0.23 seconds.

Terminology

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
</table>
| SORT to order Spool 2 by row hash and the sort key in spool field1 eliminating duplicate rows... | Rows in the spool file are sorted by hash code using a uniqueness sort to eliminate duplicate rows. Uniqueness is based on the data in field1. The contents of field1 depend on the query and may comprise any of the following:  
   • A concatenation of all the fields in the spool row (used for queries with SELECT DISTINCT or that involve a UNION, INTERSECT, or MINUS operation).  
   • A concatenation of the row IDs that identify the data rows from which the spool row was formed (used for complex queries involving subqueries).  
   Some other value which uniquely defines the spool row (used for complex queries involving aggregates and subqueries). |
| SORT to order Spool 1 by the sort key in spool field1 | This last sort is in response to the “ORDER BY” clause attached to the primary SELECT request. |
| exclusion merge join | One of the types of join processing performed by Teradata Database. |
EXPLAIN and Standard Indexed Access

Introduction

The EXPLAIN modifier is useful in determining whether the indexes defined for a table are properly defined, useful, and efficient.

As illustrated in the preceding join examples, EXPLAIN identifies any unique indexes that may be used to process a request.

When conditional expressions use nonunique secondary indexes, EXPLAIN also indicates whether the data rows are retrieved using spooling or bit mapping.

This feature is illustrated in the following examples. Compare the two requests and their corresponding EXPLAIN descriptions.

Note that in the first request the table is small (and that DeptNo, Salary, YrsExp, and Edlev have been defined as separate, nonunique indexes), so a full-table scan is used to access the data rows.

In the second request, however, the table is extremely large. Because of this, the Optimizer determines that bit mapping of the sub-table row IDs is the faster retrieval method.

Example 1: Full-Table Scan

This request returns the following EXPLAIN report:

```
EXPLAIN
SELECT COUNT(*)
FROM Employee
WHERE DeptNo = 500
AND Salary > 25000
AND YrsExp >= 3
AND EdLev >= 12;
```

Explanation
--------------------------------------------------------
1) First, we lock PERSONNEL.Employee for read.
2) Next, we do a SUM step to aggregate from PERSONNEL.Employee by way of an all-rows scan with a condition of
   "((PERSONNEL.Employee.DeptNo = 500) AND
    ((PERSONNEL.Employee.Salary > 25000.00) AND
    ((PERSONNEL.Employee.YrsExp >= 3) AND
    (PERSONNEL.Employee.EdLev >= 12 )))".
Aggregate Intermediate Results are computed globally, then placed in Spool 2.
3) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of an all-rows scan into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated to be 1 rows. The estimated time for this step is 0.06 seconds.
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.
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN and Standard Indexed Access

Terminology

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM step to aggregate</td>
<td>The table is searched row by row, the qualifying rows are counted for each AMP on which they were found, and each count is held in a local spool file.</td>
</tr>
<tr>
<td>computed globally</td>
<td>The final computation involves all the intermediate spool-file data.</td>
</tr>
</tbody>
</table>

Example 2: Indexed Access With Bit Mapping

This request returns the following EXPLAIN report:

```sql
EXPLAIN
SELECT COUNT(*)
FROM Main
WHERE NumA = '101'
AND NumB = '02'
AND Kind = 'B'
AND Event = '001';
```

Explanation

1) First, we lock TESTING.Main for read.
2) Next, we do a BMSMS (bit map set manipulation) step that intersects the following row id bit maps:
   1) The bit map built for TESTING.Main by way of index # 12 "TESTING.Main.Kind = 'B'".
   2) The bit map build for TESTING.Main by way of index # 8 "TESTING.Main.NumB = '02'".
   3) The bit map built for TESTING.Main by way of index # 16 "TESTING.Main.Event = '001'".

   The resulting bit map is placed in Spool 3. The estimated time for this step is 17.77 seconds.
3) We do a SUM step to aggregate from TESTING.Main by way of index # 4 "TESTING.Main.NumA = '101'" and the bit map in Spool 3 (Last Use) with a residual condition of ("(TESTING.Main.NumB = '02') and ((TESTING.Main.Kind = 'B') and TESTING.Main.Event = '001'))"). Aggregate Intermediate Results are computed globally, then placed in Spool 2.
4) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of an all-rows scan into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated to be 20 rows. The estimated time for this step is 0.11 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 a result of statement 1.
**Terminology**

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A BMSMS (bit map set manipulation step) that intersects the following Row-Id bit maps:</td>
<td>On each nonunique secondary index sub-table, each data row ID is assigned a number from 0-32767. This number is used as an index into a bit map in which the bit for each qualifying row is turned on. BMSMS indicates that the intersection of sets of qualifying rows is computed by applying the logical AND operation to the bitmap representation of the sets.</td>
</tr>
<tr>
<td>1 The bit map built for … by way of index # n …</td>
<td></td>
</tr>
<tr>
<td>2 The bit map built for … by way of index # n …</td>
<td></td>
</tr>
<tr>
<td>residual condition</td>
<td>Selected rows are further qualified by one or more conditional expressions.</td>
</tr>
</tbody>
</table>
EXPLAIN and Parallel Steps

Introduction

The EXPLAIN modifier also reports whether or not statements will be processed in parallel.

Parallel Steps

The steps that can be executed in parallel are numbered, indented in the explanatory text, and preceded by the following message:

... we execute the following steps in parallel.

Parallel steps can be used to process a request submitted in a transaction (which can be a user-generated transaction, a multi-statement request, a macro, or a solitary statement that affects multiple rows).

Up to 20 parallel steps can be processed per request if channels are not required, such as a request with an equality constraint based on a primary index value.

Up to 10 channels can be used for parallel processing when a request is not constrained to a primary index value.

For example, a non-primary-constrained request that does not involve redistribution of rows to other AMPs, such as a SELECT or UPDATE, requires only two channels. A request that does involve row redistribution, such as a join or an INSERT … SELECT, requires four channels.

Example

The following BTEQ request is structured as a single transaction, and thus generates parallel-step processing.

In Teradata session mode, the transaction is structured as follows.

    BEGIN TRANSACTION
    ;INSERT Department (100,'Administration','NYC',10011)
    ;INSERT Department (600,'Manufacturing','CHI',10007)
    ;INSERT Department (500,'Engineering', 'ATL',10012)
    ;INSERT Department (600, 'Exec Office','NYC', 10018)
    ; END TRANSACTION ;

In ANSI session mode, the transaction is structured as follows.

    INSERT Department (100, 'Administration', 'NYC', 10011)
    ;INSERT Department (600, 'Manufacturing', 'CHI', 10007)
    ;INSERT Department (500, 'Engineering', 'ATL', 10012)
    ;INSERT Department (600, 'Exec Office', 'NYC', 10018)
    ;COMMIT ;

If you issue an EXPLAIN modifier against these transactions, the request returns the identical explanation in either mode, except that the last line is not returned for an ANSI mode transaction.
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN and Parallel Steps

Explanation

1) First, we execute the following steps in parallel.
   1) We do an INSERT into PERSONNEL.Department
   2) We do an INSERT into PERSONNEL.Department
   3) We do an INSERT into PERSONNEL.Department
   4) We do an INSERT into PERSONNEL.Department

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.
   No rows are returned to the user as the result of statement 2.
   No rows are returned to the user as the result of statement 3.
   No rows are returned to the user as the result of statement 4.
   No rows are returned to the user as the result of statement 6.

SQL Request and Transaction Processing
EXPLAIN Request Modifier and Partitioned Primary Index Access

Introduction

EXPLAIN reports indicate partition accesses, deletions, joins, and eliminations performed during query optimization.

Example 1

The following example demonstrates an EXPLAIN report for accessing a subset of partitions for a SELECT statement. The relevant phrase is highlighted in boldface type.

```
CREATE TABLE t1
    (a INTEGER,
     b INTEGER)
PRIMARY INDEX(a) PARTITION BY RANGE_N(
    b BETWEEN 1 AND 10 EACH 1);

EXPLAIN SELECT *
FROM t1
WHERE b > 2;
```

1) First, we lock a distinct mws."pseudo table" for read on a RowHash to prevent global deadlock for mws.t1.
2) Next, we lock mws.t1 for read.
3) We do an all-AMPs RETRIEVE step from 8 partitions of mws.t1 and with a condition of ("mws.t1.b > 2") into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 1 row. The estimated time for this step is 0.14 seconds.
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 0.14 seconds.

Terminology

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$ partitions of</td>
<td>Only $n$ of the partitions are accessed, where $n &gt; 1$. In this case, $n = 8$.</td>
</tr>
</tbody>
</table>
Example 2

The following example demonstrates an EXPLAIN report for a SELECT with an equality constraint on the partitioning column. The relevant phrase is highlighted in boldface type. The report indicates that all rows in a single partition are scanned across all AMPs.

```sql
CREATE TABLE t1
(a INTEGER,
b INTEGER)
PRIMARY INDEX(a) PARTITION BY RANGE_N(
b BETWEEN 1 AND 10 EACH 1);

EXPLAIN SELECT *
FROM t1
WHERE t1.b = 1;
```

1) First, we lock a distinct mws."pseudo table" for read on a RowHash to prevent global deadlock for mws.t1.
2) Next, we lock mws.t1 for read.
3) We do an all-AMPs RETRIEVE step from a single partition of mws.t1 with a condition of ("mws.t1.b = 1") into Spool 1 (group amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.15 seconds.
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 0.15 seconds.

Terminology

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a single partition of</td>
<td>Only one partition is accessed in processing this query.</td>
</tr>
</tbody>
</table>

Example 3

The following example demonstrates an EXPLAIN report for partitioned primary index access without any constraints on the partitioning column. The relevant phrase is in boldface type. The report indicates that all partitions are accessed by way of the primary index on a single AMP.

```sql
CREATE TABLE t1
(a INTEGER,
b INTEGER)
PRIMARY INDEX(a) PARTITION BY RANGE_N(
b BETWEEN 1 AND 10 EACH 1);

EXPLAIN SELECT *
FROM t1
WHERE t1.a = 1;
```
1) First, we do a single-AMP RETRIEVE step from all partitions of mws2.t1 by way of the primary index "mws2.t1.a = 1" with no residual conditions into Spool 1 (one_amp), which is built locally on that AMP. The size of Spool 1 is estimated with low confidence to be 2 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.

**Terminology**

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>all partitions</td>
<td>All partitions are accessed for primary index access in processing this query.</td>
</tr>
</tbody>
</table>

**Example 4**

The following example demonstrates the processing of a SELECT statement without any partition elimination. The phrase "n partitions of" does not occur in the report.

```sql
CREATE TABLE t1
(a INTEGER,
 b INTEGER)
 PRIMARY INDEX(a) PARTITION BY RANGE_N(
   b BETWEEN 1 AND 10 EACH 1);

EXPLAIN SELECT * FROM t1 WHERE b > -1;
```

1) First, we lock a distinct mws."pseudo table" for read on a RowHash to prevent global deadlock for mws.t1.
2) Next, we lock mws.t1 for read.
3) We do an all-AMPs RETRIEVE step from mws.t1 by way of an all-rows scan with a condition of ("mws.t1.b > -1") into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 1 row. The estimated time for this step is 0.15 seconds.
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 0.15 seconds.

**Example 5**

Two steps are generated to perform the partial and full partition deletions, respectively, as demonstrated in the following EXPLAIN reports. The relevant phrases are highlighted in boldface type.

```sql
CREATE TABLE t2 (
  a INTEGER,
  b INTEGER)
 PRIMARY INDEX(a) PARTITION BY RANGE N(
   b BETWEEN 1 AND 10 EACH 2);
```
EXPLAIN DELETE
    FROM t2
    WHERE b BETWEEN 4 AND 7;

1) First, we lock a distinct mws."pseudo table" for write on a
   RowHash to prevent global deadlock for mws.t2.
2) Next, we lock mws.t2 for write.
3) We do an all-AMPs DELETE from 2 partitions of mws.t2
   with a condition of ("(mws.t2.b <= 7) AND (mws.t2.b >= 4)").
4) We do an all-AMPs DELETE of a single partition from mws.t2 with a
   condition of ("(mws.t2.b <= 7) AND (mws.t2.b >= 4)").
5) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
-> No rows are returned to the user as the result of statement 1.

EXPLAIN DELETE
    FROM t2
    WHERE b BETWEEN 4 AND 8;

1) First, we lock a distinct mws."pseudo table" for write on a
   RowHash to prevent global deadlock for mws.t2.
2) Next, we lock mws.t2 for write.
3) We do an all-AMPs DELETE from a single partition of mws.t2
   with a condition of ("(mws.t2.b <= 8) AND (mws.t2.b >= 4)").
4) We do an all-AMPs DELETE of 2 partitions from mws.t2 with a
   condition of ("(mws.t2.b <= 8) AND (mws.t2.b >= 4)").
5) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
-> No rows are returned to the user as the result of statement 1.

Terminology

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>of a single partition</td>
<td>The optimizer determined that all rows in a single partition can be deleted. In some cases, this allows for faster deletion of the entire partition.</td>
</tr>
<tr>
<td>of n partitions</td>
<td>The optimizer determined that all rows in each of n partitions can be deleted, where n &gt; 1. In some cases, this allows for faster deletion of entire partitions.</td>
</tr>
</tbody>
</table>

Example 6

The following example demonstrates a spool with a partitioned primary index and a rowkey-based join. The relevant phrases are highlighted in boldface type.

```sql
CREATE TABLE t3 (  
    a INTEGER,  
    b INTEGER)  
PRIMARY INDEX(a);

CREATE TABLE t4 (  
    a INTEGER,  
    b INTEGER)  
PRIMARY INDEX(a)  
PARTITION BY b;

EXPLAIN SELECT *  
FROM t3, t4  
WHERE t3.a = t4.a  
AND    t3.b = t4.b;
```
1) First, we lock a distinct mws."pseudo table" for read on a RowHash to prevent global deadlock for mws.t3.
2) Next, we lock a distinct mws."pseudo table" for read on a RowHash to prevent global deadlock for mws.t4.
3) We lock mws.t3 for read, and we lock mws.t4 for read.
4) We do an all-AMPs RETRIEVE step from mws.t3 by way of an all-rows scan with a condition of "(NOT (mws.t3.b IS NULL )) AND (NOT (mws.t3.a IS NULL ))" into Spool 2 (all_amps), which is built locally on the AMPs. Then we do an all-AMPs **SORT to partition Spool 2 by rowkey**. The size of Spool 2 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.03 seconds.
5) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of a RowHash match scan, which is joined to mws.t4 with a condition of "NOT (mws.t4.a IS NULL)". Spool 2 and mws.t4 are joined using a rowkey-based merge join, with a join condition of "(a = mws.t4.a) AND (b = mws.t4.b)". The input table mws.t4 will not be cached in memory. The result goes into Spool 1 (all_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 1 row. The estimated time for this step is 0.20 seconds.
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 0.22 seconds.

**Terminology**

New terminology in this explanation is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SORT to partition Spool n by rowkey.</strong></td>
<td>The optimizer determined that a spool file is to be partitioned based on the same partitioning expression as a table to which the spool file is be joined. That is, the spool is to be sorted by rowkey (partition and hash). Partitioning the spool file in this way enables a faster join with the partitioned table. n is the spool number.</td>
</tr>
<tr>
<td><strong>a rowkey-based</strong></td>
<td>The join is hash-based by partition (rowkey). In this case, there are equality constraints on both the partitioning and primary index columns. This enables a faster join since each non-eliminated partition needs to be joined with at most only one other partition. When this phrase is not reported, then the join is hash-based. That is, there are equality constraints on the primary index columns from which the hash is derived. For a partitioned table, there is additional overhead incurred by processing the table in hash order. Note that with either method, the join conditions must still be validated.</td>
</tr>
</tbody>
</table>
Example 7

The following example demonstrates one step for joining two tables having the same partitioning and primary keys. The relevant phrases are highlighted in boldface type.

```
CREATE TABLE Orders
(o_orderkey INTEGER NOT NULL,
o_custkey INTEGER,
o_orderstatus CHARACTER(1) CASESPECIFIC,
o_totalprice DECIMAL(13,2) NOT NULL,
o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
o_orderpriority CHARACTER(21),
o_clerk CHARACTER(16),
o_shippriority INTEGER,
o_comment VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(
o_orderdate BETWEEN DATE '1992-01-01' AND DATE '1998-12-31'
EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);
CREATE TABLE Lineitem
(l_orderkey INTEGER NOT NULL,
l_partkey INTEGER NOT NULL,
l_suppkey INTEGER,
l_linenumber INTEGER,
l_quantity INTEGER NOT NULL,
l_extendedprice DECIMAL(13,2) NOT NULL,
l_discount DECIMAL(13,2),
l_tax DECIMAL(13,2),
l_returnflag CHARACTER(1),
l_linestatus CHARACTER(1),
l_shipdate DATE FORMAT 'yyyy-mm-dd',
lcommitdate DATE FORMAT 'yyyy-mm-dd',
l_receiptdate DATE FORMAT 'yyyy-mm-dd',
l_shipinstruct VARCHAR(25),
l_shipmode VARCHAR(10),
l_comment VARCHAR(44))
PRIMARY INDEX (l_orderkey)
PARTITION BY RANGE_N(
l_shipdate BETWEEN DATE '1992-01-01'
AND DATE '1998-12-31'
EACH INTERVAL '1' MONTH);
```

```sql
EXPLAIN SELECT *
FROM lineitem, ordertbl
WHERE l_orderkey = o_orderkey
AND l_shipdate = o_orderdate
AND (o_orderdate < DATE '1993-10-01')
AND (o_orderdate >= DATE '1993-07-01')
ORDER BY o_orderdate, l_orderkey;
```
3) We do an all-AMPs JOIN step from **3 partitions of TH.ORDERTBL** with a condition of ("(TH.ORDERTBL.O_ORDERDATE < DATE '1993-10-01') AND (TH.ORDERTBL.O_ORDERDATE >= DATE '1993-07-01')")

TH.ORDERTBL and TH.LINEITEM are joined using **a rowkey-based inclusion merge join**, with a join condition of ("TH.LINEITEM.L_ORDERKEY = TH.ORDERTBL.O_ORDERKEY").

The input tables TH.ORDERTBL and TH.LINEITEM will not be cached in memory. The result goes into Spool 3 (all_amps), which is built locally on the AMPs. The size of Spool 3 is estimated with no confidence to be 7,739,047 rows. The estimated time for this step is 1 hour and 34 minutes.

---

**Example 8**

The following example demonstrates partition elimination in an aggregation. The relevant phrase is highlighted in boldface type.

```sql
CREATE TABLE t1
(a INTEGER,
 b INTEGER)
PRIMARY INDEX(a) PARTITION BY RANGE_N(
b BETWEEN 1
AND 10
EACH 1);

EXPLAIN SELECT MAX(a)
FROM t1
WHERE b > 3;
```

**Explanation**

1) First, we lock a distinct mws."pseudo table" for read on a RowHash to prevent global deadlock for mws.t1.
2) Next, we lock mws.t1 for read.
3) We do a SUM step to aggregate from **7 partitions of mws.t1** with a condition of ("mws.t1.b > 3"). 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. The size of Spool 3 is estimated with high confidence to be 1 row. The estimated time for this step is 2.35 seconds.
4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row. The estimated time for this step is 0.17 seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

-> The contents of Spool 1 are sent back to the user as the result of statement 1.
EXPLAIN Request Modifier and MERGE Conditional Steps

Introduction

The EXPLAIN request modifier is useful in determining the conditional steps in MERGE processing.7

Terminology

New terminology in this set of EXPLAIN reports is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>if no update in &lt;step number&gt;</td>
<td>The match condition for the update phase was not met, so the conditional insert phase will be performed. &lt;step number&gt; indicates the MERGE update step.</td>
</tr>
<tr>
<td>if &lt;step number&gt; not executed,</td>
<td>Indicates the action taken if the first step in an UPDATE block in not performed. Reported only if &lt;step number&gt; is greater than 1.</td>
</tr>
</tbody>
</table>

Database Object DDL for Examples

The following DDL statements create the tables accessed by the EXPLAINed DML statement examples:

```sql
CREATE TABLE contact (  
    contact_number INTEGER,  
    contact_name CHARACTER(30),  
    area_code SMALLINT NOT NULL,  
    phone INTEGER NOT NULL,  
    extension INTEGER )  
UNIQUE PRIMARY INDEX ( contact_number );

CREATE TABLE contact_t (  
    number INTEGER,  
    name CHARACTER(30),  
    area_code SMALLINT NOT NULL,  
    phone INTEGER NOT NULL,  
    extension INTEGER)  
UNIQUE PRIMARY INDEX (number);  
```

7. MERGE conditional steps are insert operations that are performed after an unconditional update operation does not meet its matching condition only when both WHEN MATCHED and WHEN NOT MATCHED clauses are specified.
Example 1

The following example demonstrates simple conditional insert processing without trigger or join index steps. The relevant phrases in the EXPLAIN report are highlighted in boldface type:

```
EXPLAIN MERGE INTO contact_t AS t
    USING (SELECT contact_number, contact_name, area_code, phone, 
            extension
    FROM contact 
    WHERE contact_number = 8005) s
ON (t.number =  8005)
WHEN MATCHED THEN
    UPDATE SET name = 'Name beingUpdated', 
            extension = s.extension 
WHEN NOT MATCHED THEN
    INSERT (number, name, area_code, phone, extension) 
VALUES (s.contact_number, s.contact_name, 
        s.area_code, s.phone, s.extension) ;
```

*** Help information returned. 20 rows.  
*** Total elapsed time was 1 second.

Explanation
1) First, we do a single-AMP MERGE DELETE to TEST.contact_t from TEST.contact by way of a RowHash match scan. New updated rows are built and the result goes into Spool 1 (one-amp), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by row hash.
2) Next, we execute the following steps in parallel.
   1) We do a single-AMP MERGE into TEST.contact_t from Spool 1 (Last Use).
   2) If no update in 2.1, we do a single-AMP RETRIEVE step from TEST.contact by way of the unique primary index "TEST.contact.contact_number = 8005" with no residual conditions into Spool-2 (one-amp), which is built locally on that AMP. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated with high confidence to be 1 row. The estimated time for this step is 0.02 seconds.
   3) If no update in 2.1, we do a single-AMP MERGE into TEST.contact_t from Spool 2 (Last Use).
   4) 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.

Only if no update in <step number> conditional steps are reported by this EXPLAIN, indicating the steps to be performed only if the update to contact_t fails. The report indicates that the MERGE first attempts to perform an update operation (step 2.1). If no row is found to update, then the statement inserts a new row (step 3).

Example 2

The following example demonstrates slightly more complicated conditional insert processing when a join index is defined on tables t1 and t2.

The DDL for the join index is as follows:

```
CREATE JOIN INDEX j AS
    SELECT * FROM t1 LEFT OUTER JOIN t2 
    ON (t1.y1 = t2.y2);
```
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN Request Modifier and MERGE Conditional Steps

SQL Request and Transaction Processing 531

The relevant phrases in the EXPLAIN report are highlighted in boldface type:

```
EXPLAIN MERGE INTO t1
      USING VALUES(1,2) AS s(x1, y1)
    ON t1.x1 = 4
    WHEN MATCHED THEN
      UPDATE SET y1 = 5
    WHEN NOT MATCHED THEN
      INSERT(4,5);
```

*** Help information returned. 44 rows.
*** Total elapsed time was 1 second.

Explanation
---------------------------------------------------------------------------
1) First, we lock a distinct TEST."pseudo table" for read on a RowHash to prevent global deadlock for TEST.t2.
2) Next, we lock TEST.t2 for read.
3) We execute the following steps in parallel.
   1) We do a single-AMP DELETE from TEST.j by way of the primary index "TEST.j.x1 = 4" with no residual conditions.
   2) We do an all-AMPs RETRIEVE step from TEST.t2 by way of an all-rows scan with a condition of ("TEST.t2.y2 = 5") into Spool 2 (one-amp), which is redistributed by hash code to all AMPs. The size of Spool 2 is estimated with no confidence to be 1 row. The estimated time for this step is 0.02 seconds.
4) We do a single-AMP JOIN step from TEST.t1 by way of the primary index "TEST.t1.x1 = 4" with no residual conditions, which is joined to Spool 2 (Last Use). TEST.t1 and Spool 2 are left outer joined using a product join, with a join condition of ("1=1").
   The result goes into Spool 1 (one-amp), which is redistributed by hash code to all AMPs. Then we do a SORT to order Spool 1 by row hash. The size of Spool 1 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.03 seconds.
5) We execute the following steps in parallel.
   1) We do a single-AMP MERGE into TEST.j from Spool 1 (Last Use).
   2) We do a single-AMP UPDATE from TEST.t1 by way of the primary index "TEST.t1.x1 = 4" with no residual conditions.
   3) **If no update in 5.2**, we do an INSERT into TEST.t1.
   4) **If no update in 5.2**, we do an INSERT into Spool 3.
   5) **If no update in 5.2**, we do an all-AMPs RETRIEVE step from TEST.t2 by way of an all-rows scan with no residual conditions into Spool 5 (all amps), which is duplicated on all AMPs. The size of Spool 5 is estimated with low confidence to be 4 rows. The estimated time for this step is 0.02 seconds.
6) **If no update in 5.2**, we do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an all-rows scan, which is joined to Spool 5 (Last Use). Spool 3 and Spool 5 are left outer joined using a product join, with a join condition of ("y1 = y2"). The result goes into Spool 4 (one-amp), which is redistributed by hash code to all AMPs. Then we do a SORT to order Spool 4 by row hash. The size of Spool 4 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.03 seconds.
7) **If no update in 5.2**, we do a single-AMP MERGE into TEST.j from Spool 4 (Last Use).
8) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
-> No rows are returned to the user as the result of statement 1.

Again, only if no update in <step number> conditional steps are reported by this EXPLAIN. These steps are performed only if the initial unconditional update attempt to table t1 is unsuccessful. The report indicates that the MERGE first attempts to perform an update operation (step 5.2). If no row is found to update, then the statement inserts a new row (steps 5.3 and 5.4). The join index j is updated by steps 3.1, 5.1, and 7.
Example 3

The following example demonstrates conditional insert processing when an upsert trigger is defined on table \( t1 \).

The DDL for the trigger is as follows:

```sql
CREATE TRIGGER r1 AFTER INSERT ON t1
    (UPDATE t2 SET y2 = 9 WHERE x2 = 8
    ELSE INSERT t2(8,9);
);
```

The relevant phrases in the EXPLAIN report are highlighted in boldface type:

```sql
EXPLAIN MERGE INTO t1
    USING VALUES(1,2) AS s(x1, y1)
    ON t1.x1 = 4
    WHEN MATCHED THEN
        UPDATE SET y1 = 5
    WHEN NOT MATCHED THEN
        INSERT(4,5);
```

*** 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 TEST.t1 by way of the primary index "TEST.t1.x1 = 4" with no residual conditions.
   2) If no update in 1.1, we do an INSERT into TEST.t1.
   3) If no update in 1.1, we do a single-AMP UPDATE from TEST.t2 by way of the primary index "TEST.t2.x2 = 8" with no residual conditions. If the row cannot be found, then we do an INSERT into TEST.t2.

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.

The unconditional update to table \( t1 \) is attempted in step 1.1. If the update is unsuccessful, then the MERGE inserts a new row into \( t1 \) in step 1.2 and fires trigger \( r1 \), which attempts to update table \( t2 \) in step 1.3. If an update cannot be performed, then the trigger inserts a new row into table \( t2 \).

Example 4

The following example demonstrates conditional insert processing when an abort trigger is defined to fire after an update on table \( t4 \).

This example uses the conditional abort trigger defined by this DDL:

```sql
CREATE TRIGGER aborttrig AFTER UPDATE ON t4
    (UPDATE t5 SET y5 =5 WHERE x5 = 3
    ELSE INSERT t5(3,5);
    ABORT FROM t5 WHERE x5 =1;
    DELETE t3 WHERE x3 = 10;
    ABORT 'unconditional abort';
);
```
The relevant phrases in the EXPLAIN report are highlighted in boldface type:

```
EXPLAIN MERGE INTO t4
    USING VALUES(1,2) AS s(x1, y1)
ON t4.x4 = 4
WHEN MATCHED THEN
    UPDATE SET y4 = 5
WHEN NOT MATCHED THEN
    INSERT(4,5);
```

*** Help information returned. 18 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 TEST.t4 by way of the primary index "TEST.t4.x4 = 4" with no residual conditions.
   2) We do a single-AMP UPDATE from TEST.t5 by way of the primary index "TEST.t5.x5 = 3" with no residual conditions. If the row cannot be found, then we do an INSERT into TEST.t5.
2) Next, we execute the following steps in parallel.
   1) If no update in 1.1, we do a single-AMP ABORT test from TEST.t5 by way of the primary index "TEST.t5.x5 = 1" with no residual conditions.
   2) If no update in 1.1, we do a single-AMP DELETE from TEST.t3 by way of the primary index "TEST.t3.x3 = 10" with no residual conditions.
   3) If no update in 1.1, we unconditionally ABORT the transaction.
   4) If no update in 1.1, we do an INSERT into TEST.t4.
   5) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
-> No rows are returned to the user as the result of statement 1.

The unconditional update to table t4 is attempted in step 1.1. If the update is successful, then trigger aborttrig is fired, which attempts to perform an atomic upsert on table t5 in step 1.2.

If no update is made to table t4, then the MERGE inserts a new row into it in step 1.2 and fires trigger aborttrig, which attempts to perform an atomic upsert operation update on table t2 in step 1.3. If an update cannot be performed, then the trigger inserts a new row into table t2.
EXPLAIN and UPDATE (Upsert Form) Conditional Steps

Introduction

The EXPLAIN modifier is useful in determining the conditional steps in UPDATE (Upsert Form) processing.8

Terminology

New terminology in this set of EXPLAIN reports is defined as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>if no update in &lt;step number&gt;</td>
<td>The match condition for the update phase was not met, so the conditional insert phase will be performed. &lt;step number&gt; indicates the MERGE update step.</td>
</tr>
<tr>
<td>if &lt;step number&gt; not executed,</td>
<td>Indicates the action taken if the first step in an UPDATE block in not performed. Reported only if &lt;step number&gt; is greater than 1</td>
</tr>
</tbody>
</table>

Database Object DDL for Examples

The following DDL statements create the tables accessed by the EXPLAINed DML statement examples.

```sql
CREATE TABLE t1 (
    x1 INTEGER,
    y1 INTEGER );

CREATE TABLE t2 (
    x2 INTEGER NOT NULL,
    y2 INTEGER NOT NULL );
```

Example 1

The following example demonstrates simple conditional insert processing into table t1 without trigger or join index steps:

```sql
EXPLAIN UPDATE t1
    SET y1 = 3 WHERE x1 = 2
ELSE INSERT t1(2, 3);

*** Help information returned. 4 rows.
*** Total elapsed time was 1 second.
```

8. Atomic upsert conditional steps are insert operations that are performed after an unconditional update operation does not meet its matching condition.
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier

EXPLAIN and UPDATE (Upsert Form) Conditional Steps

Explanation
---------------------------------------------------------------------------
1) First, we do a single-AMP UPDATE from TEST.t1 by way of the primary index "TEST.t1.x1 = 2" with no residual conditions. If the row cannot be found, then we do an INSERT into TEST.t1. -> No rows are returned to the user as the result of statement 1.

Both the unconditional update attempt and the conditional insert are combined in a single step.

Example 2

The following example demonstrates slightly more complicated upsert processing when a join index is defined on tables t1 and t2.

The DDL for the join index is as follows:

```
CREATE JOIN INDEX j AS
  SELECT *
  FROM t1 LEFT OUTER JOIN t2 ON (t1.y1 = t2.y2);
```

```
EXPLAIN UPDATE t1
  SET y1 = 3 WHERE x1 = 2
  ELSE INSERT t1(2, 3);
```

*** Help information returned. 44 rows.
*** Total elapsed time was 1 second.

Explanation
---------------------------------------------------------------------------
1) First, we lock a distinct TEST."pseudo table" for read on a RowHash to prevent global deadlock for TEST.t2.
2) Next, we lock TEST.t2 for read.
3) We execute the following steps in parallel.
   1) We do a single-AMP DELETE from TEST.j by way of the primary index "TEST.j.x1 = 2" with no residual conditions.
   2) We do an all-AMPs RETRIEVE step from TEST.t2 by way of an all-rows scan with a condition of ("TEST.t2.y2 = 3") into Spool 2 (one-amp), which is redistributed by hash code to all AMPS. The size of Spool 2 is estimated with no confidence to be 1 row. The estimated time for this step is 0.02 seconds.
4) We do a single-AMP JOIN step from TEST.t1 by way of the primary index "TEST.t1.x1 = 2" with no residual conditions, which is joined to Spool 2 (Last Use). TEST.t1 and Spool 2 are left outer joined using a product join, with a join condition of ("(1=1)"). The result goes into Spool 1 (one-amp), which is redistributed by hash code to all AMPS. Then we do a SORT to order Spool 1 by row hash. The size of Spool 1 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.03 seconds.
5) We execute the following steps in parallel.
   1) We do a single-AMP MERGE into TEST.j from Spool 1 (Last Use).
   2) We do a single-AMP UPDATE from TEST.t1 by way of the primary index "TEST.t1.x1 = 2" with no residual conditions. If the row cannot be found, then we do an INSERT into TEST.t1.
   3) **If no update in 5.2**, we do an INSERT into Spool 3.
   4) **If no update in 5.2**, we do an all-AMPs RETRIEVE step from TEST.t2 by way of an all-rows scan with no residual conditions into Spool 5 (all_amps), which is duplicated on all AMPS. The size of Spool 5 is estimated with low confidence to be 4 rows. The estimated time for this step is 0.02 seconds.
6) **If no update in 5.2**, we do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an all-rows scan, which is joined to Spool 5 (Last Use). Spool 3 and Spool 5 are left outer joined using a product join, with a join condition of ("y1 = y2"). The result goes into Spool 4 (one-amp), which is redistributed by hash code to all AMPS. Then we do a SORT to order Spool 4 by row hash. The size of Spool 4 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.03 seconds.
7) **If no update in 5.2**, we do a single-AMP MERGE into TEST.j from Spool 4 (Last Use).
8) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
-> No rows are returned to the user as the result of statement 1.

The EXPLAIN report is more complicated because of the join index $j$. Notice the instances of the phrase "If no update in <step number>" in steps 5.3, 5.4, 6, and 7, indicating the operations undertaken if the match condition for update was not met. The report indicates that the MERGE first attempts to perform an update operation (step 5.2). If no row is found to update, then the statement inserts a new row (step 5.3). The join index $j$ is updated by steps 1 through 4, 6, and 7.

Example 3

The following upsert statement involves a simple trigger. The relevant phrases in the EXPLAIN report are highlighted in boldface type.

The DDL for the trigger is as follows:

```sql
CREATE TRIGGER r1 AFTER INSERT ON t1
    UPDATE t2 SET y2 = 9 WHERE x2 = 8
    ELSE INSERT t2(8,9);
);

EXPLAIN UPDATE t1
    SET y1 = 3 WHERE x1 = 2
    ELSE INSERT t1(2, 3);
```

*** 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 TEST.t1 by way of the primary index "TEST.t1.x1 = 2" with no residual conditions. If the row cannot be found, then we do an INSERT into TEST.t1.
   2) If no update in 1.1, we do a single-AMP UPDATE from TEST.t2 by way of the primary index "TEST.t2.x2 = 8" with no residual conditions. If the row cannot be found, then we do an INSERT into TEST.t2.
   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.

The EXPLAIN report is moderately complex because of the trigger. Step 1.1 handles the unconditional update attempt and the conditional insert, while step 1.2 handles the triggered update to table $t2$.

Notice the phrase "If no update in <step number>" in step 1.2, indicating that the step performs only if the match condition for update was not met.
Example 4

This example uses the conditional abort trigger defined by this DDL:

```sql
CREATE TRIGGER aborttrig AFTER UPDATE ON t4
  (UPDATE t5 SET y5 = 5 WHERE x5 = 3
  ELSE INSERT t5(3, 5);
  ABORT FROM t5 WHERE x5 = 1;
  DELETE t3 WHERE x3 = 10;
  ABORT 'unconditional abort';
);
```

The relevant phrases in the EXPLAIN report are highlighted in boldface type.

```sql
EXPLAIN UPDATE t4
  SET y4 = 3 WHERE x4 = 2
  ELSE INSERT t4(2, 3);
```

*** Help information returned. 18 rows.
*** Total elapsed time was 1 second.

Explanation

First, we execute the following steps in parallel.
1) We do a single-AMP UPDATE from TEST.t4 by way of the primary index "TEST.t4.x4 = 2" with no residual conditions.
2) We do a single-AMP UPDATE from TEST.t5 by way of the primary index "TEST.t5.x5 = 3" with no residual conditions. If the row cannot be found, then we do an INSERT into TEST.t5.

2) Next, we execute the following steps in parallel.
1) If no update in 1.1, we do a single-AMP ABORT test from TEST.t5 by way of the primary index "TEST.t5.x5 = 1" with no residual conditions.
2) If no update in 1.1, we do a single-AMP DELETE from TEST.t3 by way of the primary index "TEST.t3.x3 = 10" with no residual conditions.
3) If no update in 1.1, we unconditionally ABORT the transaction.
4) If no update in 1.1, we do an INSERT into TEST.t4.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

-> No rows are returned to the user as the result of statement 1.

Notice the instances of the phrase "If no update in <step number>" in steps 2.1, 2.2, 3, and 4, indicating that the step performs only if an update was not successful.

Step 1.1 handles the unconditional update attempt to t4, while step 1.2 performs the update processing defined by the trigger. Steps 2 and 3 continue to perform trigger-related operations, while step 4 performs the upsert-specified insert operation if the update to t4 fails.

Example 5

This example uses the 5 tables defined by the following DDL statements:

```sql
CREATE TABLE t7 (x7 INTEGER, y7 INTEGER);
CREATE TABLE t8 (x8 INTEGER, y8 INTEGER);
CREATE TABLE t9 (x9 INTEGER, y9 INTEGER);
```
CREATE TABLE t10 (  
x10 INTEGER,
y10 INTEGER);
CREATE TABLE t11 (  
x11 INTEGER,
y11 INTEGER);
The example also uses the following definitions for triggers r6 through r10:
CREATE TRIGGER r6 ENABLED AFTER UPDATE ON t1 
(UPDATE t7 SET y7 = 7 WHERE x7 = 6 
ELSE INSERT t7(6, 7); ) ;
CREATE TRIGGER r7 ENABLED AFTER UPDATE ON t7 
(UPDATE t8 SET y8 = 8 WHERE x8 = 7 
ELSE INSERT t8(7, 8); ) ;
CREATE TRIGGER r8 ENABLED AFTER UPDATE ON t7 
(UPDATE t9 SET y9 = 8 WHERE x9 = 7 
ELSE INSERT t9(7, 8); ) ;
CREATE TRIGGER r9 ENABLED AFTER INSERT ON t7 
(UPDATE t10 SET y10 = 9 WHERE x10 = 8 
ELSE INSERT t10(8, 9); ) ;
CREATE TRIGGER r10 ENABLED AFTER INSERT ON t7 
(UPDATE t11 SET y11 = 10 WHERE x11 = 9 
ELSE INSERT t11(9, 10); ) ;
EXPLAIN UPDATE t1 
SET y1 = 20 WHERE x1 = 30 
ELSE INSERT t1(30, 20);
*** Help information returned. 41 rows.
*** Total elapsed time was 1 second.
The relevant phrases in the EXPLAIN report are highlighted in boldface type.

Explanation
-------------------------------------------------------
1) First, we do a single-AMP UPDATE from Test.t1 by way of the primary index Test.t1.x1 = 30 with no residual conditions.
2) Next, we execute the following steps in parallel.
   1) We do a single-AMP UPDATE from Test.t2 by way of the primary index Test.t2.x2 = 1 with no residual conditions.
   2) If no update in 2.1, we do a single-AMP UPDATE from Test.t3 by way of the primary index Test.t3.x3 = 2 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t3.
   3) If no update in 2.1, we do a single-AMP UPDATE from Test.t4 by way of the primary index Test.t4.x4 = 3 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t4.
   4) If no update in 2.1, we do an INSERT into Test.t2.
   5) If no update in 2.1, we do a single-AMP UPDATE from Test.t5 by way of the primary index Test.t5.x5 = 4 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t5.
   6) If no update in 2.1, we do a single-AMP UPDATE from Test.t6 by way of the primary index Test.t6.x6 = 5 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t6.
   7) If no update in 2.1, we do an INSERT into Test.t1.
8) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
-> No rows are returned to the user as the result of statement 1.
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier
EXPLAIN and UPDATE (Upsert Form) Conditional Steps

This EXPLAIN report is more complicated because of the triggers r6, r7, r8, r9, and r10. Notice the instances of the phrase “If no update in <step number>” in steps 2.2, 3, 4, 5, 6 and 7, indicating steps that are taken only if an unconditional update operation fails.

Only step 1 and step 7 relate directly to the atomic upsert statement. Steps 2 through 6 pertain to the triggers r6, r7, r8, r9, and r10.

Example 6

This example disables the triggers r6 through r10 from the previous example and invokes the following newly defined triggers:

The DDL statements are as follows:

```
ALTER TRIGGER r6 DISABLED;
ALTER TRIGGER r7 DISABLED;
ALTER TRIGGER r8 DISABLED;
ALTER TRIGGER r9 DISABLED;
ALTER TRIGGER r10 DISABLED;

CREATE TRIGGER r11 ENABLED AFTER UPDATE ON t1
    (UPDATE t7 SET y7 = 7 WHERE x7 = 6
    ELSE INSERT t7(6, 7));

CREATE TRIGGER r12 ENABLED AFTER UPDATE ON t7
    (UPDATE t8 SET y8 = 8 WHERE x8 = 7
    ELSE INSERT t8(7, 8));

CREATE TRIGGER r13 ENABLED AFTER UPDATE ON t7
    (UPDATE t9 SET y9 = 8 WHERE x9 = 7
    ELSE INSERT t9(7, 8));

CREATE TRIGGER r14 ENABLED AFTER INSERT ON t7
    (UPDATE t10 SET y10 = 9 WHERE x10 = 8
    ELSE INSERT t10(8, 9));

CREATE TRIGGER r15 ENABLED AFTER INSERT ON t7
    (UPDATE t11 SET y11 = 10 WHERE x11 = 9
    ELSE INSERT t11(9, 10));

EXPLAIN UPDATE t1
    SET y1 = 20 WHERE x1 = 30
    ELSE INSERT t1(30, 20);
```

*** Help information returned. 23 rows.
*** Total elapsed time was 1 second.

The relevant phrases in the EXPLAIN report are highlighted in boldface type.

Explanation

1) First, we do a single-AMP UPDATE from Test.t1 by way of the primary index Test.t1.x1 = 30 with no residual conditions.
2) Next, we execute the following steps in parallel.
   1) We do a single-AMP UPDATE from Test.t7 by way of the primary index Test.t7.x7 = 6 with no residual conditions.
   2) If no update in 2.1, we do a single-AMP UPDATE from Test.t8 by way of the primary index Test.t8.x8 = 7 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t8.
3) If no update in 2.1, we do a single-AMP UPDATE from Test.t9 by way of the primary index Test.t9.x9 = 7 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t9.
4) If no update in 2.1, we do an INSERT into Test.t7.
5) If no update in 2.1, we do a single-AMP UPDATE from Test.t10 by way of the primary index Test.t10.x10 = 8 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t10.
6) If no update in 2.1, we do a single-AMP UPDATE from Test.t11 by way of the primary index Test.t11.x11 = 9 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t11.
7) If no update in 2.1, we do an INSERT into Test.t1.
8) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

-> No rows are returned to the user as the result of statement 1.

Notice the instances of the phrase “If no update in <step number>” in steps 2.2, 3, 4, 5, 6, and 7, indicating operations that are performed only if an update does not succeed.

Only steps 1 and 7 relate directly to the atomic upsert statement. The other steps all perform triggered actions specified by triggers r11 through r15.

**Example 7**

This example disables the triggers r6 through r10 from the previous example and invokes the following newly defined triggers:

The DDL statements to disable these triggers are as follows:

```
ALTER TRIGGER r11 DISABLED;
ALTER TRIGGER r12 DISABLED;
ALTER TRIGGER r13 DISABLED;
ALTER TRIGGER r14 DISABLED;
ALTER TRIGGER r15 DISABLED;
```

```
CREATE TRIGGER r16 ENABLED AFTER INSERT ON t1
(UPDATE t12 SET y12 = 11 WHERE x12 = 10
 ELSE INSERT t12(10, 11));
```

```
CREATE TRIGGER r17 ENABLED AFTER UPDATE ON t12
(UPDATE t13 SET y13 = 12 WHERE x13 = 11
 ELSE INSERT t13(11, 12));
```

```
CREATE TRIGGER r18 ENABLED AFTER UPDATE ON t12
(UPDATE t14 SET y14 = 13 WHERE x14 = 12
 ELSE INSERT t14(12, 13));
```

```
CREATE TRIGGER r19 ENABLED AFTER INSERT ON t12
(UPDATE t15 SET y15 = 14 WHERE x15 = 13
 ELSE INSERT t15(13, 14));
```

```
CREATE TRIGGER r20 ENABLED AFTER INSERT ON t12
(UPDATE t16 SET y16 = 14 WHERE x16 = 13
 ELSE INSERT t16(13, 14));
```

Now, the EXPLAIN statement:

```
EXPLAIN UPDATE t1
    SET y1 = 20 WHERE x1 = 30
ELSE INSERT t1(30, 20);
```

*** Help information returned. 25 rows.
*** Total elapsed time was 1 second.
The relevant phrases in the EXPLAIN report are highlighted in boldface type.

**Explanation**

1) First, we execute the following steps in parallel.

1) We do a single-AMP UPDATE from Test.t1 by way of the primary index Test.t1.x1 = 30 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t1.

2) **If no update in 1.1**, we do a single-AMP UPDATE from Test.t12 by way of the primary index Test.t12.x12 = 10 with no residual conditions.

3) **If no update in 1.2**, we do a single-AMP UPDATE from Test.t13 by way of the primary index Test.t13.x13 = 11 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t13.

2) Next, **if no update in 1.2**, we do a single-AMP UPDATE from Test.t14 by way of the primary index Test.t14.x14 = 12 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t14.

3) **If no update in 1.2**, we do an INSERT into Test.t12.

4) **If no update in 1.2**, we do a single-AMP UPDATE from Test.t15 by way of the primary index Test.t15.x15 = 13 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t15.

5) **If no update in 1.2**, we do a single-AMP UPDATE from Test.t16 by way of the primary index Test.t16.x16 = 13 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t16.

6) 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.

Notice the instances of the phrase “If no update in <step number>” in steps 1.2, 1.3, 2, 3, 4, and 5 indicating operations that are performed only if an update does not succeed.

Only step 1.1 relates directly to the atomic upsert statement. The other steps all perform triggered actions specified by triggers r16 through r20.

**Example 8**

This example disables the triggers r16 through r20 from the previous example and invokes the following newly defined triggers. The query plan for this example is identical to that of “Example 7” on page 540, as the EXPLAIN report at the end confirms.

```
CREATE TRIGGER r21 ENABLED AFTER INSERT ON t1
  (UPDATE t17 SET y17 = 11 WHERE x17 = 10
   ELSE INSERT t17(10, 11));

CREATE TRIGGER r22 ENABLED AFTER UPDATE ON t17
  (UPDATE t18 SET y18 = 12 WHERE x18 = 11
   ELSE INSERT t18(11, 12));

CREATE TRIGGER r23 ENABLED AFTER UPDATE ON t17
  (UPDATE t19 SET y19 = 13 WHERE x19 = 12
   ELSE INSERT t19(12, 13));

CREATE TRIGGER r24 ENABLED AFTER INSERT ON t17
  (UPDATE t20 SET y20 = 14 WHERE x20 = 13
   ELSE INSERT t20(13, 14));

CREATE TRIGGER r25 ENABLED AFTER INSERT ON t17
  (UPDATE t21 SET y21 = 14 WHERE x21 = 13
   ELSE INSERT t21(13, 14));
```
Chapter 5: Interpreting the Output of the EXPLAIN Request Modifier
EXPLAIN and UPDATE (Upsert Form) Conditional Steps

```
EXPLAIN UPDATE t1
  SET y1 = 20 WHERE x1 = 30
  ELSE INSERT t1(30, 20);

*** Help information returned. 25 rows.
*** Total elapsed time was 1 second.
```

The relevant phrases in the EXPLAIN report are highlighted in boldface type.

**Explanation**

1) We do a single-AMP UPDATE from Test.t1 by way of the primary index Test.t1.x1 = 30 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t1.

2) **If no update in 1.1**, we do a single-AMP UPDATE from Test.t17 by way of the primary index Test.t17.x17 = 10 with no residual conditions.

3) **If no update in 1.2**, we do a single-AMP UPDATE from Test.t18 by way of the primary index Test.t18.x18 = 11 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t18.

4) **Next, if no update in 1.2**, we do a single-AMP UPDATE from Test.t19 by way of the primary index Test.t19.x19 = 12 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t19.

5) **If no update in 1.2**, we do an INSERT into Test.t17.

6) **If no update in 1.2**, we do a single-AMP UPDATE from Test.t20 by way of the primary index Test.t20.x20 = 13 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t20.

7) **If no update in 1.2**, we do a single-AMP UPDATE from Test.t21 by way of the primary index Test.t21.x21 = 13 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t21.

8) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

```
-> No rows are returned to the user as the result of statement 1.
```

Notice the instances of the phrase “If no update in <step number>” in steps 1.2, 1.3, 2, 3, 4, and 5 indicating operations that are performed only if an update does not succeed.

Only step 1.1 relates directly to the atomic upsert statement. The other steps all perform triggered actions specified by triggers r21 through r25.

**Example 9**

This example performs with triggers r1 through r15 enabled, a set of conditions that adds many steps. Note that triggers r21 through r25 are already enabled in the previous example.

```
ALTER TRIGGER r1 ENABLED;
ALTER TRIGGER r2 ENABLED;
ALTER TRIGGER r3 ENABLED;
ALTER TRIGGER r4 ENABLED;
ALTER TRIGGER r5 ENABLED;
ALTER TRIGGER r6 ENABLED;
ALTER TRIGGER r7 ENABLED;
ALTER TRIGGER r8 ENABLED;
ALTER TRIGGER r9 ENABLED;
ALTER TRIGGER r10 ENABLED;
ALTER TRIGGER r11 ENABLED;
ALTER TRIGGER r12 ENABLED;
ALTER TRIGGER r13 ENABLED;
ALTER TRIGGER r14 ENABLED;
ALTER TRIGGER r15 ENABLED;
```
EXPLAIN UPDATE t1
    SET y1 = 20 WHERE x1 = 30
    ELSE INSERT t1(30, 20);

*** Help information returned. 82 rows.
*** Total elapsed time was 1 second.

The relevant phrases in the EXPLAIN report are highlighted in boldface type.

Explanation
----------------------------------------------------------------------
1) First, we do a single-AMP UPDATE from Test.t1 by way of the primary index Test.t1.x1 = 30 with no residual conditions.
2) Next, we execute the following steps in parallel.
   1) We do a single-AMP UPDATE from Test.t2 by way of the primary index Test.t2.x2 = 1 with no residual conditions.
   2) If no update in 2.1, we do a single-AMP UPDATE from Test.t3 by way of the primary index Test.t3.x3 = 2 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t3.
3) If no update in 2.1, we do a single-AMP UPDATE from Test.t4 by way of the primary index Test.t4.x4 = 3 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t4.
4) If no update in 2.1, we do an INSERT into Test.t2.
5) If no update in 2.1, we do a single-AMP UPDATE from Test.t5 by way of the primary index Test.t5.x5 = 4 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t5.
6) If no update in 2.1, we do a single-AMP UPDATE from Test.t6 by way of the primary index Test.t6.x6 = 5 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t6.
7) We execute the following steps in parallel.
   1) We do a single-AMP UPDATE from Test.t7 by way of the primary index Test.t7.x7 = 6 with no residual conditions.
   2) If no update in 7.1, we do a single-AMP UPDATE from Test.t8 by way of the primary index Test.t8.x8 = 7 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t8.
8) If no update in 7.1, we do a single-AMP UPDATE from Test.t9 by way of the primary index Test.t9.x9 = 8 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t9.
9) If no update in 7.1, we do an INSERT into Test.t7.
10) If no update in 7.1, we do a single-AMP UPDATE from Test.t10 by way of the primary index Test.t10.x10 = 8 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t10.
11) If no update in 7.1, we do a single-AMP UPDATE from Test.t11 by way of the primary index Test.t11.x11 = 9 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t11.
12) If no update in 7.1, we do an INSERT into Test.t1.
13) We execute the following steps in parallel.
   1) If no update in 1, we do a single-AMP UPDATE from Test.t12 by way of the primary index Test.t12.x12 = 10 with no residual conditions.
   2) If no update in 13.1, we do a single-AMP UPDATE from Test.t13 by way of the primary index Test.t13.x13 = 11 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t13.
14) If no update in 13.1, we do a single-AMP UPDATE from Test.t14 by way of the primary index Test.t14.x14 = 12 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t14.
15) If no update in 13.1, we do an INSERT into Test.t12.
16) If no update in 13.1, we do a single-AMP UPDATE from Test.t15 by way of the primary index Test.t15.x15 = 13 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t15.
17) If no update in 13.1, we do a single-AMP UPDATE from Test.t16 by way of the primary index Test.t16.x16 = 13 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t16.
18) We execute the following steps in parallel.
   1) If no update in 1, we do a single-AMP UPDATE from Test.t17 by way of the primary index Test.t17.x17 = 10 with no residual conditions.
   2) If no update in 18.1, we do a single-AMP UPDATE from Test.t18 by way of the primary index Test.t18.x18 = 11 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t18.
19) If no update in 18.1, we do a single-AMP UPDATE from Test.t19 by way of the primary index Test.t19.x19 = 12 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t19.
20) If no update in 18.1, we do an INSERT into Test.t17.
21) **If no update in 18.1,** we do a single-AMP UPDATE from Test.t20 by way of the primary index Test.t20.x20 = 13 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t20.

22) **If no update in 18.1,** we do a single-AMP UPDATE from Test.t21 by way of the primary index Test.t21.x21 = 13 with no residual conditions. If the row cannot be found, then we do an INSERT into Test.t21.

23) 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.

Notice the instances of the phrase “If no update in <step number>” in steps 2.2, 3, 4, 5, 6, 7.2, 8, 9, 10, 11, 12, 13.1, 13.2, 14, 15, 16, 17, 18.1, 18.2, 19, 20, 21, and 22, indicating operations that are performed only if an update does not succeed.

Only steps 1 and 12 relate directly to the atomic upsert statement. The other steps all perform triggered actions specified by the triggers.

Also note that steps 13.1 and 18.1 are based on the action in step 1, because the triggers r16 and r21 are defined with an insert triggering statement on the subject table t1.
CHAPTER 6 Query Capture Facility

This chapter describes the Query Capture Database (QCD) that supports the various Database Query Analysis tools such as the Query Capture Facility, the Visual EXPLAIN utility, and the Teradata Database 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.

See the following manuals and chapters for additional information about the various query analysis tools and SQL statements that are relevant to this chapter:

- Teradata Index Wizard User Guide
- Teradata Statistics Wizard User Guide
- Teradata Visual Explain User Guide
- SQL Data Manipulation Language
  - COLLECT STATISTICS (QCD Form)
  - DROP STATISTICS (QCD Form)
  - DUMP EXPLAIN
  - INITIATE INDEX ANALYSIS
  - INITIATE PARTITION ANALYSIS
  - INSERT EXPLAIN
- Chapter 7: “Database Foundations for the Teradata Index and Statistics Wizards”
Compatibility Issues With Prior Teradata Database 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 Teradata Database V2R5.0 are not usable with the Teradata Database 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 does not provide a default QCD. Because of this, DUMP EXPLAIN and INSERT EXPLAIN statements require you to specify a QCD name.
  
  QCD applications that worked prior to Teradata Database V2R5.0 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 create your QCD databases using the procedures described in “Creating the QCD Tables” on page 551.

QCD Information Source

The principal source of the captured information in QCD is the white tree produced by the 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 performed by the Optimizer.

Statistical and other demographic information in the QCD is captured using the following set of SQL statements:

- COLLECT DEMOGRAPHICS
- COLLECT STATISTICS (QCD Form)
- INSERT EXPLAIN … WITH STATISTICS

See SQL Data Manipulation Language for further information.

Applications of QCF and QCD

Teradata Database supports the following applications of QCF and QCD:

- QCD provides the foundation for the Teradata Database 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.
Chapter 6: Query Capture Facility

Functional Overview of the Query Capture Facility

- 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 Database Design for information about physical capacity planning for your QCDs.

DBS Control Flags for Tuning the Workload Cache Limits for Index and Partition Analysis

Index Wizard processing in Teradata Database allocates workload cache for the purpose of caching information retrieved from the QCD and other temporary working structures and needed during Analysis and Validation phases of recommending indexes and partitioning expressions for various SQL workloads.

Two DBS Control performance group flags control the respective workload cache sizes for the Analysis and Validation phases:

- IAMaxWorkloadCache
- IVMaxWorkloadCache

The following table provides some of the details for these flags. See Utilities for further information.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Purpose</th>
<th>Valid Range (megabytes)</th>
<th>Default (megabytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAMaxWorkloadCache</td>
<td>Defines the maximum size of the Index Wizard workload cache for Analysis operations.</td>
<td>32 - 256</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>This parameter is applicable to both the INITIATE INDEX ANALYSIS and INITIATE PARTITION ANALYSIS statements.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVMaxWorkloadCache</td>
<td>Defines the maximum size of the Index Wizard workload cache for Validation operations.</td>
<td>1 - 32</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>This parameter is applicable to all SQL statements issued within a session when DIAGNOSTIC &quot;Validate Index&quot; has been enabled.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Query Capture Database

Introduction

The Query Capture Database, or QCD, is the minimum set of tables required to store the 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 also documented in this chapter.

Use the Control Center feature of the Visual EXPLAIN tool (see “Procedure Using the Visual EXPLAIN Utility” on page 551) or BTEQ using the CreateQCF table (see “Procedure Using BTEQ” on page 552) 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.

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 exists to feed the values of the artificial sequential number columns defined for several attributes in QCD.
Chapter 6: Query Capture Facility
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 553.

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 552.

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.
12 Populate the QCD tables using the appropriate tools:
   • INSERT EXPLAIN statements to capture query plans. See SQL Data Manipulation Language 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.
13 End of procedure.
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`.

1. Start BTEQ.
2. Change your current database to the database in which the QCF tables are to be created as follows.
   ```sql
   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:
   a. `.SET WIDTH 254`
   b. `.EXPORT FILE = file_name`
   c. `SELECT TabDefinition
      FROM systemfe.CreateQCF
      ORDER BY SeqNumber;`
   d. `.EXPORT FILE = file_name`
   e. `.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 b.
   f. End of sub-procedure.
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.
5. Populate the QCD tables using the appropriate tools:
   - INSERT EXPLAIN statements to capture query plans. See SQL Data Manipulation Language 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 553.

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.
8. End of procedure.

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.

1. Start BTEQ.
2. Change your current database to the database from which the QCF tables are to be dropped as follows.

   ```sql
   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.
   a .EXPORT FILE = file_name
   b SELECT DelTable
       FROM systemfe.CleanupQCF
       ORDER BY SeqNumber;
   c EXPORT RESET;
   d .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.
   e End of sub-procedure.
4 End of procedure.
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 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 556 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 SQL Data Manipulation Language for the syntax of the DIAGNOSTIC "Validate Index" statement.
QCD Query Macros and Views

Introduction

Teradata Database 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 Database 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:

<table>
<thead>
<tr>
<th>Version</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Access restricted to information inserted by the current user in the QCD.</td>
</tr>
<tr>
<td>NonX</td>
<td>Access permitted to information inserted by any user of the QCD.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>User Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Loads, views, and deletes own plans or workloads only.</td>
</tr>
<tr>
<td>Power</td>
<td>Loads and views plans or workloads inserted by any user.</td>
</tr>
<tr>
<td></td>
<td>Deletes own plans or workloads only.</td>
</tr>
<tr>
<td>Administrator</td>
<td>Loads, view, and deletes any plan created by any user.</td>
</tr>
<tr>
<td></td>
<td>Deletes QCD tables.</td>
</tr>
<tr>
<td></td>
<td>QCD creator has Administrator privileges granted by default.</td>
</tr>
</tbody>
</table>
### Specific User Category Privileges

The following table indicates the specific privileges granted to each user category:

<table>
<thead>
<tr>
<th>User Category</th>
<th>Database Object Type</th>
<th>Privileges Granted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>QCD tables</td>
<td>• INSERT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• UPDATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SELECT on the following tables:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• IndexRecommendations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SeqNumber</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TableStatistics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DELETE on the following table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AnalysisLog</td>
</tr>
<tr>
<td></td>
<td>X views</td>
<td>SELECT</td>
</tr>
<tr>
<td></td>
<td>X macros</td>
<td>EXEC</td>
</tr>
<tr>
<td></td>
<td>Non-X views</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>Non-X macros</td>
<td>none</td>
</tr>
<tr>
<td>Power</td>
<td>QCD tables</td>
<td>• INSERT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• UPDATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SELECT on the following tables:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• IndexRecommendations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SeqNumber</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TableStatistics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DELETE on the following table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AnalysisLog</td>
</tr>
<tr>
<td></td>
<td>X views</td>
<td>SELECT</td>
</tr>
<tr>
<td></td>
<td>X macros</td>
<td>EXEC</td>
</tr>
<tr>
<td></td>
<td>Non-X views</td>
<td>SELECT</td>
</tr>
<tr>
<td></td>
<td>Non-X macros</td>
<td>EXEC (excluding DELETE plan and workload macros)</td>
</tr>
<tr>
<td>User Category</td>
<td>Database Object Type</td>
<td>Privileges Granted</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Administrator</td>
<td>QCD tables</td>
<td>• DELETE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DROP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• INSERT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SELECT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• UPDATE</td>
</tr>
<tr>
<td>X views⁺</td>
<td></td>
<td>• DELETE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DROP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• INSERT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SELECT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• UPDATE</td>
</tr>
<tr>
<td>X macros</td>
<td></td>
<td>• DROP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• EXECUTE</td>
</tr>
<tr>
<td>Non-X views</td>
<td></td>
<td>• DELETE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DROP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• INSERT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SELECT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• UPDATE</td>
</tr>
<tr>
<td>Non-X macros</td>
<td></td>
<td>• DROP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• EXECUTE</td>
</tr>
</tbody>
</table>

a. X views use the value for UDB.Key in the Query table (see “Query” on page 594) to restrict access to query plans to the user who captures them, while the Non-X views do not restrict that access.
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 560
- “AnalysisStmts” on page 562
- “DataDemographics” on page 564
- “Field” on page 566
- “Index_Field” on page 569
- “IndexColumns” on page 570
- “IndexMaintenance” on page 571
- “IndexRecommendations” on page 573
- “IndexTable” on page 578
- “JoinIndexColumns” on page 582
- “Predicate” on page 589
- “Predicate_Field” on page 591
- “QryRelX” on page 592
- “Query” on page 594
- “QuerySteps” on page 598
- “Relation” on page 609
- “SeqNumber” on page 615
- “StatsRecs” on page 617
- “TableStatistics” on page 619
- “User_Database” on page 622
- “UserRemarks” on page 623
- “ViewTable” on page 625
- “Workload” on page 627
- “WorkloadQueries” on page 628
- “WorkloadStatus” on page 629
- “XMLQCD” on page 631
### AnalysisLog

#### Function

Records the log information for index analysis by the Teradata Index Wizard utility.

*AnalysisLog* maintains the checkpoint information that is recorded when you enable the CHECKPOINT option for an INITIATE INDEX ANALYSIS request.

#### Table Definition

```sql
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkloadID</td>
<td>• Uniquely identifies the workload.</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table.</td>
</tr>
<tr>
<td>RecommendationID</td>
<td>Uniquely identifies the index recommendation generated for the specific index analysis.</td>
</tr>
<tr>
<td>IndexNameTag</td>
<td>• User-specified name for the index analysis.</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table.</td>
</tr>
<tr>
<td>SeqNumber</td>
<td>A sequence number to identify individual rows that belong to a multirow <em>AnalysisStatus</em>.</td>
</tr>
<tr>
<td></td>
<td>The sequence begins at 1.</td>
</tr>
</tbody>
</table>
### AnalysisLog

#### Attribute | Definition
--- | ---
Cflag | Used with multirow AnalysisStatus.

<table>
<thead>
<tr>
<th>IF the row is ...</th>
<th>THEN Cflag is set to ...</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

UserName | Name of the user performing the index analysis.

AnalysisStatus | Identifies the status of the index analysis.
This column contains encoded information that the index wizard infrastructure uses. During a checkpoint operation, this information is updated.

StartTime | Timestamp for the beginning of the index analysis.

UpdateTime | Timestamp for the moment the AnalysisStatus column is most recently updated.
AnalysisStmts

Function

Supports combined use of the CHECKPOINT and TIMELIMIT options for the INITIATE INDEX ANALYSIS and RESTART INDEX ANALYSIS statements, particularly in the context of submitting an INITIATE INDEX ANALYSIS request that is halted for some reason, followed by one or more RESTART INDEX ANALYSIS requests.

The table provides the following information to support generating index recommendations in this scenario:

- Identifies a group of logically related INITIATE INDEX ANALYSIS and RESTART INDEX ANALYSIS requests performed as a related sequence.
- Provides the sequence number for a given RESTART INDEX ANALYSIS request within the group.
- Provides the SQL text for the INITIATE INDEX ANALYSIS or RESTART INDEX ANALYSIS request used to produce a given set of index recommendations.

Table Definition

The following CREATE TABLE request defines the AnalysisStmts table:

```sql
CREATE SET TABLE QCD.analysisstmts, NO FALLBACK, NO BEFORE JOURNAL, NO AFTER JOURNAL, CHECKSUM=DEFAULT (
    WorkLoadID INTEGER NOT NULL,
    IndexNameTag VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
    RecommendationID INTEGER NOT NULL,
    IASeqNumber INTEGER NOT NULL,
    IAStmtText VARCHAR(1024) CHARACTER SET UNICODE NOT CASESPECIFIC)
PRIMARY INDEX (WorkLoadID);
```
Attribute Definitions

The following table defines the *AnalysisStmts* table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkLoadID</td>
<td>• Uniquely identifies the workload.</td>
</tr>
<tr>
<td></td>
<td>• NUPI for the table.</td>
</tr>
<tr>
<td>IndexNameTag</td>
<td>User-specified name for the index analysis.</td>
</tr>
<tr>
<td>RecommendationID</td>
<td>Uniquely identifies a set of index recommendations.</td>
</tr>
<tr>
<td>IASeqNumber</td>
<td>System-assigned sequence number for this index analysis.</td>
</tr>
<tr>
<td>IASmtText</td>
<td>The SQL text for the request being analyzed.</td>
</tr>
</tbody>
</table>

Related Topics

See the following topics for further information related to the *AnalysisStmts* table:

- “IndexRecommendations” on page 573
- “SeqNumber” on page 615
- “INITIATE INDEX ANALYSIS” in *SQL Data Manipulation Language*
- “RESTART INDEX ANALYSIS” in *SQL Data Manipulation Language*
DataDemographics

Function

Contains table demographic information for use by the Teradata Index Wizard and Visual EXPLAIN client utilities.

Table Definition

The following CREATE TABLE request 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MachineName</td>
<td>• The name of the system to which TableName belongs.</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table.</td>
</tr>
<tr>
<td>TableName</td>
<td>• The name of the table on which demographic data has been collected.</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table.</td>
</tr>
<tr>
<td>DatabaseName</td>
<td>• The name of the containing database for TableName.</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table.</td>
</tr>
<tr>
<td>DBSize</td>
<td>The size in KB of data blocks in TableName.</td>
</tr>
<tr>
<td>CollectedTime</td>
<td>The timestamp value when the data demographics were collected.</td>
</tr>
</tbody>
</table>
### Data Demographics

Data Demographics does not contain information about the spatial distribution of tables across the AMPs.

You can view those details by querying the system view `DBC.TableSizeVX`.

### Related Topics

See the following chapter and manuals for more information about the Teradata Index Wizard and Teradata Visual Explain utilities:

- *Teradata Index Wizard User Guide*
- *Teradata Visual Explain User Guide*
- Chapter 7: “Database Foundations for the Teradata Index and Statistics Wizards”

### Spatial Distribution of a Table Across AMPs

Data Demographics does not contain information about the spatial distribution of tables across the AMPs.

You can view those details by querying the system view `DBC.TableSizeVX`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<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 <code>AMPNumber</code> 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 <code>QueryID</code> depends on how the demographics are captured.</td>
</tr>
<tr>
<td></td>
<td><strong>IF demographics are captured by this statement ...</strong></td>
</tr>
<tr>
<td></td>
<td><strong>THEN QueryID is ...</strong></td>
</tr>
<tr>
<td></td>
<td>COLLECT DEMOGRAPHICS null.</td>
</tr>
<tr>
<td></td>
<td>INSERT EXPLAIN the unique ID of the query.</td>
</tr>
<tr>
<td>IndexName</td>
<td>A comma-separated list of the names of the primary and secondary indexes for the table.</td>
</tr>
<tr>
<td></td>
<td>If an index has no name, then it is represented in this column by a comma-separated list of the names of the columns that compose it.</td>
</tr>
<tr>
<td></td>
<td>DemographicsID Set to 1 if the demographics are captured by a COLLECT DEMOGRAPHICS or INSERT EXPLAIN AND DEMOGRAPHICS statement.</td>
</tr>
<tr>
<td></td>
<td>1 indicates that the capture is on the system on which the row is inserted.</td>
</tr>
<tr>
<td></td>
<td><em>DemographicsID</em> has different values if the demographics are imported rather than captured directly.</td>
</tr>
</tbody>
</table>
Field

Function
Captures all the columns used or referenced in a captured query plan.

Table Definition
The following CREATE TABLE request 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   CHARACTER(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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FieldID</td>
<td>• Unique column identifier within a specific table.</td>
</tr>
<tr>
<td></td>
<td>• Partial USI for the table.</td>
</tr>
<tr>
<td>RelationKey</td>
<td>• Unique identifier for the relation in which the column is defined within</td>
</tr>
<tr>
<td></td>
<td>a certain database.</td>
</tr>
<tr>
<td></td>
<td>• NUPI for the table.</td>
</tr>
<tr>
<td></td>
<td>• Partial USI for the table.</td>
</tr>
<tr>
<td>Name</td>
<td>The name of the captured column.</td>
</tr>
</tbody>
</table>
### Chapter 6: Query Capture Facility

#### Field

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
</table>
| FldAlias              | • Alias name for the column.  
                        | • Null if none exists.       |
| QueryID               | The unique ID for the query. |
| ValueAccessFrequency  | The number of times the column is used with an equality condition. |
| JoinAccessFrequency   | The number of times the column is used in a join condition. |
| RangeAccessFrequency  | The number of times the column is used in a range condition. |
| ChangeRate            | The change rating value for the column. |
|                       | **IF the column is ...** | **THEN ChangeRate is set to this value ...** |
| modified              | 1           |
| not modified          | 0           |
| DataLength            | The maximum length of the column. |
| StatsKind             | 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 column 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 or the SQL DIAGNOSTIC “Validate Index” statement.  
                        | Statistics for the column being analyzed are retrieved from the QCD. |
|                       | **Code** | **Description** |
| P                     | Dictionary statistics. |
| S                     | QCD statistics. |
| null                  | No statistics. |
| NumNulls              | Number of nulls for the column or index.  
                        | The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available. |
| NumIntervals          | Number of intervals for the column or index.  
                        | The data for this column is retrieved from interval 0 of the available statistics. The column 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 column is set to null if no statistics are available. |
### Attribute Table

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModeValue</td>
<td>Modal value for the interval. The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available.</td>
</tr>
<tr>
<td>ModeFreq</td>
<td>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 column is set to null if no statistics are available.</td>
</tr>
<tr>
<td>TotalValues</td>
<td>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 column is set to null if no statistics are available.</td>
</tr>
<tr>
<td>TotalRows</td>
<td>Cardinality of the table. The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available.</td>
</tr>
</tbody>
</table>

### Related Topics

See the following chapter and manuals for more information about the Teradata Index Wizard utility.

- *Teradata Index Wizard User Guide*
- *Chapter 7: “Database Foundations for the Teradata Index and Statistics Wizards”*
**Index_Field**

**Function**

Captures all the index columns used in the query plan.

**Table Definition**

The following CREATE TABLE request defines the `Index_Field` table:

```sql
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
</table>
| RelationKey | • Unique identifier for the relation in which the captured index columns 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 column identifier within a specific table.  
             • Partial USI for the table. |
IndexColumns

Function
Captures the columns that form the index identified by IndexID during index analysis by the Teradata Index Wizard.

See “JoinIndexColumns” on page 582 for the definition of a QCD table that captures similar information for join index recommendations.

Table Definition
The following CREATE TABLE request defines the IndexColumns table:

```sql
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkloadID</td>
<td>Uniquely identifies the workload analyzed to create this secondary index recommendation.</td>
</tr>
</tbody>
</table>
| 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 index 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. |
IndexMaintenance

Function

*IndexMaintenance* stores the estimated costs incurred by the INSERT, UPDATE, MERGE, or DELETE requests in maintaining the recommended indexes stored in *IndexRecommendations*.

*IndexMaintenance* contains one row for each SQL request-index combination where maintenance costs are required for the index. You can query this table directly to retrieve information about the estimated cost of maintaining the indexes recommended by the Teradata Index Wizard for a given workload.

Table Definition

The following CREATE TABLE statement defines the *IndexMaintenance* table:

```sql
CREATE TABLE IndexMaintenance (  
  WorkLoadID INTEGER NOT NULL,  
  RecommendationID INTEGER NOT NULL,  
  IndexNameTag VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,  
  SQLStatementID INTEGER NOT NULL,  
  SecondaryIndexID INTEGER DEFAULT NULL,  
  BaseTableID BYTE(6) NOT NULL,  
  BaseTableName VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,  
  DatabaseName VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,  
  IndexType INTEGER NOT NULL,  
  IndexTypeText VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC,  
  JINumber INTEGER DEFAULT NULL,  
  MaintCosts FLOAT DEFAULT 0)  
PRIMARY INDEX (WorkloadID);
```

Attribute Definitions

The following table defines the *IndexMaintenance* table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkloadID</td>
<td>• Uniquely identifies the workload analyzed to create the attribute data.</td>
</tr>
<tr>
<td></td>
<td>• NUPI for the table.</td>
</tr>
<tr>
<td>RecommendationID</td>
<td>Uniquely identifies the recommendation ID in the <em>IndexRecommendations</em></td>
</tr>
<tr>
<td></td>
<td>table.</td>
</tr>
<tr>
<td>IndexNameTag</td>
<td>User-specified name for the index analysis.</td>
</tr>
<tr>
<td>BaseTableID</td>
<td>The ID of the base table that is being updated.</td>
</tr>
<tr>
<td>BaseTableName</td>
<td>The name for the base table referenced by BaseTableID.</td>
</tr>
</tbody>
</table>
### Attribute Table

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JINumber</td>
<td>The sequentially assigned number for the recommended join index in the IndexRecommendations table. This column is applicable only if the value for the IndexType column corresponds to a valid join index type.</td>
</tr>
<tr>
<td>SQLStatementID</td>
<td>The value for QueryID in the Query table, which is a unique identifier for the query generated by the system when the query plan is captured.</td>
</tr>
<tr>
<td>SecondaryIndexID</td>
<td>The IndexID of the recommended secondary index in the IndexRecommendations table. This column is applicable only if the value for the IndexType column corresponds to a valid secondary index type.</td>
</tr>
<tr>
<td>IndexType</td>
<td>A number that identifies the type of index recommended. Each unique index type is associated with its own IndexTypeText. The valid IndexType codes and their corresponding IndexTypeText strings are as follows:</td>
</tr>
<tr>
<td>IndexTypeText</td>
<td>The textual representation of IndexType. The valid IndexTypeText strings and their meanings are as follows:</td>
</tr>
</tbody>
</table>

#### IndexType Code Table

<table>
<thead>
<tr>
<th>IndexType Code</th>
<th>IndexTypeText</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unique Secondary Index (USI)</td>
</tr>
<tr>
<td>2</td>
<td>Value-Ordered Secondary Index (VOSI)</td>
</tr>
<tr>
<td>3</td>
<td>Hash-Ordered Secondary Index (HOSI)</td>
</tr>
<tr>
<td>4</td>
<td>Nonunique Secondary Index (NUSI)</td>
</tr>
<tr>
<td>5</td>
<td>Simple Join Index (JI)</td>
</tr>
<tr>
<td>6</td>
<td>Aggregate Join Index (JIAGG)</td>
</tr>
</tbody>
</table>

#### IndexType Text Table

<table>
<thead>
<tr>
<th>IndexType Text</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOSI</td>
<td>Hash-Ordered Secondary Index</td>
</tr>
<tr>
<td>JI</td>
<td>Simple Join Index</td>
</tr>
<tr>
<td>JIAGG</td>
<td>Aggregate Join Index</td>
</tr>
<tr>
<td>NUSI</td>
<td>Nonunique Secondary Index</td>
</tr>
<tr>
<td>USI</td>
<td>Unique Secondary Index</td>
</tr>
<tr>
<td>VOSI</td>
<td>Value-Ordered Secondary Index</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaintCosts</td>
<td>The estimated cost in milliseconds to update the recommended index structures.</td>
</tr>
</tbody>
</table>
IndexRecommendations

Function

Contains information about the index recommendations made by the Teradata Index Wizard utility.

You can query this table to retrieve the index definitions the Index Wizard recommends. IndexRecommendations also records the options specified during index analysis for later retrieval.

Table Definition

The following CREATE TABLE request defines the IndexRecommendations table:

```sql
CREATE TABLE IndexRecommendations(
    WorkLoadID INTEGER NOT NULL,
    UserName VARCHAR(30) CHARACTER SET UNICODE NOT NULL,
    TimeOfAnalysis TIMESTAMP(0) 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 CHARACTER(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 SMALLINT DEFAULT NULL,
    SearchSpaceSize SMALLINT,
    ChangeRateThreshold BYTEINT,
    ColumnPerIndex SMALLINT,
    ColumnsPerJoinIndex SMALLINT DEFAULT NULL,
);```
### Attribute Definitions

The following table defines the `IndexRecommendations` table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkloadID</td>
<td>• Uniquely identifies the workload. • NUPI for the table.</td>
</tr>
<tr>
<td>UserName</td>
<td>Name of the user performing the index analysis.</td>
</tr>
<tr>
<td>TimeOfAnalysis</td>
<td>The timestamp when the index recommendations were analyzed. You can compare this column with the modified timestamp for <code>TableName</code> to verify the correctness of the recommendation before applying it to the system.</td>
</tr>
<tr>
<td>RecommendationID</td>
<td>Uniquely identifies a set of index recommendations.</td>
</tr>
<tr>
<td>QueryID</td>
<td>Uniquely identifies the <code>QueryID</code> of the workload for which the current entry is an index recommendation.</td>
</tr>
<tr>
<td>IndexID</td>
<td>Uniquely identifies a unique secondary index recommended by the Teradata Index Wizard for a table. Set null, meaning not applicable, when the value of <code>IndexType</code> is 5 or 6, indicating a join index.</td>
</tr>
<tr>
<td>IndexNameTag</td>
<td>Name of the index recommendation as specified in the <code>INITIATE INDEX ANALYSIS</code> statement (see <code>SQL Data Manipulation Language</code>).</td>
</tr>
<tr>
<td>TableName</td>
<td>Name of the table for which the row defines an index recommendation.</td>
</tr>
<tr>
<td>DatabaseName</td>
<td>Name of the database containing <code>TableName</code>.</td>
</tr>
<tr>
<td>TableID</td>
<td>The unique internal identifier for <code>TableName</code>.</td>
</tr>
</tbody>
</table>
### Attribute | Definition
--- | ---
IndexType | A number that identifies the type of index recommended. Each unique index type is associated with its own `IndexTypeText`.

<table>
<thead>
<tr>
<th><code>IndexType Code</code></th>
<th><code>IndexTypeText</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unique Secondary Index (USI)</td>
</tr>
<tr>
<td>2</td>
<td>Value-Ordered Secondary Index (VOSI)</td>
</tr>
<tr>
<td>3</td>
<td>Hash-Ordered Secondary Index (HOSI)</td>
</tr>
<tr>
<td>4</td>
<td>Nonunique Secondary Index (NUSI)</td>
</tr>
<tr>
<td>5</td>
<td>Simple Join Index (JI)</td>
</tr>
<tr>
<td>6</td>
<td>Aggregate Join Index (JIAGG)</td>
</tr>
</tbody>
</table>

IndexTypeText | The textual representation of `IndexType`.
The valid `IndexTypeText` strings and their meanings are as follows:

<table>
<thead>
<tr>
<th><code>IndexTypeText</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOSI</td>
<td>Hash-Ordered Secondary Index</td>
</tr>
<tr>
<td>JI</td>
<td>Simple Join Index</td>
</tr>
<tr>
<td>JIAGG</td>
<td>Aggregate Join Index</td>
</tr>
<tr>
<td>NUSI</td>
<td>Nonunique Secondary Index</td>
</tr>
<tr>
<td>USI</td>
<td>Unique Secondary Index</td>
</tr>
<tr>
<td>VOSI</td>
<td>Value-Ordered Secondary Index</td>
</tr>
</tbody>
</table>

StatisticsInfo | The statistics, if any, used to make the index recommendations.

OriginalCost | The estimated cost of the query in milliseconds before implementing the recommended indexes.

NewCost | The estimated cost of the query in milliseconds after implementing the recommended indexes.

SpaceEstimate | The estimated space in bytes the recommended index occupies when created.

TimeEstimate | The estimated time in milliseconds required to implement the index recommendation.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DropFlag</td>
<td>Identifies whether the specified index is to be added or dropped.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</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>
<tr>
<td>IndexDDL</td>
<td>The DDL text of the CREATE INDEX, DROP INDEX, CREATE JOIN INDEX, or DROP JOIN INDEX request for the index recommended by the Teradata Index Wizard.</td>
</tr>
<tr>
<td>StatsDDL</td>
<td>The DDL text of the COLLECT STATISTICS (QCD form) requests 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>
<tr>
<td>SearchSpaceSize</td>
<td>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.</td>
</tr>
<tr>
<td>ChangeRateThreshold</td>
<td>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 <code>ChangeRateThreshold</code> is available for selection as a candidate index during index analysis.</td>
</tr>
<tr>
<td>ColumnsPerIndex</td>
<td>The maximum number of columns permissible in the index as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis.</td>
</tr>
<tr>
<td>ColumnsPerJoinIndex</td>
<td>The integer value used during analysis to control the maximum number of columns in a recommended Join Index as specified by the Teradata Index Wizard or by the INITIATE INDEX ANALYSIS SET <code>boundary_option</code> specification.</td>
</tr>
<tr>
<td>IndexMaintMode</td>
<td>The integer value used during analysis to control how estimated index maintenance costs are used. The value is specified by the Teradata Index Wizard or using the INITIATE INDEX ANALYSIS SET <code>boundary_option</code> specification.</td>
</tr>
<tr>
<td>JINumber</td>
<td>An integer value sequence number that identifies the recommended Join Index table for a given index analysis. The column is set to null for index types other than Join Index.</td>
</tr>
</tbody>
</table>
**JITableName**

A system-assigned name for the Join Index.

If **JINumber** is null, then so is **JITableName**.

The naming convention for **JITableName** is as follows:

\[JI\_RecommendationID\_BaseTableName\_JINumber,\]

where **JI** is a literal string and the values for **RecommendationID**, **BaseTableName**, and **JINumber** are those used in the eponymously named columns of the row, converted to character format where necessary.

**TimeLimit**

A user-specified time limit, in whole number of minutes, for the duration of the analysis.

**TimeLimitExceeded**

Indicates whether or not the recommendation was generated by an INITIATE INDEX ANALYSIS request whose specified time limit expired.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>The partition analysis was not interrupted because the specified INITIATE INDEX ANALYSIS time limit was exceeded. This is the default.</td>
</tr>
<tr>
<td>T</td>
<td>The partition analysis was interrupted because the specified INITIATE INDEX ANALYSIS time limit was exceeded. This means that the final recommendations might have been affected.</td>
</tr>
</tbody>
</table>
Chapter 6: Query Capture Facility

IndexTable

Function

Describes all indexes on the tables specified by the query.

Table Definition

The following CREATE TABLE request defines IndexTable:

```
CREATE TABLE IndexTable (  
  IndexNum     INTEGER NOT NULL,  
  RelationKey  INTEGER NOT NULL,  
  OrderBy      CHARACTER(1) NOT NULL,  
  AccessInfo   CHARACTER(1) NOT NULL,  
  FieldOnly    CHARACTER(1) NOT NULL,  
  RangeConstraint CHARACTER(1) NOT NULL,  
  IndexFlag    CHARACTER(1),  
  IndexName    VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC,  
  IndexType    CHARACTER(1),  
  UniqueFlag   CHARACTER(1),  
  IndexKind    CHARACTER(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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexNum</td>
<td>• Unique identifier for the captured index.</td>
</tr>
<tr>
<td></td>
<td>• Partial USI for the table.</td>
</tr>
<tr>
<td>RelationKey</td>
<td>• Unique identifier for the relation in which the captured index is defined.</td>
</tr>
<tr>
<td></td>
<td>• NUPI for the table.</td>
</tr>
<tr>
<td></td>
<td>• Partial USI for the table.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OrderBy</td>
<td>Defines whether the index has an associated ORDER BY clause.</td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td><strong>Description</strong></td>
</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>
<tr>
<td>AccessInfo</td>
<td>Specifies if the index is a covering index, bit map, or neither.</td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td><strong>Description</strong></td>
</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>
<tr>
<td>Field1Only</td>
<td>Defines whether the index is a join index and Field1 is the only part needed.</td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>F</td>
<td>Not a join index requiring Field1 only. This generally means one of two things:</td>
</tr>
<tr>
<td></td>
<td>• The index is not a join index.</td>
</tr>
<tr>
<td></td>
<td>• The index is a compressed join index.</td>
</tr>
<tr>
<td>T</td>
<td>Join index requiring Field1 only.</td>
</tr>
<tr>
<td></td>
<td>This means that the join index is not compressed.</td>
</tr>
<tr>
<td>RangeConstraint</td>
<td>Flag for value-ordered indexes that have a range constraint used by the query plan.</td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>F</td>
<td>There is no range constraint on the index.</td>
</tr>
<tr>
<td></td>
<td>The flag is set to F whether the index is used in the plan or not.</td>
</tr>
<tr>
<td>T</td>
<td>There is a range constraint on the value-ordered index used in the plan.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>IndexFlag</td>
<td>Flag indicating whether the index was used in the query plan.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</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.</td>
</tr>
<tr>
<td></td>
<td><em>IndexFlag</em> is also set to T if a subset of the partitions of a partitioned primary index is accessed because of partition elimination.</td>
</tr>
<tr>
<td>IndexName</td>
<td>• The name of the index if it has one.</td>
</tr>
<tr>
<td></td>
<td>• Null if this is not a named index.</td>
</tr>
<tr>
<td>IndexType</td>
<td>Flag indicating the type of the index.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>H</td>
<td>Hash-ordered secondary covering index.</td>
</tr>
<tr>
<td>J</td>
<td>Join index.</td>
</tr>
<tr>
<td>K</td>
<td>Primary key.</td>
</tr>
<tr>
<td>N</td>
<td>Hash index.</td>
</tr>
<tr>
<td>O</td>
<td>Value-ordered secondary 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>
<tr>
<td>U</td>
<td>Unique constraint.</td>
</tr>
<tr>
<td>V</td>
<td>Value-ordered secondary index.</td>
</tr>
<tr>
<td>UniqueFlag</td>
<td>Flag indicating whether the index is unique or nonunique.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>F</td>
<td>Nonunique index.</td>
</tr>
<tr>
<td>T</td>
<td>Unique index.</td>
</tr>
</tbody>
</table>
Chapter 6: Query Capture Facility

IndexTable

SQL Request and Transaction Processing 581

IndexKind Flag indicating whether the index is permanent or simulated.
Indexes are simulated using the index validation function of the Teradata Index Wizard (see "Index and Partitioning Expression Validation" on page 652).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>TotalValues</td>
<td>The total number of values in the index other than the modal value.</td>
</tr>
<tr>
<td>TotalRows</td>
<td>The cardinality of the table.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Permanent index.</td>
</tr>
<tr>
<td>S</td>
<td>Simulated index.</td>
</tr>
</tbody>
</table>
Function
Captures the columns that form the join index identified by JINumber during index analysis by the Teradata Index Wizard.

See “IndexColumns” on page 570 for the definition of the QCD table that captures similar information for secondary index recommendations.

Table Definition
The following CREATE TABLE request defines JoinIndexColumns:

```sql
CREATE TABLE JoinIndexColumns (
    WorkLoadID INTEGER NOT NULL,
    RecommendationID INTEGER NOT NULL,
    TableID BYTE(6) NOT NULL,
    JINumber INTEGER NOT NULL,
    ColumnName VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC DEFAULT NULL,
    AliasName VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC DEFAULT NULL,
    Field1Flag CHARACTER(1) DEFAULT NULL,
    Field2Flag CHARACTER(1) DEFAULT NULL,
    RowIDFlag CHARACTER(1) DEFAULT NULL,
    AggregateFunc BYTEINT DEFAULT NULL,
    PrimaryIndexPosition BYTEINT DEFAULT NULL,
    GroupByPosition BYTEINT DEFAULT NULL
) PRIMARY INDEX (RecommendationID, TableID, JINumber);
```

Attribute Definitions
The following table defines the JoinIndexColumns table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkLoadID</td>
<td>Uniquely identifies the workload analyzed to create this join index recommendation.</td>
</tr>
</tbody>
</table>
| RecommendationID | • Uniquely identifies a set of index recommendations in the IndexRecommendations table.  
                  | • Partial NUPI for the table.                                                |
| TableID          | • The unique internal identifier in the IndexRecommendations table for the base table on which the join index is defined.  
<pre><code>              | • Partial NUPI for the table.                                               |
</code></pre>
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
</table>
| JINumber     | • The system-assigned sequence number for the join index in the IndexRecommendations table.  
               • Partial NUPI for the table.                                                |
| ColumnName   | Name of the join index column.                                               |
| AliasName    | The correlation name assigned to a column or aggregate function in the join index definition. |
| Field1Flag   | Indicates whether the column is part of the column_1 (non-compressed columns) select list in the join index definition. |
| Field2Flag   | Indicates whether the column is part of the column_2 (compressed columns) select list in the join index definition. |
| RowIDFlag    | Indicates whether the value for the column is the reserved word ROWID.       |

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field1Flag</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>F</td>
<td>False. The column is not part of the column_1 select list.</td>
</tr>
<tr>
<td>T</td>
<td>True. The column is part of the column_1 select list.</td>
</tr>
<tr>
<td>Field2Flag</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>F</td>
<td>False. The column is not part of the column_2 select list.</td>
</tr>
<tr>
<td>T</td>
<td>True. The column is part of the column_2 select list.</td>
</tr>
<tr>
<td>RowIDFlag</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>F</td>
<td>False. The value for the column is not ROWID.</td>
</tr>
<tr>
<td>T</td>
<td>True. The value for the column is ROWID.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AggregateFunc</td>
<td>Indicates whether an aggregate function is applied to ColumnName and, if so, the type of aggregation performed.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>No aggregation.</td>
</tr>
<tr>
<td>1</td>
<td>The column is used in an aggregate SUM operation.</td>
</tr>
<tr>
<td>2</td>
<td>The column is used in an aggregate COUNT operation.</td>
</tr>
<tr>
<td>3</td>
<td>The column is used in a COUNT(*) operation.</td>
</tr>
<tr>
<td>PrimaryIndexPosition</td>
<td>Indicates whether ColumnName is a component of the primary index for the join index.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>The column is not part of the primary index definition for the join index.</td>
</tr>
<tr>
<td>&gt;1</td>
<td>The column is part of the primary index definition for the join index. The value represents the position of ColumnName within the primary index definition for the join index.</td>
</tr>
<tr>
<td>GroupByPosition</td>
<td>Indicates whether ColumnName is a component of the GROUP BY clause in the join index definition.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>The column is not part of the GROUP BY specification for the join index.</td>
</tr>
<tr>
<td>&gt;1</td>
<td>The column is part of the GROUP BY specification for the join index. The value represents the position of ColumnName within the GROUP BY specification for the join index.</td>
</tr>
</tbody>
</table>
Chapter 6: Query Capture Facility

PartitionRecommendations

Function

Captures the recommended partitioning expressions generated by an INITIATE PARTITION ANALYSIS statement and identified by a WorkLoadID value.

PartitionRecommendations contains one row for each combination of query and table partitioning where the new Optimizer plan from the recommended partitioning either benefits or degrades workload performance as a result of the new partitioning.

If a particular table partitioning impacts more than one query, INITIATE PARTITION ANALYSIS writes multiple rows in the table to identify each affected query. The recommended PARTITION BY expression is stored as SQL text in the ExpressionText column.

See “RangePartExpr” on page 607 for the definition of the QCD table that captures related detail information about recommended partitioning expressions.

Table Definition

The following CREATE TABLE request defines PartitionRecommendations:

```sql
CREATE SET TABLE qcd.partitionrecommendations (
    WorkLoadID  INTEGER NOT NULL,
    UserName     VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
    TimeOfAnalysis  TIMESTAMP(6) NOT NULL,
    RecommendationID INTEGER NOT NULL,
    QueryID         INTEGER NOT NULL,
    ResultNameTag   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,
    ExpressionText  VARCHAR(10000) CHARACTER SET UNICODE NOT CASESPECIFIC,
    ExpressionType  CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC,
    RecreateText    VARCHAR(15000) CHARACTER SET UNICODE NOT CASESPECIFIC,
    OriginalCost    FLOAT,
    NewCost         FLOAT,
    SpaceEstimate   FLOAT,
    TimeEstimate    FLOAT,
    StatsDDL        VARCHAR(10000) CHARACTER SET UNICODE NOT CASESPECIFIC,

);```
Attribute Definitions

The following table defines the `PartitionRecommendations` table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkLoadID</td>
<td>An identifier of the workload that was analyzed.</td>
</tr>
<tr>
<td>UserName</td>
<td>Name of the user performing the partition analysis.</td>
</tr>
<tr>
<td>TimeOfAnalysis</td>
<td>A timestamp recording the time at which this analysis completed.</td>
</tr>
<tr>
<td>RecommendationID</td>
<td>An identifier for the set of partitioning recommendations this recommendation belongs to.</td>
</tr>
<tr>
<td>QueryID</td>
<td>Unique identifier for the query that benefits from this recommendation.</td>
</tr>
<tr>
<td>ResultNameTag</td>
<td>User-assigned name for the set of recommendations identified by <code>RecommendationID</code>.</td>
</tr>
<tr>
<td>TableName</td>
<td>Name of the table on which the partitioning recommendation is being made.</td>
</tr>
<tr>
<td>DatabaseName</td>
<td>Name of the database containing <code>TableName</code>.</td>
</tr>
<tr>
<td>TableID</td>
<td>Unique internal identifier for <code>TableName</code>.</td>
</tr>
<tr>
<td>ExpressionText</td>
<td>SQL text of the recommended PARTITION BY expression.</td>
</tr>
<tr>
<td>ExpressionType</td>
<td>The form of partitioning used by <code>ExpressionText</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>The partitioning expression is built from a RANGE_N function.</td>
</tr>
</tbody>
</table>

RecreateText SQL text of statements that will recreate the table with the recommended partitioning expression.

The statements assume the table is populated and so require the current rows to be temporarily stored in another table.

**Caution:** Do not execute these SQL requests without first verifying that they are compliant with your system requirements (see "Index and Partitioning Expression Creation Process" on page 655).

OriginalCost The estimated cost in milliseconds to execute the query without the recommendation.
### Attribute | Definition
--- | ---
NewCost | The estimated cost in milliseconds to execute the query with the recommendation. If this value is higher than *OriginalCost*, the recommendation has negatively impacted this query.
SpaceEstimate | The additional incremental space, in bytes, needed to store the table with the recommended partitioning.
TimeEstimate | Reserved for future use. Estimated time, in milliseconds, to recreate the table with the recommended partitioning.
StatsDDL | SQL text for collecting statistics on the recommended partitioning expression.
Remarks | Additional details regarding the recommendation.
AnalysisTimeLimit | User-specified time limit, in whole minutes, for the duration of the analysis.
TimeLimitExceeded | Indicates whether or not the recommendation was generated by an INITIATE PARTITION ANALYSIS request whose specified time limit expired.

### Code | Description
--- | ---
F | The partition analysis was *not* interrupted because the specified INITIATE PARTITION ANALYSIS time limit was exceeded. This is the default.
T | The partition analysis was interrupted because the specified INITIATE PARTITION ANALYSIS time limit was exceeded. This means that the final recommendations might have been affected.

AnalysisData | Reserved for future use.
Example: Recommended Single-Range Partition

The following INITIATE PARTITION ANALYSIS request makes the entries in PartitionRecommendations reported by the query that follows it.

```sql
INITIATE PARTITION ANALYSIS ON recent_orders
FOR MyWorkload
IN MyQCD AS IPA_recent_orders;

SELECT ExpressionText
FROM MyQCD.PartitionRecommendations
WHERE ResultNameTag = 'IPA_recent_orders';

ExpressionText
-----------------
PARTITION BY RANGE_N (order_date
    BETWEEN DATE '2004-01-01'
    AND DATE '2005-12-31'
    EACH INTERVAL '1' MONTH)
```

Note that this is the PartitionRecommendations component of “Example: Recommended Single-Range Partition” on page 608.
Predicate

Function

Describes any index, join, or residual conditions applied for specific AMP steps in a query.

Table Definition

The following CREATE TABLE request defines the Predicate table:

```
CREATE TABLE Predicate (
    PredicateID   INTEGER NOT NULL,
    StepID        INTEGER NOT NULL,
    PredicateKind CHARACTER(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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PredicateID</td>
<td>• Unique identifier for the predicate.</td>
</tr>
<tr>
<td></td>
<td>• UPI for the table.</td>
</tr>
<tr>
<td>StepID</td>
<td>Unique identifier for the AMP step.</td>
</tr>
<tr>
<td>PredicateKind</td>
<td>Describes the kind of predicate condition associated with this step.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Additional join condition.</td>
</tr>
<tr>
<td>G</td>
<td>Range constraint.</td>
</tr>
<tr>
<td></td>
<td>Used for value-ordered relations.</td>
</tr>
<tr>
<td>I</td>
<td>Condition associated with an index.</td>
</tr>
<tr>
<td>J</td>
<td>Join condition.</td>
</tr>
<tr>
<td>L</td>
<td>Condition on left relation in a join.</td>
</tr>
<tr>
<td>Q</td>
<td>Partition elimination occurs for a source condition.</td>
</tr>
<tr>
<td></td>
<td>This is a residual condition on the left or right table in a join or on a</td>
</tr>
<tr>
<td></td>
<td>single-table retrieval. Partition elimination occurs prior to accessing</td>
</tr>
<tr>
<td></td>
<td>the rows, so the condition applies only to rows retrieved from</td>
</tr>
<tr>
<td></td>
<td>partitions that were not eliminated.</td>
</tr>
</tbody>
</table>
### Chapter 6: Query Capture Facility

#### Predicate

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PredicateKind</td>
<td>(continued)</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</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>
<tr>
<td>PredicateText</td>
<td>Full text of the predicate as it appears in the EXPLAIN report.</td>
</tr>
</tbody>
</table>
Predicate_Field

Function

Associates the list of columns specified in a captured predicate with the parent relation and predicate.

Table Definition

The following CREATE TABLE request 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PredicateID</td>
<td>Uniquely identifies the predicate.</td>
</tr>
<tr>
<td>RelationKey</td>
<td>• Unique identifier for the referenced table.</td>
</tr>
<tr>
<td></td>
<td>• NUPI for the Predicate_Field table.</td>
</tr>
<tr>
<td>FieldID</td>
<td>Unique identifier for the column.</td>
</tr>
</tbody>
</table>
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 request defines the QryRelX table:

```sql
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>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RowType</td>
<td>Defines whether the row handles text overflow from the Query table or from the Relation table.</td>
</tr>
<tr>
<td>Code</td>
<td>Definition</td>
</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>
<tr>
<td>KeyValue</td>
<td>Defines the implicit primary key for QryRelX rows.</td>
</tr>
<tr>
<td>IF RowType is this value ...</td>
<td>THEN the value is the ...</td>
</tr>
<tr>
<td>Q</td>
<td>QueryID</td>
</tr>
<tr>
<td>R</td>
<td>RelationKey</td>
</tr>
<tr>
<td>Attribute</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>SeqNumber</td>
<td>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 a unique SeqNumber value. The value of SeqNumber begins at 1 and is incremented by 1 for each new text row required.</td>
</tr>
<tr>
<td>Text</td>
<td>The overflow QueryText or TableDDL text. The upper bound for this text is 30,000 characters per row up to a total of 1 megabyte of overflow SQL query text.</td>
</tr>
</tbody>
</table>
Chapter 6: Query Capture Facility

Query

Function

Describes information about captured queries.

Table Definition

The following CREATE TABLE request 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueryID</td>
<td>• Unique identifier for the query generated by the system when the query plan is captured.</td>
</tr>
<tr>
<td></td>
<td>• UPI for the table.</td>
</tr>
<tr>
<td>UDB_Key</td>
<td>Identifier for the user who captured the plan by submitting either a DUMP EXPLAIN or INSERT EXPLAIN statement (see SQL Data Definition Language for descriptions of these statements).</td>
</tr>
<tr>
<td>MachName</td>
<td>Name of the test machine on which the query plan is captured.</td>
</tr>
</tbody>
</table>
## Query

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumAMPs</td>
<td>Number of AMPs in the configuration.</td>
</tr>
<tr>
<td>NumPEs</td>
<td>Number of PEs in the configuration.</td>
</tr>
<tr>
<td>NumNodes</td>
<td>Number of nodes in the configuration.</td>
</tr>
<tr>
<td>ReleaseInfo</td>
<td>Teradata Database release number under which the query was captured.</td>
</tr>
<tr>
<td></td>
<td>The value is defined in <code>DBC.DBCInfo</code>.</td>
</tr>
<tr>
<td>VersionInfo</td>
<td>Teradata Database version number under which the query was captured.</td>
</tr>
<tr>
<td></td>
<td>The value is defined in <code>DBC.DBCInfo</code>.</td>
</tr>
<tr>
<td>PENum</td>
<td>Number of the parsing engine on which the query was processed.</td>
</tr>
<tr>
<td>QueryText</td>
<td>The SQL DML text of the captured query.</td>
</tr>
<tr>
<td></td>
<td>If the text exceeds the upper limit of 20,000 characters, then it overflows to the <code>QryRelX</code> table. See “Overflow” on page 596 and “QryRelX” on page 592.</td>
</tr>
<tr>
<td>DateTimeStamp</td>
<td>Timestamp that identifies when the captured query was performed.</td>
</tr>
<tr>
<td></td>
<td>Useful for distinguishing among multiple performances of the same query on the same machine under the same software version and release.</td>
</tr>
<tr>
<td>QueryName</td>
<td>The name of the query, if provided, as specified in the AS clause.</td>
</tr>
<tr>
<td>Frequency</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Chapter 6: Query Capture Facility

#### Query

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatementTypes</td>
<td>A comma-separated list of 3-character statement type codes.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</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>MRG</td>
<td>MERGE 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>
</tbody>
</table>

| DefaultDBName   | The name of the default database at the time the query plan is captured.   |
| Overflow        | Specifies whether QueryText exceeds 20 000 characters.                     |
|                 | If QueryText exceeds 20 000 characters, then all text beyond that boundary is truncated. |
| Flag            | Description                                                                |
| F               | QueryText <= 20 000 characters.                                            |
| T               | QueryText > 20 000 characters and has been truncated.                       |
### Chapter 6: Query Capture Facility

#### Query

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Identifies whether Query stores the complete query text or a truncated version.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 SQL Data Manipulation Language). 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 592).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ValidatedPlan</th>
<th>Specifies whether the plan was captured in validation or non-validation mode. The value is always set to F during the capture.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ImportedPlan</th>
<th>Specifies whether the query plan was captured on the current system or imported from another system. The system always sets the flag to F during the capture.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
QuerySteps

Function

Each row in the table lists the attributes of any step executed by the system corresponding to the request. 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. In this case, the individual rows are distinguished by their row type codes.

Table Definition

The following CREATE TABLE request defines the QuerySteps table:

```sql
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         CHARACTER(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC NOT NULL,
    StepKind        CHARACTER(2) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
    ParallelKind    CHARACTER(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
    AMPUsage        CHARACTER(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
    TriggerType     CHARACTER(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
    EstCPUCost      FLOAT,
    EstIOCost       FLOAT,
    EstNetworkCost  FLOAT,
    EstHRCost       FLOAT
     Cost           FLOAT,
    MaxCost         FLOAT,
    SourceRelation1 INTEGER,
    SourceRelation2 INTEGER,
    TargetRelation1 INTEGER,
    TargetRelation2 INTEGER,
    StepAttributeNameType CHARACTER(10) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
    StepAttributeValue VARCHAR(100) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
    LockType        CHARACTER(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
    RowHashFlag     CHARACTER(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
    NoWaitFlag      CHARACTER(1) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
    Cardinality     DECIMAL(18,0) DEFAULT 0,
    IndexMaintCostEst FLOAT DEFAULT 0

```
Primary index (QueryID)
INDEX SK_StepID (StepID);

**Attribute Definitions**

The following table defines the *QuerySteps* table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>StepID</td>
<td>• Unique identifier for the step.</td>
</tr>
<tr>
<td></td>
<td>• NUSI for the table.</td>
</tr>
<tr>
<td>QueryID</td>
<td>• Unique identifier for the query.</td>
</tr>
<tr>
<td></td>
<td>• NUPI for the table.</td>
</tr>
<tr>
<td>StepNum</td>
<td>The number of the step whose text is reported by this <em>QuerySteps</em> row.</td>
</tr>
<tr>
<td></td>
<td><strong>IF the step is ...</strong> THEN the value is ...</td>
</tr>
<tr>
<td></td>
<td>not performed in parallel the step number.</td>
</tr>
<tr>
<td></td>
<td>performed in parallel the number of the main step.</td>
</tr>
<tr>
<td>ParallelStepNum</td>
<td>The number of the parallel step whose text is reported by this <em>QuerySteps</em> row.</td>
</tr>
<tr>
<td></td>
<td><strong>IF the step is ...</strong> THEN the value is ...</td>
</tr>
<tr>
<td></td>
<td>not performed in parallel 0.</td>
</tr>
<tr>
<td></td>
<td>performed in parallel the number of the parallel step.</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.</td>
</tr>
<tr>
<td></td>
<td><strong>Code</strong> <strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>A The row describes attributes for a query plan step beyond the first row for that step, which is coded with a <em>RowType</em> of G.</td>
</tr>
<tr>
<td></td>
<td>The additional attributes are described by the <em>StepAttributeType</em> and <em>StepAttributeValue</em> columns. The remaining columns in an A row type are set null.</td>
</tr>
<tr>
<td></td>
<td>G 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>
</tbody>
</table>
## Chapter 6: Query Capture Facility

### QuerySteps

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>StepKind</td>
<td>Describes the kind of step characterized by this row.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 product 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>HF</td>
<td>Dynamic hash join 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>Attribute</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>StepKind</strong></td>
<td>Code Description</td>
</tr>
<tr>
<td></td>
<td>NJ</td>
</tr>
<tr>
<td></td>
<td>OJ</td>
</tr>
<tr>
<td></td>
<td>PJ</td>
</tr>
<tr>
<td></td>
<td>RJ</td>
</tr>
<tr>
<td></td>
<td>SA</td>
</tr>
<tr>
<td></td>
<td>SO</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td></td>
<td>SR</td>
</tr>
<tr>
<td></td>
<td>ST</td>
</tr>
<tr>
<td></td>
<td>SU</td>
</tr>
<tr>
<td></td>
<td>UP</td>
</tr>
<tr>
<td><strong>ParallelKind</strong></td>
<td>Describes whether the step can be done in parallel with its preceding step or not.</td>
</tr>
<tr>
<td></td>
<td>Code</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>S</td>
</tr>
<tr>
<td><strong>AMPUsage</strong></td>
<td>Describes how AMPs are used to process the step.</td>
</tr>
<tr>
<td></td>
<td>Code</td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>G</td>
</tr>
<tr>
<td></td>
<td>O</td>
</tr>
<tr>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>
Chapter 6: Query Capture Facility

QuerySteps

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriggerType</td>
<td>Describes triggering associated with this step.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

EstCPUCost | The CPU time for the step as estimated by the Optimizer, expressed in milliseconds. |
EstIOCost  | The I/O service time for the step as estimated by the Optimizer, expressed in milliseconds. |
EstNetworkCost | The BYNET service time for the step as estimated by the Optimizer, expressed in milliseconds. |
EstHRCost  | Other miscellaneous costs for the step as estimated by the Optimizer, expressed in milliseconds. |
Cost       | Estimated cost of performing this step, expressed in milliseconds. Similar to the time estimates provided by EXPLAIN reports, cost values should not be taken as absolute times, but rather as relative values that can be evaluated as proportions with respect to other cost estimates. |
MaxCost    | Estimated worst case cost of performing this step. |
SourceRelation1 | Number of the principal source relation in this step. For example, if the step is a join step and StepAttributeType is SingleRowL, then SourceRelation1 is the number of the left relation in the join. If StepAttributeType is SingleRowR, then SourceRelation1 is null and SourceRelation2 contains the number of the right relation in the join. |
SourceRelation2 | Number of an additional source relation in this step. |
TargetRelation1   | Number of the spool file or table for which the step operation results or acts upon. |
TargetRelation2   | Number of an additional spool file, if one is required, to hold additional results of this step. |
### Chapter 6: Query Capture Facility

#### QuerySteps

- **StepAttributeType**
  - Indicates the attribute that is described by the row.

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Description</th>
<th>Step Where Attribute Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginRQ</td>
<td>None.</td>
<td>First step of recursive block.</td>
</tr>
<tr>
<td>EndRQ</td>
<td>Step number of the corresponding BeginRQ.</td>
<td>Last step of recursive block.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GlobalFlag</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>F</td>
<td>Flag not set. Intermediate aggregate results computed locally.</td>
</tr>
<tr>
<td>T</td>
<td>Flag set. Intermediate aggregate results computed globally.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupKey</td>
<td>Grouping column.</td>
</tr>
<tr>
<td>IndexNum</td>
<td>Index number.</td>
</tr>
<tr>
<td>JoinType</td>
<td>Join type in the join step.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Full outer</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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Fraction</td>
</tr>
<tr>
<td>I</td>
<td>Fixed</td>
</tr>
</tbody>
</table>
## Attribute Type

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Description</th>
<th>Step Where Attribute Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeftIndex</td>
<td>Index number.</td>
<td>Index used in the left relation in the join step.</td>
</tr>
<tr>
<td>MergeMode</td>
<td></td>
<td>Merge delete and merge update steps indicating the merge type used in the step.</td>
</tr>
<tr>
<td>PartCount</td>
<td>Number of partitions used for the hash join.</td>
<td>Hash join step only.</td>
</tr>
<tr>
<td>RightIndex</td>
<td>Index number.</td>
<td>Index used in the right relation in the join step.</td>
</tr>
<tr>
<td>SingleRowL</td>
<td>Single-row optimization was applied to the left relation in the join step during a DUMP EXPLAIN or an INSERT EXPLAIN operation.</td>
<td>Join step.</td>
</tr>
<tr>
<td>SingleRowR</td>
<td>Single-row optimization was applied to the right relation in the join step during a DUMP EXPLAIN or an INSERT EXPLAIN operation.</td>
<td>Join step.</td>
</tr>
<tr>
<td>SMSKind</td>
<td>Intersect, Minus, Union</td>
<td>BitMap step indicating the set manipulation operation performed.</td>
</tr>
</tbody>
</table>

### MergeMode

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>Match row hash.</td>
</tr>
<tr>
<td>R</td>
<td>Match row ID.</td>
</tr>
<tr>
<td>W</td>
<td>Match whole row.</td>
</tr>
</tbody>
</table>
### StepAttributeType (continued)

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Description</th>
<th>Step Where Attribute Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceIndex</td>
<td>Index number.</td>
<td>Index used in the source relation in the Stat step.</td>
</tr>
<tr>
<td>StatOpt</td>
<td></td>
<td>Stat function step.</td>
</tr>
<tr>
<td>Svalue</td>
<td>Sample size.</td>
<td>Sample specified in the Sample step.</td>
</tr>
<tr>
<td>TableIndex</td>
<td>Index number.</td>
<td>Update step indicating the index the step used.</td>
</tr>
<tr>
<td>True or False</td>
<td>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>
</tbody>
</table>

### StepAttributeValue
Indicates the value of the attribute described by the row.

### LockType
Defines the severity of the lock specified by the query plan for this step.

See Chapter 9: “Locking and Transaction Processing” for information about lock severities.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

### RowHashFlag
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>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>The table is not locked on a row hash.</td>
</tr>
<tr>
<td>T</td>
<td>The table is locked on a row hash.</td>
</tr>
</tbody>
</table>
### NoWaitFlag

Indicates if the no wait option is set for the lock step.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>The no wait option is disabled.</td>
</tr>
<tr>
<td>T</td>
<td>The no wait option is enabled.</td>
</tr>
</tbody>
</table>

### Cardinality

The estimated number of output rows or affected rows estimated for this given step by the Optimizer. This value is applicable only to steps that retrieve or modify rows.

### IndexMaintCostEst

The cost estimated by the Index Wizard to maintain the indexes affected by this step, expressed in milliseconds. This value is applicable only to steps that modify rows.
Function

Stores the individual details of a recommended range partitioning expression based on the RANGE_N function. The intent of this table is to provide the Index Wizard with additional flexibility for displaying the recommended partitioning expression.

Each row in this table represents one range of a given RANGE_N function, where the range boundary values and range size are stored as SQL text strings. The set of rows belonging to a given RANGE_N function can be identified by its common values in columns WorkloadID, RecommendationID, and TableID, and can be ordered by a sequence value that is stored in the table.

See “PartitionRecommendations” on page 585 for information about the SQL text for recommended partitioning expressions.

Table Definition

The following CREATE TABLE request defines the RangePartExpr table:

```
CREATE SET TABLE RangePartExpr (  
    WorkLoadID       INTEGER NOT NULL,  
    RecommendationID INTEGER NOT NULL,  
    TableID          BYTE(6) NOT NULL,  
    TestColumn       VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,  
    RangeStart       VARCHAR(18) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,  
    RangeEnd         VARCHAR(18) CHARACTER SET UNICODE NOT CASESPECIFIC,  
    RangeSize        VARCHAR(50) CHARACTER SET UNICODE NOT CASESPECIFIC,  
    RangeSequence    INTEGER NOT NULL )  
PRIMARY INDEX (RecommendationID, TableID);
```  

Attribute Definitions

The following table defines the RangePartExpr table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkLoadID</td>
<td>An identifier of the workload that was analyzed.</td>
</tr>
<tr>
<td></td>
<td>A component of the NUPI on RangePartExpr.</td>
</tr>
<tr>
<td>RecommendationID</td>
<td>An identifier for the set of partition recommendations that this</td>
</tr>
<tr>
<td></td>
<td>recommendation belongs to.</td>
</tr>
<tr>
<td></td>
<td>A component of the NUPI on RangePartExpr.</td>
</tr>
</tbody>
</table>
Example: Recommended Single-Range Partition

The following INITIATE PARTITION ANALYSIS request makes the entries in RangePartExpr reported by the query that follows it.

```
INITIATE PARTITION ANALYSIS ON recent_orders
FOR MyWorkload
IN MyQCD AS IPA_recent_orders;
```

```
SELECT TestColumn, RangeStart, RangeEnd, RangeSize, RangeSequence
FROM MyQCD.RangePartExpr AS rpe,
     MyQCd.PartitionRecommendations AS pr
WHERE rpe.RecommendationId = pr.RecommendationId
AND     ResultNameTag = 'IPA_recent_orders';
```

```
<table>
<thead>
<tr>
<th>TestColumn</th>
<th>RangeStart</th>
<th>RangeEnd</th>
<th>RangeSize</th>
<th>RangeSequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>order_date</td>
<td>DATE '2004-01-01'</td>
<td>DATE '2005-12-31'</td>
<td>INTERVAL '1' MONTH</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Note that this is the RangePartExpr component of “Example: Recommended Single-Range Partition” on page 588.
Chapter 6: Query Capture Facility

Relation

Function

Describes all table and spool files in the access plan for the captured query.

Table Definition

The following CREATE TABLE request defines the Relation table:

```sql
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
  SpoolCompressedAllowed CHARACTER(1)
  SpoolSize     REAL)
PRIMARY INDEX (QueryID),
UNIQUE INDEX (RelationKey);
```

Attribute Definitions

The following table defines the Relation table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelationKey</td>
<td>• Unique identifier for the relation within its database.</td>
</tr>
<tr>
<td></td>
<td>• USI for the Relation table.</td>
</tr>
</tbody>
</table>
## Chapter 6: Query Capture Facility

### Relation

#### Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
</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 derived tables, global temporary tables, hash indexes, join indexes, permanent tables, volatile tables, and spool files. |

#### RelationKind

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Relation is a derived table. The Name attribute contains the name of the derived table.</td>
</tr>
<tr>
<td>G</td>
<td>Relation is a global temporary table.</td>
</tr>
<tr>
<td>H</td>
<td>Relation is a hash index.</td>
</tr>
<tr>
<td>J</td>
<td>Relation is a join index.</td>
</tr>
<tr>
<td>P</td>
<td>Relation is a permanent table.</td>
</tr>
<tr>
<td>S</td>
<td>Relation is a spool file.</td>
</tr>
<tr>
<td>V</td>
<td>Relation is a volatile table.</td>
</tr>
</tbody>
</table>

#### SortInfo

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>The relation is not sorted.</td>
</tr>
<tr>
<td>T</td>
<td>The relation is sorted.</td>
</tr>
</tbody>
</table>
### Attribute
- **SortKind**: The way the relation is sorted. Only used when the value for SortInfo is T.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
## Chapter 6: Query Capture Facility

### Relation

#### SortKey

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:

```
SortKey1, SortKey2, ..., SortKeyn
```

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:

- `database_name.table_name.column_name`
- `spool_number.column_name`

Spool numbers are used in place of database.table names whenever the table information is not available.

#### GeogInfo

Describes the configuration geography of the relation.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>H</td>
<td>Relation is hash-distributed.</td>
</tr>
</tbody>
</table>

#### Cached

Describes whether the relation is cached or not.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Relation is not cached.</td>
</tr>
<tr>
<td>T</td>
<td>Relation is cached.</td>
</tr>
</tbody>
</table>

#### SyncScan

Describes whether a relation is eligible for synchronized scanning or not.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

#### Cardinality

Estimated cardinality of the relation.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidence</td>
<td>Describes the confidence level for the estimated cardinality.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</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>
<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.</td>
</tr>
<tr>
<td></td>
<td>If the text exceeds the upper limit of 20,000 characters, then it</td>
</tr>
<tr>
<td></td>
<td>overflows to the QryRelX table. See “Overflow” on page 613 and</td>
</tr>
<tr>
<td></td>
<td>“QryRelX” on page 592.</td>
</tr>
<tr>
<td>TableName</td>
<td>The non-aliased name of the relation.</td>
</tr>
<tr>
<td>PartitionInfo</td>
<td>Identifies whether a table or spool has a partitioned primary index.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</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>
<tr>
<td>Overflow</td>
<td>Specifies whether there is overflow query text stored in QryRelX or</td>
</tr>
<tr>
<td></td>
<td>not. The default value is F.</td>
</tr>
<tr>
<td>Flag</td>
<td>Description</td>
</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>
</tbody>
</table>
### Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Identifies whether Relation stores the complete relation DDL or a truncated version.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<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 SQL Data Manipulation Language). 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 592).</td>
</tr>
</tbody>
</table>

| Version          | 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. |

| SpoolCompressedAllowed | Identifies whether target spool files can contain compressed columns or not. |

<table>
<thead>
<tr>
<th>Flag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Spool cannot contain compressed columns.</td>
</tr>
<tr>
<td>T</td>
<td>Spool can contain compressed columns.</td>
</tr>
</tbody>
</table>

| SpoolSize | The size of the spool file for this relation in bytes. If there is no spool for the request, the column is null. |
**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 request defines the `SeqNumber` table:

```sql
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 request:

```sql
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIndex</td>
<td>An artificial column used as the NUPI for this table.</td>
</tr>
<tr>
<td>WorkLoadID</td>
<td>Unique identifier for the workload.</td>
</tr>
<tr>
<td>MachConfigID</td>
<td>Unique identifier for the configuration of a particular hardware configuration stored in QCD.</td>
</tr>
<tr>
<td>QueryID</td>
<td>Unique identifier for the query.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Definition</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UDB_KEY</td>
<td>Unique identifier on MachConfigID for the user or database that performed the query.</td>
</tr>
<tr>
<td>StepID</td>
<td>Unique identifier for a particular AMP step in the query.</td>
</tr>
<tr>
<td>RelationKey</td>
<td>Unique identifier for a table, derived table, or spool file used in the query.</td>
</tr>
<tr>
<td>PredicateID</td>
<td>Unique identifier for the predicate used in the query.</td>
</tr>
<tr>
<td>RecommendationID</td>
<td>Unique identifier for a particular set of index recommendations.</td>
</tr>
<tr>
<td>QCDVersion</td>
<td>Describes the version of QCD currently running.</td>
</tr>
<tr>
<td>DemographicsID</td>
<td>Identifies whether the demographics were captured on the current system or imported from another system. Initially set to 2.</td>
</tr>
<tr>
<td>StatisticsID</td>
<td>Identifies whether the statistics were captured by a COLLECT STATISTICS (QCD Form) or INSERT EXPLAIN request. Initially set to 2.</td>
</tr>
</tbody>
</table>
StatsRecs

Function

Each row set contains a COLLECT STATISTICS request and related information for collecting the recommended statistics generated by a DUMP EXPLAIN or INSERT EXPLAIN request specified with a CHECK STATISTICS clause.

Table Definition

The following CREATE TABLE request defines the StatsRecs table:

```sql
CREATE SET TABLE StatsRecs (
    QueryID   INTEGER,
    StatsID   INTEGER,
    DatabaseName VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC,
    TableName VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC,
    FieldID   INTEGER,
    FieldName VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC,
    Level     INTEGER,
    StatsDDL  VARCHAR(2500) CHARACTER SET LATIN NOT CASESPECIFIC,
PRIMARY INDEX (QueryID, StatsID);
```

Attribute Definitions

The following table defines the StatsRecs table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueryID</td>
<td>• Unique identifier for the query to which the recommendation applies. \</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table. \</td>
</tr>
<tr>
<td>StatsID</td>
<td>• Uniquely identifies a statistics collection recommendation for QueryID.</td>
</tr>
<tr>
<td></td>
<td>There are multiple StatsID values corresponding to a single QueryID for multicoloum statistics recommendations. \</td>
</tr>
<tr>
<td></td>
<td>In other words, a set of rows having the same value in StatsId represents all the columns for a given recommendation. \</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table. \</td>
</tr>
<tr>
<td>DatabaseName</td>
<td>Name of the containing database for TableName. \</td>
</tr>
<tr>
<td>TableName</td>
<td>Name of the table in which FieldName is defined. \</td>
</tr>
<tr>
<td>FieldID</td>
<td>Unique identifier for FieldName within TableName. \</td>
</tr>
<tr>
<td>FieldName</td>
<td>Name of the column in the statistics recommendation. \</td>
</tr>
</tbody>
</table>
### Level

A representation of the confidence the Optimizer has in the usefulness of the statistics recommendations it has generated for this query-column set combination.

The level is determined by a number of factors, including the following:

- The number of columns in the recommendation.
- Whether the recommendations are for collecting single column statistics or multicolumn statistics.

This measure is designed to help you prioritize which statistics you want to collect, particularly in situations where you cannot afford to collect statistics on a long list of multicolumn recommendations or for different combinations of multicolumn recommendations.

You can also aggregate levels to rank different recommendations.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1     | The primary recommendation, also referred to as a recommendation made with High Confidence.  
This recommendation is more likely to help the Optimizer to generate the least costly plan possible than any others, and you should always implement them unless you have very good reasons not to.  
Recommendations to collect single-column and multicolumn statistics with all the columns are considered to be primary. |
| 2     | An optional or alternative recommendation for the multicolumn statistics, also referred to as a recommendation made with Low Confidence.  
Optional recommendations provide an alternative to collecting statistics on multiple nonindex columns. These recommendations can be useful when their columns are infrequently specified together as an equality condition within the given workload or when your site determines that it would be too costly to collect statistics on several different combinations of multicolumn recommendations, but unlike the case for Level 1 recommendations, you should not feel compelled to implement them.  
Optional recommendations are provided based on information collected from usable indexes and usable columns from the table described by the `TableName` column. |

### StatsDDL

The DDL text of the `COLLECT STATISTICS` statement used to populate this row set of the table.

For multicolumn statistics, the text of the DDL statement is stored in the row having the lowest `FieldID` value. In this case, the content of this column for the other rows is null.
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 request defines the *TableStatistics* table:

```sql
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 CHARACTER(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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MachineName</td>
<td>The name of the system to which the table belongs.</td>
</tr>
<tr>
<td>TableName</td>
<td>Name of the table for which the row defines statistics.</td>
</tr>
<tr>
<td>DatabaseName</td>
<td>Name of the containing database or user for <em>TableName</em>.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Statistics are for a PARTITION column.</td>
</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>
</tbody>
</table>

| IndexID | The unique identifier for the index to which the statistics represented by this row pertain. |

<table>
<thead>
<tr>
<th>IF IndexType has this value ...</th>
<th>THEN IndexID is ...</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StatisticsInfo</th>
<th>Contains the column or index statistics.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataType</td>
<td>Indicates the data type of the column. The value is the same that is returned in the datatype column 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>
</tbody>
</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 column 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 request defines the User_Database table:

```sql
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDB_Key</td>
<td>• Unique identifier for the user or database.</td>
</tr>
<tr>
<td></td>
<td>• UPI for the table.</td>
</tr>
<tr>
<td>UDB_ID</td>
<td>Unique identifier for the user or database on the system on which the captured query was performed.</td>
</tr>
<tr>
<td>MachineName</td>
<td>Name of the production system on which the database identified by UDB_ID exists.</td>
</tr>
<tr>
<td>UDB_Name</td>
<td>Name of the user or database identified by UDB_ID.</td>
</tr>
</tbody>
</table>
**UserRemarks**

**Function**

Stores user comments about captured plans written using either the Teradata System Emulation Tool or Visual EXPLAIN client utilities.

**Table Definition**

The following CREATE TABLE request defines the *UserRemarks* table:

```sql
CREATE TABLE UserRemarks(
    QueryID INTEGER,
    StepID INTEGER,
    WorkloadID INTEGER,
    RowType CHARACTER(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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueryID</td>
<td>The ID of the query about which remarks are being saved. Partial NUPI for</td>
</tr>
<tr>
<td></td>
<td>the table.</td>
</tr>
<tr>
<td>StepID</td>
<td>The ID of the step about which remarks are being saved.</td>
</tr>
<tr>
<td>WorkloadID</td>
<td>The ID of the workload to which the query belongs.</td>
</tr>
<tr>
<td>RowType</td>
<td>Specifies which utility collected the remarks and whether they were saved</td>
</tr>
<tr>
<td></td>
<td>or generated.</td>
</tr>
<tr>
<td>SeqNumber</td>
<td>A system-generated number (1 for the first row, 2 for second, and so on)</td>
</tr>
<tr>
<td></td>
<td>to indicate the sequential order of remarks.</td>
</tr>
</tbody>
</table>

**Code**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Comments generated automatically by the Teradata System Emulation Tool utility during import.</td>
</tr>
<tr>
<td>V</td>
<td>Comments saved by the Visual EXPLAIN utility.</td>
</tr>
</tbody>
</table>
Related Topics

See the following manuals for more information about the Teradata System Emulation Tool and Teradata Visual Explain utilities.

- Teradata System Emulation Tool User Guide
- Teradata Visual Explain User Guide
ViewTable

Function

Captures the DDL for any views used in a query.

Table Definition

The following CREATE TABLE request defines ViewTable:

```sql
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueryID</td>
<td>• Unique identifier for the query.</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table.</td>
</tr>
<tr>
<td>ViewName</td>
<td>• Name of the view.</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table.</td>
</tr>
<tr>
<td>DBName</td>
<td>• Name of the database.</td>
</tr>
<tr>
<td></td>
<td>• Partial NUPI for the table.</td>
</tr>
</tbody>
</table>
## ViewTable

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViewText</td>
<td>Stores the DDL view text.</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.</td>
</tr>
</tbody>
</table>

### IF ViewText is ... THEN SeqNumber is set to ...

<table>
<thead>
<tr>
<th>Condition</th>
<th>SeqNumber Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 30,000 characters</td>
<td>1</td>
</tr>
<tr>
<td>&gt; 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>
</tbody>
</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.

### Flag

<table>
<thead>
<tr>
<th>Flag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>ViewText DDL is truncated.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>See SQL Data Manipulation Language.</td>
</tr>
<tr>
<td>T</td>
<td>Full ViewText DDL is stored.</td>
</tr>
<tr>
<td></td>
<td>Overflow ViewText DDL is stored in additional rows in ViewTable. Individual overflow rows are identified by their respective SeqNumber.</td>
</tr>
</tbody>
</table>
Workload

Function

Workload table has one entry for each workload defined in the database.

Table Definition

The following CREATE TABLE request 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkloadID</td>
<td>• Internally generated unique (within the database) identifier for the</td>
</tr>
<tr>
<td></td>
<td>workload.</td>
</tr>
<tr>
<td></td>
<td>• USI for the table.</td>
</tr>
<tr>
<td>WorkloadName</td>
<td>• User-specified unique workload name.</td>
</tr>
<tr>
<td></td>
<td>• UPI for the table.</td>
</tr>
<tr>
<td>CreatedTimeStamp</td>
<td>Timestamp when the workload was created.</td>
</tr>
<tr>
<td>LastModified</td>
<td>Timestamp when the workload was last modified.</td>
</tr>
<tr>
<td>CreatorName</td>
<td>Name of the user that created the workload.</td>
</tr>
</tbody>
</table>
Chapter 6: Query Capture Facility

WorkloadQueries

Function

Captures the queries that belong to the workload named by WorkloadID.

Table Definition

The following CREATE TABLE request defines the WorkloadQueries table:

```sql
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
</table>
| WorkloadID | • The workload ID that identifies the data in the row.  
|           | • NUPI for the table.  
|           | • One half of the USI column set for the table.  |
| QueryID   | • Unique qualifier for a query. Used to map it to the workload.  
|           | • One half of the USI column set 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 request defines the WorkloadStatus table:

```sql
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkloadID</td>
<td>• Unique identifier for the workload.</td>
</tr>
<tr>
<td></td>
<td>• Partial UPI for the table.</td>
</tr>
<tr>
<td>RecommendationID</td>
<td>Unique identifier for the set of indexes recommended for this workload</td>
</tr>
<tr>
<td></td>
<td>after an index analysis.</td>
</tr>
<tr>
<td>IndexNameTag</td>
<td>• Name of the index recommendation as specified in the INITIATE INDEX</td>
</tr>
<tr>
<td></td>
<td>ANALYSIS statement (see SQL Data Definition Language).</td>
</tr>
<tr>
<td></td>
<td>• Partial UPI for the table.</td>
</tr>
<tr>
<td>ValidatedSystem</td>
<td>Name of the system on which index recommendations are last validated by</td>
</tr>
<tr>
<td></td>
<td>the Teradata Index Wizard.</td>
</tr>
<tr>
<td>ValidatedTimeStamp</td>
<td>Timestamp of the index validation.</td>
</tr>
</tbody>
</table>
### Attribute Definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValidatedQCD</td>
<td>Name of the QCD on which the validation is performed by the Teradata Index Wizard.</td>
</tr>
<tr>
<td>RecosAppliedSystem</td>
<td>Name of the system on which index recommendations are last created by the Teradata Index Wizard.</td>
</tr>
<tr>
<td>Remarks</td>
<td>Free form textual remarks about the workload.</td>
</tr>
</tbody>
</table>
XMLQCD

**Function**

Captures the XML QCD output from an INSERT EXPLAIN request submitted with the IN XML option.

**Table Definition**

The following CREATE TABLE request defines the XMLQCD table:

```sql
CREATE MULTISET TABLE qcd.XMLQCD, NO FALLBACK, NO BEFORE JOURNAL,
NO AFTER JOURNAL, CHECKSUM = DEFAULT (ID INTEGER NOT NULL,
Kind CHAR(1) NOT NULL CHARACTER SET LATIN NOT CASESPECIFIC,
Seq INTEGER NOT NULL,
Length INTEGER NOT NULL,
Text VARCHAR(31000) NOT NULL CHARACTER SET UNICODE)
UNIQUE PRIMARY INDEX (ID, Kind, Seq);
```

**Attribute Definitions**

The following table defines the XMLQCD table attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
</table>
| ID        | • Unique identifier for the set of rows associated with this XML document.  
            • Partial UPI for the table. |
| Kind      | • A flag that describes the kind of operation characterized by this row.  
            • Partial UPI for the table. |

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>The system captures the value for ID from <code>qcd.SeqNumber.QueryID</code> (see “SeqNumber” on page 615) whenever this query capture operation is performed.</td>
</tr>
</tbody>
</table>

| Seq | • Unique identifier for the query that specifies the order of the generated row.  
    The first row generated from a given XML document is assigned the sequence number 1, the second row is assigned a sequence number 2, and so on.  
    • Partial UPI for the table. |
### Attribute | Definition
--- | ---
**Length** | The number of characters that remain in the XML document beginning with this row. The equation for determining the approximate value for Length is as follows:

\[
\text{Length} = (\text{Total number of characters} - (\text{Seq} - 1) \times 31000)
\]

For example:

<table>
<thead>
<tr>
<th>IF Seq is ...</th>
<th>THEN Length is the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>total number of characters.</td>
</tr>
<tr>
<td>2</td>
<td>total number of characters minus 31000.</td>
</tr>
<tr>
<td>3</td>
<td>total number of characters minus 62000.</td>
</tr>
</tbody>
</table>

and so on.

**Text** | The Seq\textsuperscript{th} slice of the XML document. |
CHAPTER 7 Database Foundations for the Teradata Index and Statistics Wizards

This chapter describes the platform-based architecture for the Teradata Index Wizard and Teradata Statistics Wizard utilities.

The Teradata Index Wizard helps you to reengineer existing databases by providing a method to propose optimal sets of partitioned primary indexes, secondary indexes, and single-table join indexes as well as PPI partitioning expressions to support particular SQL query workloads. This process permits you to analyze the potential performance enhancing effects of various join and secondary and single-table join indexes and PPI partitioning expressions 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 join and secondary and single-table join 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 Database Design for more information on ATM).

In general, the processes described in this chapter apply equally well to partition analysis, which uses the same software systems to generate recommended partitioning expressions, which it also stores in your QCD (see Chapter 6: “Query Capture Facility” for a description of the relevant partitioning expression recommendation tables).

The chapter also explains some of the methods the Teradata Statistics Wizard uses to derive its recommendations.


Detailed information about related tools and procedures is found in the following sources:

- Chapter 6: “Query Capture Facility”
- Chapter 8: “Target Level Emulation”
- “COLLECT DEMOGRAPHICS” in SQL Data Manipulation Language
- “COLLECT STATISTICS (QCD Form)” in SQL Data Manipulation Language
- “INITIATE INDEX ANALYSIS” in SQL Data Manipulation Language
- “INITIATE PARTITION ANALYSIS” in SQL Data Manipulation Language
- “DUMP EXPLAIN” in SQL Data Manipulation Language

1. The Teradata Index Wizard recommends only partitioning expressions based on RANGE_N functions for partitioned primary indexes.

2. The Teradata Index Wizard does not make recommendations for sparse or multitable join indexes, or for hash indexes, nor does it recommend dropping existing PPI partitioning expressions.
- “INSERT EXPLAIN” in *SQL Data Manipulation Language*
- “RESTART INDEX ANALYSIS” in *SQL Data Manipulation Language*
- *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 and single-table join index and PPI partitioning expression assignment. The analysis can produce recommendations to delete, as well as to add, indexes to the existing physical design. No drop recommendations are made for existing partitioning expressions.

The Index Wizard works in two modes: Wizard mode and What If? mode.

In Wizard mode, the Index Wizard provides a menu-driven interface to guide you through the steps of identifying a workload, and then identifying and implementing a set of indexes that should help to improve the performance of that workload. See “General Procedure for Using the Teradata Index Wizard to Optimize Index and Partitioning Expression Selection” on page 637 for details of the processes that wizard mode provides.

In What If? mode, the Index Wizard allows you to provide a set of recommended indexes or partitioning expressions. The Index Wizard then simulates the set of indexes you submit and produces a report of the estimated overall improvement to the workload if the proposed indexes, or partitioning expressions, or both, had been implemented.

The Index Wizard also provides query-by-query details indicating the improvement for each individual query in a defined workload, and whether or not your simulated indexes were used by the Optimizer to develop the new query plan. The limit on query text size the Index Wizard can process is 1 megabyte. Overflow text is stored in the table `qcd.QryRelX` (see “QryRelX” on page 592), where `qcd` represents the name you have assigned to the containing QCD database.

See *Teradata Index Wizard User Guide* for details.

The logic for evaluating and selecting indexes and PPI partitioning expressions is located in the Parsing Engine. The architecture for the Index Wizard consists of the SQL Optimizer (see Chapter 2: “Query Rewrite and Optimization”) and additional code to generate and simulate candidate indexes. The approach used for index selection is let the Optimizer choose the indexes it “wants” to use, meaning those indexes that it first searches for when presented with an SQL request to optimize. The index analysis process calls the Optimizer repeatedly with different sets of simulated candidate indexes. The analysis takes promising columns from the Optimizer's analysis of WHERE conditions done by the Optimizer and uses them as the initial input to the search phase of the process.

---

3. The Teradata Index Wizard recommends only partitioning expressions based on RANGE_N functions for partitioned primary indexes.
The analysis then uses a combinatorial search engine to explore the solution space, which consists of various combinations of the promising columns as well as other design factors:

- In the case of secondary indexes, the design factors include different combinations of columns that make up a composite index, the use of extra columns to cover queries, and different types of secondary indexes such as value-ordered indexes in addition to the more standard hashed indexes.
- In the case of join indexes, the primary design factors include different combinations of columns that make up the primary index definition for the join index, the use of additional columns to cover queries, and the use of the ROWID to define noncovering join indexes.
- In the case of partitioned primary indexes, the primary design factor explored is the optimal level of granularity for each candidate partitioning column.

The Index Wizard only recommends single-level PPIs with partitioning expressions based on RANGE_N functions, but you can infer recommendations for multilevel PPIs from any report that recommends multiple partitioning expressions, then experiment with combining some or all of those recommendations to create a multilevel PPI.

The conclusion to draw from this is that the Index Wizard is not merely an automated version of physical database design rules that were previously performed manually. The selection process considers a large number of candidates, and calls the Optimizer to evaluate and rank each candidate index by its effect on the estimated query cost. This approach ensures consistency with the choice of indexes made by the Optimizer while avoiding duplication of the Optimizer cost model within the design tool.

The major architectural components of the Index Wizard are the following four modules:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query analyzer</td>
<td>Forms a list of interesting, or indexable, columns by calling the predicate analysis code of the SQL Optimizer.</td>
</tr>
<tr>
<td></td>
<td>Examines the projection list and other parts of a query for covering information.</td>
</tr>
<tr>
<td>Index search engine</td>
<td>Combinatorial search algorithm capable of performing a random search on a large solution space of candidate indexes and partitioning expressions.</td>
</tr>
<tr>
<td>Index simulator</td>
<td>Simulates indexes during the query optimization phase of index and partitioning expression analysis.</td>
</tr>
<tr>
<td>Query optimizer</td>
<td>The system SQL Optimizer. Index and partitioning expression analysis uses its estimated plan costs to evaluate candidate indexes.</td>
</tr>
</tbody>
</table>

Note that the Partition Analysis system uses the same architectural components as the Index Analysis system.

See Teradata Index Wizard User Guide for details about the various Teradata Index Wizard processes.
General Procedure for Using the Teradata Index Wizard to Optimize Index and Partitioning Expression Selection

The following table outlines the general procedure followed in performing index and partitioning expression analysis in wizard mode using the Teradata Index Wizard:

<table>
<thead>
<tr>
<th>Step</th>
<th>Activity Name</th>
<th>Activity Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Identify a workload.</td>
<td>Identify a set of SQL requests that constitutes a workload for consideration for analysis.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The following repositories are potential sources for identifying workload components:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Database Query Log (see SQL Data Definition Language and Database Administration).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Query Capture Database (see Chapter 6: “Query Capture Facility”).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See “Workload Identification” on page 641 for details.</td>
</tr>
<tr>
<td>2</td>
<td>Define the workload.</td>
<td>Perform the AddWorkload macro to define a new workload in a query capture database.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See “Workload Definition” on page 643 for details.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Also see “INSERT EXPLAIN” in SQL Data Manipulation Language.</td>
</tr>
<tr>
<td>3</td>
<td>Analyze indexes.</td>
<td>Perform an index analysis on the defined workload to produce a set of index recommendations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See “Index and Partitioning Expression Analysis” on page 646 for details.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Also see “INITIATE INDEX ANALYSIS” and “INITIATE PARTITION ANALYSIS” in SQL Data Manipulation Language.</td>
</tr>
<tr>
<td>4</td>
<td>Report the analysis.</td>
<td>View summary and detailed information about the workload, the existing physical design, the changes proposed by the Index Wizard, and the costs versus benefits of the index and partition recommendations relative to the workload.</td>
</tr>
<tr>
<td>5</td>
<td>Validate the indexes.</td>
<td>Validate the recommended indexes on your production system or on a test system with the aid of the Teradata System Emulation Tool (see Teradata System Emulation Tool User Guide for details).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is an optional step in the process.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See “Index and Partitioning Expression Validation” on page 652 and “DIAGNOSTIC “Validate Index”” in SQL Data Manipulation Language for details.</td>
</tr>
</tbody>
</table>
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 8: “Target Level Emulation”) to emulate production environments when performing index analyses on a test system.

- The Teradata Index Wizard produces recommendations for partitioned primary indexes, secondary indexes, and single-table join indexes only.

  a  A workload is identified. See “Workload Identification” on page 641.

  b  Query plans that characterize the workload are collected in the QCD. This is referred to as defining the workload. See “Workload Definition” on page 643.

4. The Teradata Index Wizard recommends only partitioning expressions based on RANGE_N functions for partitioned primary indexes.
Index analysis and validation are performed on the QCD data. See “Index and Partitioning Expression Analysis” on page 646 and “Index and Partitioning Expression Validation” on page 652.

The Teradata Index Wizard begins the analysis by invoking either the SQL INITIATE INDEX ANALYSIS statement or the INITIATE PARTITION ANALYSIS statement (see “INITIATE INDEX ANALYSIS” and “INITIATE PARTITION ANALYSIS” in SQL Data Manipulation Language).

INITIATE INDEX ANALYSIS selects candidate indexes for the submitted workload and submits each of them with the workload to the Optimizer. Note that you can restart timed out index analyses using appropriate CHECKPOINT and TIMELIMIT option specifications for the INITIATE INDEX ANALYSIS/RESTART INDEX ANALYSIS statements (see “INITIATE INDEX ANALYSIS” and “RESTART INDEX ANALYSIS” in SQL Data Manipulation Language for details) or Teradata Index Wizard (see Teradata Index Wizard User Guide).

<table>
<thead>
<tr>
<th>IF the optimizer generates a query plan that ...</th>
<th>THEN the Teradata Index Wizard ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>does not use the candidate index</td>
<td>drops that index from consideration as a recommendation.</td>
</tr>
<tr>
<td>uses the candidate index</td>
<td>keeps that index for recommendation.</td>
</tr>
</tbody>
</table>

Alternatively, INITIATE PARTITION ANALYSIS selects candidate partitioning expressions for tables in the submitted workload and submits each of them with the workload to the Optimizer.

See “How Index and Partitioning Expression Analyses Are Done” on page 646 for details of this process.

End of subprocess.

End of process.
DBS Control Flags for Tuning the Workload Cache Limits for Index and Partition Analysis

Index Wizard processing in Teradata Database allocates workload cache for the purpose of caching information retrieved from the QCD and other temporary working structures that are needed during the Analysis and Validation phases of recommending indexes and partitioning expressions for various SQL workloads.

Two DBS Control performance group flags control the respective workload cache sizes for the Analysis and Validation phases:

- IAMaxWorkloadCache
- IVMaxWorkloadCache

The following table provides some of the details for these flags. See Utilities for further information.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Purpose</th>
<th>Valid Range (megabytes)</th>
<th>Default (megabytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAMaxWorkloadCache</td>
<td>Defines the maximum size of the Index Wizard workload cache for Analysis operations.</td>
<td>32 - 187</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>This parameter is applicable to the following SQL statements only:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- INITIATE INDEX ANALYSIS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- INITIATE PARTITION ANALYSIS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “INITIATE INDEX ANALYSIS” and “INITIATE PARTITION ANALYSIS” in SQL Data Manipulation Language for details about these statements.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVMaxWorkloadCache</td>
<td>Defines the maximum size of the Index Wizard workload cache for Validation operations.</td>
<td>1 - 32</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>This parameter is applicable to all SQL statements issued within a session when DIAGNOSTIC &quot;Validate Index&quot; has been enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “DIAGNOSTIC “Validate Index”” in SQL Data Definition Language for details about this statement.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Workload Identification

Introduction

The objective of workload identification is to isolate the set of requests, called a workload, you want to analyze for the possibility of enhancing their performance by redefining the secondary and single-table join 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.
  
  Your queries are taking longer to complete than you think they should, but you are unable to identify precisely what the cause of the slow responses is.

- Sift through the following data stores in search of candidate queries for performance enhancement:
  
  - The Database Query Log using the Teradata Index Wizard (see Teradata Index Wizard User Guide for details).
  
  - The Query Capture Database using tools like Visual EXPLAIN or Teradata System Emulation Tool.

After determining the set of queries to be analyzed, you can identify the workload using the Teradata Index Wizard tool or the Teradata System Emulation Tool (see Teradata System Emulation Tool User Guide).

See Teradata Index Wizard User Guide for details about this process.
Workload Identification Process

The process of identifying workloads is indicated by the following graphic:

<table>
<thead>
<tr>
<th>Key</th>
<th>Input Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>View query plan</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Feed SQL text</td>
<td>You can directly type the text for the queries that are to make up a workload or you can load them from the Teradata platform.</td>
<td></td>
</tr>
<tr>
<td>Query DBQL</td>
<td>You can fetch logged Database Query Log SQL text for index analysis by means of the Index Wizard.</td>
<td></td>
</tr>
<tr>
<td>Get target query</td>
<td>You can export a set of queries for analysis using Teradata System Emulation Tool.</td>
<td></td>
</tr>
</tbody>
</table>
Workload Definition

Introduction

The objective of workload definition is to capture the set of query plans associated with the requests 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 requests 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 SQL Data Manipulation Language).

You are limited to the following SQL statements when you define a workload using an INSERT EXPLAIN request:

- DELETE
- INSERT
- SELECT
- UPDATE

You cannot include any load operations, such as a MultiLoad job, in your workload specifications.

See Teradata Index Wizard User Guide for details about this process.

Workloads Defined For the Index Wizard Are Not the Same As Those Defined With Teradata Workload Analyzer or Teradata Dynamic Workload Analyzer

The workloads defined by the Index Wizard have no functional relationship to the similarly named workloads defined by the Teradata Workload Analyzer and the Teradata Dynamic Workload Manager.


Workload Definition Macros

Teradata Database supplies 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:

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insert workload</td>
<td>The workload is defined in the specified QCD on the production system.</td>
</tr>
<tr>
<td></td>
<td>Export workload</td>
<td>Once it has been created, the workload can be exported from the production system to the test system using the Teradata System Emulation Tool.</td>
</tr>
<tr>
<td></td>
<td>Import workload</td>
<td>In the second stage of this process, the workload is imported to the test system from the production system using either Teradata System Emulation Tool or the Teradata Index Wizard.</td>
</tr>
</tbody>
</table>

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 8: “Target Level Emulation”).
Mapping ATM Activities to Defining a Workload

Many phases of the Activity Transaction Modeling process (see Database Design) map well to the reengineering phases of workload definition. The following table indicates those mappings:

<table>
<thead>
<tr>
<th>ATM Activity</th>
<th>Workload Definition Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify and define attribute domains and constraints for physical columns.</td>
<td>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.</td>
</tr>
<tr>
<td>Identify and model database applications.</td>
<td>Not applicable.</td>
</tr>
<tr>
<td>Model application processing activities to include their transactions and run frequencies.</td>
<td>Submit the SQL requests as transactions and run frequencies using the INSERT EXPLAIN statement or the Index Wizard. 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.</td>
</tr>
<tr>
<td>Model transactions to identify the tables used and the columns required for value or join access and estimating cardinalities</td>
<td>The workload analysis includes this activity. The system assimilates the information for the set of SQL requests submitted as a workload, and then stores it in a Query Capture Database.</td>
</tr>
<tr>
<td>Summarize value and join access information across all transactions.</td>
<td>The data stored in QCD as part of workload analysis is used to summarize the query information.</td>
</tr>
<tr>
<td>Compile a preliminary set of data demographics by estimating table cardinalities and value distributions and assigning change ratings.</td>
<td>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.</td>
</tr>
</tbody>
</table>
Introduction

The objective of Index Analysis (including Partitioning Expression Analysis) is to recommend a set of new indexes and PPI partitioning expressions using CREATE INDEX, CREATE JOIN INDEX, or ALTER TABLE … MODIFY PRIMARY INDEX statements on the different tables referenced in the workload.

In the case of secondary and join indexes, Index Analysis also provides a set of existing indexes that it recommends be dropped using DROP INDEX requests when the analysis supports removing those indexes.

Index Analysis does not recommend dropping existing partitioning expressions, assuming that they might be important to the performance of either of the following client utilities:

- Archive/Restore
- MultiLoad insert and delete operations

Depending on the size and complexity of the defined workload, Analysis operations can be resource intensive, particularly because the repeated calls to the Optimizer with different sets of candidate indexes or partitioning expressions or both incurs heavy CPU usage. Because of this, you should generally avoid performing Index Analysis operations (meaning the INITIATE INDEX ANALYSIS and INITIATE PARTITION ANALYSIS statements) on a production system.

See Teradata Index Wizard User Guide for details about this process.

See Teradata Index Wizard User Guide for details about this process.

See Teradata Archive/Recovery Utility Reference and Teradata MultiLoad Reference for details about how to use those utilities.

How Index and Partitioning Expression Analyses Are Done

The following stages describe the process used by the Index Wizard to perform an index analysis:

1. The candidate search space and proposed index list for the workload are set null.
2. The workload candidate search space is built as follows:
   a. For each SQL request in the workload:
      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 (including recommendations that would increase update costs to the base table or index subtables), producing the resultant statement proposed index list.
3 Add the statement proposed index list to the workload candidate search space.
4 Determine if any table constraints have been violated.

<table>
<thead>
<tr>
<th>IF a table constraint violation is ...</th>
<th>THEN ...</th>
</tr>
</thead>
</table>
| found                                 | 1 Invoke the Index Search Engine to filter candidate indexes from the workload candidate search space.  
2 Build a new proposed workload 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.a.4. |
| not found                             | Go to Stage 2.a.2. |

5 Set the workload candidate search space to the workload proposed index list.

b Act on any checkpoints that have been set for the index analysis.

<table>
<thead>
<tr>
<th>IF a checkpoint is ...</th>
<th>AND the specified checkpoint frequency ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>specified</td>
<td>is hit</td>
<td>write the proposed index list for the workload to the QCD AnalysisLog table.</td>
</tr>
<tr>
<td>not specified</td>
<td></td>
<td>go to Stage 3.0.0.</td>
</tr>
</tbody>
</table>

3 Filter the proposed index list for the workload using the best indexes analyzed to this point to produce the index recommendations.
4 Write the recommendations to the QCD IndexRecommendations and PartitionRecommendations tables (see “IndexRecommendations” on page 573 and “PartitionRecommendations” on page 585).
5 Delete any remaining AnalysisLog table entries.
6 End of process.

You can specify various index analysis parameters to control the choice of candidate indexes using the various boundary options of the INITIATE INDEX ANALYSIS and INITIATE PARTITION ANALYSIS statements (see SQL Data Manipulation Language for details) or the Teradata Index Wizard (see Teradata Index Wizard User Guide). 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 adding recommended new indexes: they can also include recommendations to drop existing indexes.

Note, too, that index recommendations never include volatile tables that are accessed by the SQL requests 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 stages collapse into a single stage.
### Key

<table>
<thead>
<tr>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get target data</td>
<td>Use Teradata System Emulation Tool (see <em>Teradata System Emulation Tool User Guide</em>) to collect production system data such as environmental cost parameters and random AMP statistical samples.</td>
</tr>
<tr>
<td>Emulate the production environment</td>
<td>Use the Teradata Index Wizard client interface (see <em>Teradata Index Wizard User Guide</em>) to set up the production system data at session level on a test system so it can emulate the production environment.</td>
</tr>
<tr>
<td>Get index and partitioning recommendations from the Teradata test system.</td>
<td>Perform the index analysis on the test system. Use the INITIATE INDEX ANALYSIS or INITIATE PARTITION ANALYSIS statement (see SQL Data Manipulation Language), or Teradata Index Wizard client utility to submit the SQL request set to be analyzed. INITIATE INDEX ANALYSIS collects the generated index recommendations and saves them in the IndexRecommendations table of the specified QCD. INITIATE PARTITION ANALYSIS collects the generated PPI partitioning expression recommendations and saves them in the PartitionRecommendations table of the specified QCD. Note that index recommendations also include relevant CREATE INDEX, CREATE JOIN INDEX, and DROP INDEX, requests plus the associated COLLECT STATISTICS statements for any recommended indexes, while partition recommendations include only the relevant ALTER TABLE ... MODIFY PRIMARY INDEX SQL text for adding partitioning expressions. The Index Wizard never recommends dropping partitioning expressions.</td>
</tr>
<tr>
<td>Get index and partitioning recommendations from the Teradata production system</td>
<td>When this process is performed on a production system, this step is identical to the previous step except the analysis is done on a Teradata production system rather than a Teradata test system.</td>
</tr>
</tbody>
</table>
INITIATE INDEX ANALYSIS requests insert their final recommendations into the specified IndexRecommendations QCD table (see “IndexRecommendations” on page 573), which you can then query. The system stores the definition of a recommended index in a CHARACTER column in the form of SQL text for CREATE INDEX and DROP INDEX requests.

INITIATE PARTITION ANALYSIS statements insert their recommendations into the specified PartitionRecommendations QCD table (see “PartitionRecommendations” on page 585). The system stores the definition of a recommended partitioning expression in a CHARACTER column in the form of SQL text for a PARTITION BY clause for an ALTER TABLE … MODIFY PRIMARY INDEX request.

A separate row is stored in each of these tables for each workload query that is impacted by the recommendation.

In the case of Secondary and Join Indexes, there is a row for every query whose new cost benefits from the recommended index.

In the case of partitioning recommendations, there is a row for every query whose cost benefits or suffers from the recommended PPI. The fact that the PPI is being recommended means that it reduces the total workload cost, but it is possible that the cost of one or more individual queries within the specified workload has increased.

Such rows can be identified in the QCD table PartitionRecommendations by comparing the estimated new cost with the original cost, both of which are stored in separate columns within the relevant row.

For obvious reasons, you should pay particular attention to any such rows prior to implementing a recommended PPI partitioning expression.

To provide additional flexibility in displaying the recommendations, the system stores the individual details for a given recommendation in separate columns and rows in another set of QCD tables.

For example, the individual columns of a recommended composite secondary index are stored separately in QCD table IndexColumns (see “IndexColumns” on page 570).

Similarly, the individual range-start, range-end, and range-size values that make up a recommended RANGE_N partitioning expression are stored in QCD table RangePartExpr (see “RangePartExpr” on page 607).
In addition to SQL text definitions and query cost information, the final index recommendations from an INITIATE INDEX ANALYSIS request also include the following information in IndexRecommendations:

- The estimated time required to create a recommended secondary or join index.
- The disk space required to store a recommended secondary or join index subtable.
- The additional maintenance costs incurred by SQL update requests that modify columns in the recommended indexes.

For recommended PPI partitioning expressions, an INITIATE PARTITION ANALYSIS request stores slightly different information in PartitionRecommendations:

- The system does not record the estimated time required to add a recommended partitioning expression.
- The system does record the disk space required to store the internal partition number in the base table or index subtable rows.
Index and Partitioning Expression Validation

Introduction

You can validate the indexes and partitioning expressions recommended by the Index Wizard by comparing query response latencies with the same workload against the existing index or partitioning expression set against response latencies for the same query set and workload using the recommended index and partitioning expression sets. This is an optional, but recommended, step in the database query analysis process. It is expected that you will often validate the union of INITIATE INDEX ANALYSIS and INITIATE PARTITION ANALYSIS recommendations for a given workload, or perhaps more likely, a selected subset of those recommendations, in an integrated Validation session.

The index and partitioning expression recommendations from an Analysis operation can be further evaluated in a special test mode referred to as Validation mode. Validation mode is essentially a special session mode that permits query capture with simulated indexes or primary index partitioning. The resulting captured plan includes any cost benefits or degradations realized by having the simulated index or partitioning expression defined and emulated.

While a session is in Validation mode, you can collect additional statistics that were not available during Analysis in order to provide more accurate costing. Unlike Analysis, there is no reason not to perform the Validation phase on a production system.

The system emulates indexes by issuing appropriate CREATE INDEX and CREATE JOIN INDEX statements that record the definition in a private session context. The index definitions are not stored in the dictionary, nor are the index subtables actually populated.

Similarly, the system emulates recommended partitioning expressions by issuing appropriate ALTER TABLE … MODIFY PRIMARY INDEX requests. Note that ALTER TABLE requests of this form are not ordinarily allowed on populated tables. However, they are permitted in Validation mode as a method for specifying an emulated PPI.

In addition to evaluating recommendations from a prior Analysis, Validation mode can also be used to evaluate user-specified indexes or partitioning expressions. This capability is commonly referred to as What If? analysis, and it allows sophisticated users to experiment with their own candidate indexes and partitioning expressions. The only restriction is that any user-specified candidates tested while your session is in Validation mode must be of the same type and class recommended by Analysis. For example, you cannot validate a multitable join index in validation mode because it is not supported in Analysis.

A given Validation mode session can include multiple simulated indexes or partitioning expressions or both. Unlike Analysis, it is possible to evaluate candidate PPIs alongside other candidate index types in the same Validation operation.

See SQL Data Manipulation Language for additional information about validating indexes recommended by the Teradata Index Wizard.

See Teradata Index Wizard User Guide for details about this process.
Index and Partitioning Expression Validation Process

The process of validating recommended indexes and partitioning expressions is indicated by the following graphic. For any callout of Index in the graphic, you should read Index or Partitioning Expression, or both.

![Diagram showing the process of validating recommended indexes and partitioning expressions.](1101A460)
### Key

<table>
<thead>
<tr>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get index and partitioning recommendations</td>
<td>The Index Wizard utility retrieves the index and partitioning expression 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 <em>SQL Data Manipulation Language</em> for more information.</td>
</tr>
<tr>
<td>- Validate recommendations</td>
<td>The actions required to complete these activities depend on whether the analysis is performed on a production system or a test system.</td>
</tr>
<tr>
<td>- Import/export QCD statistics</td>
<td></td>
</tr>
<tr>
<td><strong>FOR this type of system ...</strong></td>
<td><strong>The following actions must be taken ...</strong></td>
</tr>
<tr>
<td>production</td>
<td>Rerun the entire workload after the previous activity in Index and Partitioning Expression Validation has been performed. The recommendations are made available to the Optimizer for generating the query plan.</td>
</tr>
<tr>
<td>test</td>
<td>The following actions must be taken in the following order:</td>
</tr>
<tr>
<td>1</td>
<td>Export the statistics from the previous stage of Index and Partitioning Expression Validation from the production system using Teradata System Emulation Tool.</td>
</tr>
<tr>
<td>2</td>
<td>Import the exported statistics from the production system to the test system using Teradata System Emulation Tool.</td>
</tr>
<tr>
<td>3</td>
<td>Rerun the entire workload. The recommendations are made available to the test system Optimizer for generating the query plan.</td>
</tr>
<tr>
<td>Compare the query plans with and without implementing the recommended indexes</td>
<td>Use the client Visual EXPLAIN tool (<em>see Teradata Visual Explain User Guide</em>) to compare the query plans with and without incorporating the index and partitioning expression recommendations to evaluate the extent of query processing enhancement produced by the recommendations that were made in the Analysis phase.</td>
</tr>
</tbody>
</table>
Index and Partitioning Expression Application

Introduction

At this stage of the database query analysis, you apply the indexes and partitioning expressions recommended by the Index and Partitioning Expression Analysis stage to the production database.

See Teradata Index Wizard User Guide for details about this process.

Index and Partitioning Expression Creation Process

The process of applying the recommended indexes, or partitioning expressions, or both, to the production system is indicated by the following graphic. For any callout of Index in the graphic, you should read Index or Partitioning Expression, or both.
You can choose to implement all, some, or none, of the index or partitioning recommendations from a given Analysis.

In the case of indexes, the Index Wizard client utility has a menu option that automatically issues an SQL request containing the corresponding CREATE INDEX or CREATE JOIN INDEX text for you.

In the case of PPIs, a recommended table partitioning requires a more complex set of SQL requests to recreate a populated table. The Index Wizard client utility provides a set of SQL requests in the form of a script to demonstrate one possible method of converting an existing table to a PPI table.

You are expected to examine and modify the script for the specific needs of your particular system. For example, you must verify that there is enough available disk space to temporarily hold two copies of the table, and you must also determine what privileges must be regranted and which existing indexes must be recreated.

You can indirectly execute the script by copying the displayed contents to the Execute SQL window that is accessible from the Tools menu of the Index Wizard.

The script creates a new permanent table, partitions it as desired, executes an appropriate INSERT ... SELECT request to insert the rows from the source NPPI table into it, drops the source table, and then renames the new table with the name of the old source table.

To reduce the possibility of naming collisions, the new table (before renaming) exists temporarily, before it is renamed, with a name of PPI_base_table_name where base_table_name is truncated if necessary.

See “An Extended Example” on page 658 for an example of using this method.
Effects of Request Cache Peeking on INSERT EXPLAIN and DUMP EXPLAIN Outcomes

When a data parcel is submitted with an INSERT EXPLAIN or DUMP EXPLAIN request, the plan might be generated with peeked USING and CURRENT_DATE or DATE values, or both (see “Peeking at the Request Cache” on page 36). If any of these values are peeked, then the query plan shows them.

If no data parcel is submitted with an INSERT EXPLAIN or DUMP EXPLAIN request, the resulting plan is generated without peeking at USING, CURRENT_DATE, or DATE values, so it is a generic plan by definition. Note that the Visual Explain, Teradata System Emulation Tool, Teradata Index Wizard, and Teradata Statistics Wizard tools do not accept USING data as input while capturing query plans using INSERT EXPLAIN or DUMP EXPLAIN requests.

The Teradata Index and Statistics wizards internally generate plans for workload queries in order to estimate workload costs, which are then used to determine optimal index recommendations and statistics that should be collected. When queries in the workloads specify USING request modifiers, the plan is generated without peeking at USING, CURRENT_DATE, or DATE values. Because of these factors, Request Cache peeking has no impact on the resulting index or statistics recommendations. Given that workload analyses should be independent of USING values, this behavior is correct.
An Extended Example

Introduction

This section provides a start-to-finish example of the Index Wizard steps described in the previous topics. Although sophisticated users can manually code and execute the SQL requests shown in this example, the majority of users are likely to use the Index Wizard client interface.\(^5\)

This example assumes that you have defined the following tables:

```sql
CREATE TABLE sales_history (  
    product_code  CHARACTER(8),  
    store_number  INTEGER,  
    transaction_date DATE,  
    quantity_sold  INTEGER,  
    other_columns  CHARACTER(50))  
PRIMARY INDEX (product_code, store_number, transaction_date);

CREATE TABLE products (  
    product_code  CHARACTER(8),  
    description   VARCHAR(50),  
    product_category  CHARACTER(10),  
    price         DECIMAL(10,2),  
    quantity_avail  INTEGER)  
PRIMARY INDEX (product_code);
```

Recall that this is a multistage process consisting of the following high level steps that must be performed in the order indicated:

1. Capture the query information
2. Create the workload.
3. Associate the workload requests with the workload identifier.
4. Analyze the workload.
   - This is a two-step process. The two steps can be performed in any order.
     - Do an index analysis on the workload.
     - Do a partition analysis on the workload.
5. View the recommendations made by the index and partition analyses.
6. View the estimated improvement in query costs that can be accrued by implementing the index and partition recommendations.
7. Validate the recommendations.
   - This is a six-stage process consisting of the following stages:
     a. Turn on validation mode for the session.
     b. Simulate creating the recommended indexes and primary index partitioning.

5. Note that if you were doing this example using the Index Wizard utility, it would call the identical SQL requests that are shown here. The only difference is the GUI interface presented by the client utility instead of the SQL Assistant or BTEQ interface you would use to submit those SQL requests.
c Simulate collecting the recommended statistics on the recommended indexes and partitioning columns.
d Capture the new query plans for the queries in the workload.
e Turn off validation mode for the session.
f Analyze the steps in the captured plans for cost reductions and increased partition elimination.

8 Execute the validated recommendations.
9 End of process.

Apart from steps 8 and 9, you should perform this analysis on a test system, not your production system.

**Stage 1: Capture the Query Information**

In the first stage of the process, you use INSERT EXPLAIN requests to capture workload information into your QCD database.

For this example, you first capture data for the query that gives year-to-date sales quantities for each product. Because you specify the WITH STATISTICS option, the system automatically collects statistics on the `transaction_date` column, which is one of the three columns making up the composite NUPI for table `SalesHistory`.

```sql
INSERT EXPLAIN WITH STATISTICS INTO qcd AS q1
    SELECT product_code, SUM(quantity_sold)
    FROM SalesHistory
    WHERE transaction_date >= DATE '2006-01-01'
    GROUP BY product_code;
```

Next, you capture data for the query that returns various summarized quantities for a particular value of the `product_category` column for table `products`. Again, because you specify the WITH STATISTICS option, the system automatically collects statistics on the `product_category` column, which is a non-index column for the table.

```sql
INSERT EXPLAIN WITH STATISTICS INTO qcd AS q2
    SELECT SUM(quantity_avail), AVG(quantity_avail),
           MIN(quantity_avail)
    FROM Products
    WHERE product_category = 'Laptop';
```

**Stage 2: Create the New Workload**

In the second stage of the process, you use the Index Wizard-supplied macro `QCD.addWorkload` to create a new workload for your queries.

```sql
EXEC qcd.addWorkload('MyWorkload');
```
Stage 3: Associate the Workload Queries With the Workload Identifier

In the third stage of the process, you associate the two queries in the workload with the newly created workload identifier *MyWorkload*.

```sql
INSERT INTO qcd.workloadqueries
SELECT workload.workloadid, query.queryid, 1
FROM qcd.workload, qcd.query
WHERE workload.workloadname = 'MyWorkload'
AND query.queryname IN ( 'q1', 'q2');
```

Stage 4: Analyze the Workload

The fourth stage of the process consists of two independent stages: index analysis and partition analysis.

Arbitrarily, you begin by performing an index analysis on the workload looking for NUSIs, as indicated by the WITH INDEX TYPE option specification of index type 4, which represents NUSIs.

```sql
INITIATE INDEX ANALYSIS
FOR MyWorkload IN qcd AS MyIndexes
WITH INDEX TYPE 4;
```

Equally arbitrarily, you then perform a partition analysis on the workload.

```sql
INITIATE PARTITION ANALYSIS
FOR MyWorkload IN qcd AS MyPPIs;
```

You could just as easily have done the partition analysis first and the index analysis second.

Stage 5: View the Recommendations Made By the Analysis

In the fifth stage of the process, you first view the NUSIs recommended by the index analysis:

```sql
SELECT queryname, indexddl
FROM qcd.indexrecommendations AS ir,
     qcd.workload AS wl,
     qcd.query AS q
WHERE ir.workloadid = wl.workloadid
AND ir.queryid = q.queryid
AND workloadname = 'MyWorkload'
AND ir.indexnametag = 'MyIndexes';
```

Assume the following output from this query:

```
queryname      indexddl
---------------  -------
Q2              CREATE INDEX (product_category) ON Products
```
Next you view the partitioning recommended by the partition analysis:

```sql
SELECT queryname, expressiontext
FROM qcd.partitionrecommendations AS pr,
     qcd.workload AS wl,
     qcd.query AS q
WHERE pr.workloadid = wl.workloadid
AND pr.queryid = q.queryid
AND workloadname = 'MyWorkload'
AND pr.resultnametag = 'MyPPIs';
```

Assume the following output from this query:

<table>
<thead>
<tr>
<th>queryname</th>
<th>expressiontext</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>PARTITION BY RANGE_N(transaction_date BETWEEN DATE '2000-01-01' AND DATE '2006-06-30') EACH INTERVAL '7' DAY);</td>
</tr>
</tbody>
</table>

### Stage 6: View the Estimated Cost Improvements

In the sixth stage of the process, you view the estimated query cost improvements when using the recommended PPI for this workload:

```sql
SELECT queryname, originalcost, newcost
FROM qcd.partitionrecommendations AS pr,
     qcd.workload AS wl,
     qcd.query AS q
WHERE pr.workloadid = wl.workloadid
AND pr.queryid = q.queryid
AND workloadname = 'MyWorkload'
AND pr.resultnametag = 'MyPPIs';
```

Assume the following output from this query:

<table>
<thead>
<tr>
<th>queryname</th>
<th>originalcost</th>
<th>newcost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>4.530000E 004</td>
<td>2.580000E 002</td>
</tr>
</tbody>
</table>

### Stage 7: Validate the Recommendations

In the seventh stage of the process, you validate the recommendations made by the analysis.

You can optionally validate the recommendations on your production system.

**Stage 7a:** Turn on Validation mode for the current session:

```
DIAGNOSTIC "VALIDATE INDEX" ON FOR SESSION;
```

**Stage 7b, part 1:** Simulate creation of the recommended NUSI:

```
CREATE INDEX (product_category) ON Products;
```

6. The formatting of this output is arbitrary and might not correspond with the actual output generated by your system.
Stage 7b, part 2: Simulate altering the table definition for the recommended partitioning:

```
ALTER TABLE SalesHistory
    MODIFY PRIMARY INDEX PARTITION BY RANGE_N(transaction_date
        BETWEEN DATE '2000-01-01'
        AND DATE '2006-06-30'
        EACH INTERVAL '7' DAY);
```

Note that this ALTER TABLE statement is not valid outside of Validation mode.

Stage 7c, part 1: Simulate collecting statistics on the recommended NUSI:

```
COLLECT STATISTICS FOR SAMPLE 100 PERCENT
    INTO qcd ON SalesHistory COLUMN(transaction_date);
```

The system collects statistics on the underlying column data.

Stage 7c, part 2: Simulate collecting statistics on the recommended partitioning column:

```
COLLECT STATISTICS ON SalesHistory COLUMN(transaction_date);
```

Stage 7d: Capture the new query plans for the queries in the workload:

```
INSERT EXPLAIN INTO qcd AS q1_validate
    SELECT product_code, SUM(quantity_sold)
    FROM SalesHistory
    WHERE transaction_date >= DATE '2006-01-01'
    GROUP BY product_code;

INSERT EXPLAIN INTO qcd AS q2_validate
    SELECT SUM(quantity_avail), AVG(quantity_avail),
        MIN(quantity_avail)
    FROM Products
    WHERE product_category = 'Laptop'
```

Stage 7e: Turn off validation mode:

```
DIAGNOSTIC "VALIDATE INDEX" NOT ON FOR SESSION;
```

Stage 7f, part 1: Examine the steps in the captured plan for query Q1 to see if partition elimination occurred:

```
SELECT steptext
    FROM qcd.querystepsview
    WHERE QueryId =
        (SELECT MAX(QueryId)
            FROM qcd.query
            WHERE queryname = 'q1_validate')
    ORDER BY stepnum;
```

Assume the following partial output from this query:

```
steptext
-------
3) We do an all-AMPs SUM step to aggregate from 26 partitions of
   "DFES.SalesHistory with a condition of ("DFES.SalesHistory.transaction_date >=
   DATE '2006-01-01')"
```

7. Note that you can perform a similar query for Q2 as well. That query is omitted from this example because it is essentially redundant.
Stage 7f, part 2: Examine the step costs in the captured plan for Q1 to see if costs are reduced:\(^8\)

\[
\begin{align*}
\text{SELECT stepnum, stepkind, cost} \\
\text{FROM qcd.querystepsview} \\
\text{WHERE cost} &> 0 \\
\text{AND QueryId} = \\
&\left(\text{SELECT MAX(QueryId)}\right) \\
&\left(\text{FROM qcd.query}\right) \\
&\left(\text{WHERE queryname} = \text{'q1_validate'}\right) \\
\text{ORDER BY stepnum;}
\end{align*}
\]

Stage 8: Execute the Indexing and Partitioning Recommendations

Retrieve the recommended SQL DDL to recreate the table with the recommended PPI:

\[
\begin{align*}
\text{SELECT recreatetext} \\
\text{FROM qcd.partitionrecommendations AS pr,} \\
&\text{qcd.workload AS wl,} \\
&\text{qcd.query AS q} \\
\text{WHERE pr.workloadid} &= \text{wl.workloadid} \\
\text{AND pr.queryid} &= \text{q.queryid} \\
\text{AND workloadname} &= \text{'MyWorkload'} \\
\text{AND pr.resultnametag} &= \text{'MyPPIs'};
\end{align*}
\]

Assume this request returns the following result:\(^9\)

\[
\begin{align*}
\text{recreatetext} \\
\text{-----------} \\
\text{CREATE TABLE _PPI_SalesHistory AS (} \\
&\text{ SELECT *} \\
&\text{ FROM SalesHistory)} \\
\text{WITH NO DATA;} \\
\text{ALTER TABLE _PPI_SalesHistory} \\
\text{MODIFY PRIMARY INDEX PARTITION BY} \\
\text{ RANGE N(transaction_date} \\
&\text{ BETWEEN DATE '1997-01-01'} \\
&\text{ AND DATE '2006-12-31'} \\
&\text{ EACH INTERVAL '1' DAY);} \\
\text{INSERT INTO _PPI_SalesHistory} \\
&\text{ SELECT *} \\
&\text{ FROM SalesHistory;} \\
\text{DROP TABLE SalesHistory;} \\
\text{RENAME TABLE _PPI_SalesHistory TO SalesHistory;} \\
\end{align*}
\]

8. Note that you can perform a similar query for Q2 as well. That query is omitted from this example because it is essentially redundant.

9. The formatting of this output is arbitrary and might not correspond with the actual output generated by your system.
Examples of Partitioned Primary Indexes the Index Wizard/INITIATE PARTITION ANALYSIS Might Recommend

Introduction

This topic provides examples of PPI partitioning expressions that the Index Wizard and INITIATE PARTITION ANALYSIS might recommend, as well as example that the Index Wizard and INITIATE PARTITION ANALYSIS never recommend.

The examples all use the following tables:

```sql
CREATE TABLE sales_history (  
  product_code CHARACTER(8),  
  store_number INTEGER,  
  transaction_date DATE,  
  quantity_sold INTEGER,  
  other_columns CHARACTER(50))  
PRIMARY INDEX(product_code,store_number,transaction_date);

CREATE TABLE products (  
  product_code CHARACTER(8),  
  description VARCHAR(50),  
  prod_division_number INTEGER,  
  product_category CHARACTER(10),  
  price DECIMAL(10,2),  
  quantity_avail INTEGER)  
PRIMARY INDEX(product_code);

CREATE TABLE call_detail (  
  phone_number DECIMAL(10),  
  call_date DATE,  
  call_start TIMESTAMP,  
  call_duration INTEGER,  
  caller_id INTEGER,  
  other_columns CHARACTER(30))  
PRIMARY INDEX(phone_number,call_date)  
PARTITION BY RANGE_N(CAST phone_number/100000.00000 AS INTEGER)  
  BETWEEN 0  
  AND 99999  
  EACH 2);

CREATE TABLE supplier (  
  supplier_code CHARACTER(8),  
  supp_region_code INTEGER,  
  phone_number DECIMAL(10),  
  supplier_name VARCHAR(50),  
  other_columns CHARACTER(30))  
UNIQUE PRIMARY INDEX(supplier_code);
```
Examples of Partitioning Expressions the Index Wizard/INITIATE PARTITION ANALYSIS Might Recommend for These Tables

The following expressions are examples of partitioning expressions that Partition Analysis might recommend for the tables defined in “Introduction” on page 664. Note that the recommended partitioning expressions are all based on the RANGE_N function. Teradata Index Wizard only recommends partitioning expressions that are based on RANGE_N.

```
PARTITION BY RANGE_N(transaction_date
    BETWEEN DATE '2002-01-01'
    AND DATE '2006-12-31'
    EACH INTERVAL '7' DAY);

PARTITION BY RANGE_N(transaction_date
    BETWEEN DATE '2002-01-01'
    AND DATE '2006-12-31'
    EACH INTERVAL '1' MONTH);

PARTITION BY RANGE_N(store_number
    BETWEEN 100
    AND 999
    EACH 1);

PARTITION BY RANGE_N(store_number
    BETWEEN 100
    AND 999
    EACH 10);
```

Note that in the following example, the column `prod_division_number` is not a component of the NUPI for the `products` table:

```
PARTITION BY RANGE_N(prod_division_number
    BETWEEN 1
    AND 99
    EACH 1);
```

Note that in the following example, the system recommends a change to the existing partitioning expression for the `call_detail` table:

```
PRIMARY INDEX(phone_number, call_date)
PARTITION BY RANGE_N(call_date
    BETWEEN DATE '2002-01-01'
    AND DATE '2006-12-31'
    EACH INTERVAL '7' DAY);
```
Optimizer Statistics Recommendations and the Teradata Statistics Wizard

Introduction

The principal focus of query tuning is to provide reliable summary information about the data to the Optimizer. This is done by collecting accurate statistics, which are then stored in a synoptic data structure known as an interval histogram (see “Interval Histograms” on page 161). The correct choice of the column and index sets on which statistics should be collected can help the Optimizer generate better query plans, dramatically improving query performance, and reducing the collection overhead. It can be difficult to understand how the Optimizer uses statistics, and so to decide what statistics are needed, without an automated method to recommend them. That automated method is the Teradata Statistics Wizard, which is a client-based GUI interface for obtaining statistics recommendations for particular queries or query workloads that you submit to it for analysis.

The best place to implement automated recommendations is within the Optimizer itself. There are many advantages of building the automation logic inside the Optimizer:

- Such a system can recommend statistics that the Optimizer has looked for, but found not to be available. These are the only correct recommendations for new statistics to be collected.
- Such recommendations are adaptive for two reasons:
  - Because they are sensitive to all enhancements made to the Optimizer software.
  - Because they are sensitive to new kinds of statistics that can be collected, such as those on system-derived PARTITION columns.
- Such recommendations are flexible because they can also be made for join or hash indexes. Simply adding a new join index to a system qualifies it for being a potential recommendation for any given workload.

This situation also leads to the following indirect advantage: If you create a join or hash index, but do not collect any statistics on it, and the Optimizer does not recommend any statistics for it, then you can safely assume that the join or hash index does not qualify to be recommended for use by the given query or workload.

The following documentation provides information about what kinds of recommendations the Statistics Wizard gives, given the previously stated cardinality estimation enhancements, and how to interpret them. Also see Teradata Statistics Wizard User Guide for information about how to use the Statistics Wizard.
Recommendations on the Primary Index and Secondary Indexes

If no primary or secondary index statistics are available, the Optimizer generates the information it needs using random AMP sampling (see “Random AMP Sampling” on page 178).

However, if there are any single-table predicates on the index columns, then the information acquired by random AMP sampling is not sufficient, with the exception of unique index and equality predicates. The information in an interval histogram is required to more accurately estimate the selectivity of these single-table predicates. In such cases, the recommendations to collect statistics on these indexes are given with High Confidence.

If the index columns are used only to make joins, then the interval histogram statistics might not be required. In such cases, the recommendations are given with Low Confidence. The Statistics Wizard recommends implementing its Low Confidence recommendations on indexes when the estimations are off or the data is not fairly uniform.

See “StatsRecs” on page 617 for information about the various confidence levels reported by the Index Wizard. The confidence level of an index recommendation is stored in the Levels column of this QCD table.

No recommendations are provided for unique indexes if there are no nonequality predicates on those indexes. Similarly, no recommendations are made for a NUPI if all-AMP sampling is done, and an interval histogram is not required to make the necessary estimates.

Special Recommendations on PPI Tables

If there are any predicates on the partitioning columns of a PPI and the system does partition elimination, or if there are any join predicates on the primary index or the partitioning column of the PPI table, then the Statistics Wizard recommends single-column partition statistics on the system-derived PARTITION column with High Confidence.

If the system does partition elimination, then multicolumn PARTITION statistics are recommended with Low Confidence on the system-derived PARTITION column and other join columns. This helps to derive the column correlations for the other join columns based on the qualified partitions.

For example, suppose you submit the following query:

```sql
SELECT *
FROM t1_ppi, t2
WHERE t1_ppi.partition_column BETWEEN 10 AND 20
AND t1_ppi.x1 = t2.y1;
```

If the qualified partitions are (4, 7) then it is helpful for the Optimizer to derive the demographics of the join column t1_ppi.x1 for the given partitions (4, 7) in order to provide more accurate estimation and costing. The recommendations need to be implemented only if the partitioning column and the join columns are correlated.
**Single-Table Cardinality Estimations**

The single-table estimation logic examines all single-table predicates for each relation and estimates the combined selectivity or the total qualified rows using single-column or multicolumn statistics.

The Statistics Wizard recommends statistics for every usable predicate column with High Confidence if they are not already available. Note that statistics are not used for some complex predicates. When the Statistics Wizard encounters such predicates, it does not provide recommendations for them.

For example, \( x_1 > 10 \rightarrow \) histogram on \( x_1 \) is used to estimate the selectivity for this term, so a recommendation is given to collect statistics on \( x_1 \) if none are already available.

```sql
CASE
  WHEN x1 > 10
  THEN y1
  ELSE x1 - 10
END = 50 \rightarrow \) histogram on \( x_1 \)
```

**Special Considerations for Sampled and Stale Statistics**

There are no special considerations for sampled or stale statistics. The Statistics Wizard considers them as it would any other statistics when it recommends statistics that should be collected for the query or workload you submit for analysis. In other words, if there are sampled or stale statistics, the Statistics Wizard assumes they are current and uses them to make its recommendations. This is yet another reason why it is important to keep your statistics up to date.

**Single-Column Statistics**

Statistics Wizard recommendations:

- Single-column statistics recommendations are given with High Confidence for every predicate column on which no statistics are available.

- If the Optimizer has information other than statistics available to it, such as the number of unique values as determined from CHECK or Referential Integrity constraints, the Optimizer can use them for equality predicates. In this case, single-column recommendations are given with Low Confidence.

For example, consider the predicate \( t1.x1 > 10 \) AND \( t1.y1 < 20 \). If statistics are not available on columns \( t1.x1 \) and \( t1.y1 \), the recommendation is made to collect statistics on \( t1.x1 \) and \( t1.y1 \) with High Confidence.
For the predicate \( t1.x1 = 10 \), if statistics are not available and the number of unique values is derived using either a CHECK or referential integrity constraint, then the Optimizer uses the column density, which is \( \frac{1}{\text{NumUniqueValues}} \), as the selectivity for the predicate. In this case, the recommendations are still given on \( t1.x1 \), but with Low Confidence.

- If the estimates are made using extrapolation, then it is very likely that the statistics are stale, so the statistics are recommended only with Low Confidence.

In general, if the distribution of a column is fairly uniform and close to the actual value, then you do not need to implement recommendations made with Low Confidence. You should always implement recommendations the Statistics Wizard makes with High Confidence, however.

**Multicolumn Statistics**

Multicolumn statistics recommendations are given with High Confidence for multiple, single-table, equality predicates if the cumulative size of these columns does not exceed 16 bytes.

For example, given the predicates, \( t1.x1 = 10 \) and \( t1.y1 = 20 \), the recommendation is given with High Confidence to collect multicolumn statistics on \((t1.x1, t1.y1)\).

Multicolumn statistics are required if the predicate columns are correlated. Because the column correlations are not known to the Optimizer, it might recommend both multicolumn and single-column collections for the same set of columns. Multicolumn recommendations are implemented only when there is high degree of correlation among the columns or when the estimates are known to be inaccurate.

**Join Cardinality Estimates, RowsPerValue, and Skew Detection**

Statistics Wizard recommendations:

- If there is a single join predicate and statistics are missing on either side of the condition, the Statistics Wizard makes single-column statistics recommendations with High Confidence.

- If there are multiple join predicates, then the Optimizer needs to know the demographics on the combined columns to accurately estimate the join cardinality, rows per value, and skew. In the absence of multicolumn statistics, the individual demographics are combined by assuming the columns are independent. It is not a problem to combine individual statistics for independent column combinations such as \( \text{car make} \) and \( \text{color} \), but the Optimizer must have true multicolumn statistics for highly correlated column combinations such as \( \text{car make} \) and \( \text{model} \).

Because the column correlations are not known to the Optimizer, assuming the columns are correlated, the Statistics Wizard always recommends multicolumn statistics on all the join columns in a predicate with High Confidence. It also recommends some optional statistics on the single columns or on a subset of the join columns with Low Confidence when it is prohibitively expensive to collect all the multicolumn statistics or different combinations of multicolumn statistics.
For example, if the join columns are \((a_1, b_1, c_1, d_1)\) and individual statistics have been collected on all of them, then the Statistics Wizard recommends multicol

- If there are some multicol

  If there are some usable single-table predicates that can help to derive the column correlations for the join columns, then the Statistics Wizard recommends multicol

You should always implement these recommendations if there is a high degree of correlation between the single-table predicate columns and the join columns. For example, given \(a_1 > 10\) and \(b_1 = b_2\), then the Optimizer can use multicol

You should first apply the recommendations that incur the least overhead and help the largest number of queries, such as statistics on individual fields. If those statistics do not help, then you should collect the recommended multicol

**Aggregate Estimates**

Similar to the Join Cardinality estimation logic, the Optimizer needs to have multicol

If the aggregation is in the inner query block (aggregate derived table or view) and if its estimates are propagated to the outer block, then you should follow the guidelines similar to the situation for Join Cardinality estimates.

If this is the final aggregation, and the estimate of the final result spool is not that important, then you can ignore the recommendations. If Teradata Dynamic Workload Manager rules are implemented to abort the queries based on the estimated number of results rows, then the final spool estimates might be important.
Recommendations When Single-Table Join Indexes Are Used

When single-table join indexes are used for estimates, you might not need to collect statistics on the base table columns that are covered by the join index with some exceptions.

For example, statistics are still required on the base table index columns that have single-table predicates in order to do access path analysis.

Similarly, statistics might be required if the covered columns are used as join columns or grouping columns in a GROUP BY clause.

Consider the following query and join index definition:

```sql
SELECT * 
FROM t1 
WHERE nusi_col = 10 
AND a1 > 20;

CREATE JOIN INDEX ji AS 
SELECT a1, b1 
FROM t1 
WHERE nusi_col = 10 
AND a1 > 20;
```

In this case, the join index row count gives the final selectivity for `t1`; however, statistics on `nusi_col` might still be required in order to do access path analysis for the base table.

Now consider another query and join index definition:

```sql
SELECT * 
FROM t1, t2 
WHERE nusi_col = 10 
AND a1 > 20 
AND t1.a1 = t2.a2;

CREATE JOIN INDEX ji AS 
SELECT a1, b1 
FROM t1 
WHERE nusi_col = 10 
AND a1 > 20;
```

In this case, the join index cardinality provides the final selectivity for `t1`; however, statistics on `nusi_col` might still be required in order to do access path analysis for the base table.

When usable single-table join indexes are found for single-table cardinality estimations, and the columns referenced in the join predicates or GROUP BY clause are projected from the join index, then the Statistics Wizard recommends statistics on the join index columns if they are not available. For the previous example, the Statistics Wizard recommends collecting statistics on `ji.a1` since the column `a1` is projected in the join index. If this column is not projected, then the Statistics Wizard recommends collecting statistics on the base table columns `t1.a1`. 
Flow of Single-Table Cardinality Estimation

Suppose you define the following base table and join index:

```sql
CREATE TABLE t1
(
    a1 INTEGER,
    b1 INTEGER,
    c1 CHARACTER(1) CHECK (c1 IN ('a', 'b','c','d')),
    d1 DATE);
CREATE JOIN INDEX ji1 AS
    SELECT b1, d1
    FROM t1
    WHERE a1=10
    AND b1>15;
```

You then submit the following query:

```sql
SELECT *
FROM t1
WHERE a1=10
AND b1>30
AND d1 = '12/03/2005'
AND c1='c';
```

The following demographic information is available to the Optimizer for processing this query:

- The interval histogram for join index `ji1.b1` indicates that 300 out of the total of 400 rows in `ji1` satisfy the predicate `ji1.b1>30`.
- The interval histogram for `t1.d1` shows that it has 100 values, and that 100 of the 10,000 rows satisfy the predicate `d1 = '12/03/2005'`.
- The interval histogram for `t1.a1` shows that it has 20 values, and that 500 of the 10,000 rows satisfy the predicate `a1=10`.
- There is a CHECK constraint defined on `t1.c1` that permits only the values a, b, c, or d.
- The value mapping from `(a1,b1)` to `d1` is 1:5.

The following steps show how the cardinality of the query result is estimated:

1. Use all available sources to calculate the selectivity for applicable predicates. The selectivity matrix is filled in as follows:

<table>
<thead>
<tr>
<th>Predicates</th>
<th>Selectivity Type</th>
<th>Number of Rows Selected</th>
<th>Selectivity</th>
<th>Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (Reserved for final selectivity)</td>
<td>Combined</td>
<td>10000</td>
<td>1.00</td>
<td>F</td>
</tr>
<tr>
<td>a1=10 AND b1&gt;30</td>
<td>Join Index (ji1)</td>
<td>300</td>
<td>0.03</td>
<td>F</td>
</tr>
<tr>
<td>a1=10</td>
<td>Field (histogram)</td>
<td>500</td>
<td>0.05</td>
<td>F</td>
</tr>
<tr>
<td>d1='12/03/2005'</td>
<td>Field (histogram)</td>
<td>100</td>
<td>0.01</td>
<td>F</td>
</tr>
<tr>
<td>c1='c'</td>
<td>Field (from CHECK constraint)</td>
<td>2500</td>
<td>0.25</td>
<td>F</td>
</tr>
</tbody>
</table>
2 Use the predicate entry in the selectivity matrix with the highest selectivity ($d_1 = '12/03/2005'$) and mark any predicates that overlap with it as Used.

The only overlapping predicate is itself, so it is marked as Used.

Set the combined Selectivity using the highest selectivity entry.

<table>
<thead>
<tr>
<th>Predicates</th>
<th>Selectivity Type</th>
<th>Number of Rows Selected</th>
<th>Selectivity</th>
<th>Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1 = '12/03/2005'$</td>
<td>Combined</td>
<td>100</td>
<td>1.00</td>
<td>F</td>
</tr>
<tr>
<td>$a_1=10 \text{ AND } b_1&gt;30$</td>
<td>Join Index ($ji1$)</td>
<td>300</td>
<td>0.03</td>
<td>F</td>
</tr>
<tr>
<td>$a_1=10$</td>
<td>Field (histogram)</td>
<td>500</td>
<td>0.05</td>
<td>F</td>
</tr>
<tr>
<td>$d_1 = '12/03/2005'$</td>
<td>Field (histogram)</td>
<td>100</td>
<td>0.01</td>
<td>T</td>
</tr>
<tr>
<td>$c_1='c'$</td>
<td>Field (from CHECK constraint)</td>
<td>2500</td>
<td>0.25</td>
<td>F</td>
</tr>
</tbody>
</table>

3 Choose the next highest selective predicate entry, which is $a_1=10 \text{ AND } b_1>30$, and mark any predicates that overlap with it as Used.

The overlapping predicates are $a_1=10 \text{ AND } b_1>30 \text{ AND } d_1 = '12/03/2005'$. Note that $a_1=10 \text{ AND } b_1>30$ is a superset of $a_1=10$, so $a_1=10$ is not included.

Combine its selectivity with the previously calculated combined selectivity. Because there is a mapping relationship between ($a_1,b_1$) and $d_1$, the combined selectivity is determined as follows:

$$\text{Combined selectivity} = 0.03 \times 0.01 \times \frac{100}{5} = 0.006$$

<table>
<thead>
<tr>
<th>Predicates</th>
<th>Selectivity Type</th>
<th>Number of Rows Selected</th>
<th>Selectivity</th>
<th>Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1=10 \text{ AND } b_1&gt;30 \text{ AND } d_1 = '12/03/2005'$</td>
<td>Combined</td>
<td>60</td>
<td>0.006</td>
<td>F</td>
</tr>
<tr>
<td>$a_1=10 \text{ AND } b_1&gt;30$</td>
<td>Join Index ($ji1$)</td>
<td>300</td>
<td>0.03</td>
<td>T</td>
</tr>
<tr>
<td>$a_1=10$</td>
<td>Field (histogram)</td>
<td>500</td>
<td>0.05</td>
<td>T</td>
</tr>
<tr>
<td>$d_1 = '12/03/2005'$</td>
<td>Field (histogram)</td>
<td>100</td>
<td>0.01</td>
<td>T</td>
</tr>
<tr>
<td>$c_1='c'$</td>
<td>Field (from CHECK constraint)</td>
<td>2500</td>
<td>0.25</td>
<td>F</td>
</tr>
</tbody>
</table>

4 Choose the predicate with the next highest selectivity, $c_1='c'$, and mark any of the predicates that overlap with it as Used. The overlapping predicates are $a_1=10 \text{ AND } b_1>30 \text{ AND } d_1 = '12/03/2005' \text{ AND } c_1='c'$.

Combine its selectivity with the selectivities that have already been combined so. Because there is no value mapping relationship between ($a_1,b_1,d_1$) and $c_1$, the combined selectivity is calculated using a fudge factor, as follows:

$$\text{Combined selectivity} = 0.006 \times \text{MAX}(0.25, 0.75) = 0.006 \times 0.75 = 0.0045$$
The final combined selectivity for the query is 0.0045, so the cardinality of the result is determined as follows:

Number of qualifying rows = Combined selectivity \times \text{Total number of rows} = 0.0045 \times 10000 = 45

This means that the WHERE clause predicate filter for this query qualifies 45 rows.

Flow of UNION Scan Handling

Consider the following table definition and query:

```sql
CREATE TABLE t1(
    a INTEGER,
    b INTEGER,
    c INTEGER,
    d INTEGER,
    x INTEGER,
    y INTEGER);

CREATE INDEX (a,b) ON t1;
CREATE INDEX (b,c) ON t1;

SELECT * FROM t1
WHERE a IN (1,2,4,5,6,7,8,9,10) AND b=3 AND c IN (22,33);
```

The following process stages show the index recommendation flow for a UNION scan.

1. Build a data structure with pointers to the relevant predicates for each index:

<table>
<thead>
<tr>
<th>Predicates</th>
<th>Selectivity Type</th>
<th>Number of Rows Selected</th>
<th>Selectivity</th>
<th>Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1=10 AND b1&gt;30 AND dl='12/03/2005' AND cl='c'</td>
<td>Combined</td>
<td>45</td>
<td>0.0045</td>
<td>T</td>
</tr>
<tr>
<td>a1=10 AND b1&gt;30</td>
<td>JI (ji1)</td>
<td>300</td>
<td>0.030</td>
<td>T</td>
</tr>
<tr>
<td>a1=10</td>
<td>Field (histogram)</td>
<td>500</td>
<td>0.050</td>
<td>T</td>
</tr>
<tr>
<td>dl='12/03/2005'</td>
<td>Field (histogram)</td>
<td>100</td>
<td>0.010</td>
<td>T</td>
</tr>
<tr>
<td>cl='c'</td>
<td>Field (from CHECK constraint)</td>
<td>2500</td>
<td>0.250</td>
<td>T</td>
</tr>
</tbody>
</table>

FOR this index ... The data structure contains a pointer to the predicate ...

| NUSI(a,b) | a IN (1,2,4,5,6,7,8,9,10)  
| NUSI(b,c) | b=3  
| NUSI(b,c) | c IN (22,33) |
2 Cost all available UNION scans.
3 Select the best access path.

In this case, the best access path is a UNION scan on NUSI(b,c).

Generate a Retrieve step using the cursor. No change is made to the query condition.

**Aggregation Access Path Via Covering and Noncovering Indexes**

The following example shows an example of an aggregation access path developed from the information derived from both covering and noncovering indexes.

Consider the following table definition:

```sql
CREATE SET TABLE t1, NO FALLBACK, NO BEFORE JOURNAL, NO AFTER JOURNAL, CHECKSUM = DEFAULT ( 
    a1 INTEGER, 
    b1 INTEGER, 
    c1 CHARACTER(2) CHARACTER SET LATIN NOT CASESPECIFIC, 
    d1 CHARACTER(2) CHARACTER SET LATIN CASESPECIFIC, 
    e1 CHARACTER(1) CHARACTER SET LATIN UPPERCASE NOT CASESPECIFIC, 
    f1 INTEGER, 
    g1 INTEGER) 
PRIMARY INDEX pi ( a1 ), 
UNIQUE INDEX usi1 ( b1 ,c1 ), 
INDEX si1 ALL (c1,f1), 
INDEX si2 (d1,e1), 
INDEX si3 (c1,f1,g1), 
INDEX si4 (c1,g1), 
INDEX si5 (d1,g1), 
INDEX si6 (f1); 
```

The following points are the analysis of the data columns and indexes:

- The CHARACTER field `c1` is neither CASESPECIFIC nor UPPERCASE in the base table; therefore, it is considered to be a special character field. CHARACTER field `d1` is defined as CASESPECIFIC and `e1` is defined as UPPERCASE, so these two fields are not special CHARACTER fields.
- USI `usi1` contains the special CHARACTER field `c1`, but all values of `c1` are stored in the corresponding USI subtable.
- NUSI `si1` contains the special CHARACTER field `c1`, but because it is defined using the ALL option, all values of `c1` are also stored in the corresponding NUSI subtable.
• NUSI si2 contains \((d1, e1)\) which are not special CHARACTER fields, so all values of \((d1, e1)\) are stored in the NUSI subtable.
  The same is true for si5.
• NUSI si3 contains the special character field \(c1\). This NUSI was not defined using the ALL option, so only the uppercase values of \(c1\) are stored in the NUSI subtable.
  The same is true for si4.

Analyses of some queries and covering indexes are provided in the following examples:

**Query 1 (Covering Index Access, Partial-Covering Index Access and Other Access)**

```sql
SELECT COUNT (c1), UPPER (c1)
FROM t1
WHERE f1 = 10
AND c1 = 'aa';
```

The following things are true for this query:

• *si1* is a covering index with a key \((c1='aa', f1=10)\).
  This query can be answered by accessing the subtable of *si1* using the hash value of \(('aa', 10)\).

• *si3* is also a covering index because the special character field \(c1\) appears only as an argument of the COUNT and UPPER operators in the select list, and there is no CS condition on \(c1\) in the WHERE clause. However, there is no key on *si3* because \(g1\) does not have a value bound to it.
  This query can be answered by doing a full table scan on the *si3* subtable with the constraint \((c1='aa' and f1=10)\).

• *si4* is a partially covering index because it covers partial column \((c1)\) in the WHERE clause.
  This query can be answered by doing a full table scan on the *si4* subtable with the constraint \((c1='aa')\) to get the row ID values, and then using those row IDs to access the base table.

• *si6* is a partially covering index because it covers partial column \((f1)\) in the WHERE clause.
  There is also a key defined on it \((f1 = 10)\).
  This query can be answered by accessing the *si6* subtable with the hash value of \((10)\) to get the qualified row ID values, and then using those row IDs to access the base table rows.
• This query can always be answered by a Full Table Scan of the base table.

The access path planning logic costs each access path to get the best one.
Query 2 (Partial-Covering Index Access and Other Access)

```sql
SELECT MIN(c1)
FROM t1
WHERE c1 = 'aa'
AND g1 = 10;
```

The following things are true for this query:

- $si4$ is a partially covering index with a key ($c1='aa'$, $g1=10$) because the special character column $c1$ is specified for something other than the COUNT or UPPER operators in the select list.
  
  This query can be answered by accessing the $si4$ subtable first, using the hash value of ($'aa$', 10) to get the base table row ID values, and then using those row IDs to access the base table rows.

- $si3$ is also a partially covering index, but there is no key on it because no value for $f1$ is specified.
  
  This query can be answered by scanning the $si3$ subtable first with the constraint ($c1 = 'aa'$ AND $g1 = 10$) to get the qualified base table row ID values, and then using those row IDs to access the base table rows.

- $si1$ is a partially covering index because it covers the partial column ($c1$) in the WHERE clause. No key is defined on it.
  
  This query can be answered by scanning the $si1$ subtable first with the constraint ($c1 = 'aa'$) to get the qualified base table row ID values, and then using those row IDs to access the base table rows.

- $si5$ is a partially covering index because it covers partial column ($g1$) in the WHERE clause. No key is defined on it.
  
  This query can be answered by scanning the $si5$ subtable first with the constraint ($g1 = 10$) to get the qualified base table row ID values, and then using those row IDs to access the base table rows.

- This query can always be answered by a Full Table Scan of the base table.

The access path planning logic costs each access path to get the best one.
This chapter describes Target Level Emulation, a set of tools used to set up a test system with environmental cost parameters, random AMP sample statistics from a production system having a different configuration, as well as DBS Control record information from that system that affect the Optimizer. 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.

For additional information about environmental cost parameters, see “Environmental Cost Factors” on page 296.

For additional information about random AMP sampling, see “Random AMP Sampling” on page 178.

For additional information about using Target Level Emulation from a client utility GUI interface, see *Teradata System Emulation Tool User Guide*.

The SQL diagnostic statements that support Target Level Emulation are documented in *SQL Data Manipulation Language*.

---

1. See the footnote to “What Is a Query Optimizer?” on page 127 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, table-level random AMP sample statistics, and Optimizer-relevant DBS Control information from that environment and storing it in the relational tables SystemFE.Opt_Cost_Table, SystemFE.Opt_RAS_Table, and SystemFE.Opt_DBSCtl_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 and DBS Control parameter settings on SQL request 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 27 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.

2. The DBS Control information stored in SystemFE.Opt_DBSCtl_Table supports both forms of target level emulation.
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 SQL Data Manipulation Language 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 dynamically. 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 Tools and Utilities**

The target level emulation feature is closely related to other Teradata Database support tools, particularly those detailed in the following list:

- Query Capture Facility.
  See Chapter 6: “Query Capture Facility.”

- Visual Explain Tool.

- Teradata System Emulation Tool (TSET).

- Teradata Index Wizard.

- CHECK STATISTICS option of the INSERT EXPLAIN and DUMP EXPLAIN statements.³
  See the documentation for these statements in SQL Data Manipulation Language.

- Teradata Statistics Wizard.

- Statistics Collection tool (Teradata Manager).
  See Teradata Manager User Guide.

³. The CHECK STATISTICS functionality of the INSERT EXPLAIN and DUMP EXPLAIN statements produces superior results to those of the Teradata Statistics Wizard. Because of this, you should use CHECK STATISTICS in place of the Teradata Statistics Wizard.
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:
Procedures to Enable Target Level Emulation and OCES DIAGNOSTIC Statements With 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 To Enable Target Level Emulation

The following is a high level procedure for performing this task:

1. Run the DIP utility script DIPSYSFE to create the optimizer cost values table SystemFE.Opt_Cost_Table, the random AMP sample statistical values table SystemFE.Opt_RAS_Table, and the DBS Control parameters table SystemFE.Opt_DBSCtl_Table.

2. Use the DBS Control utility to enable Target Level Emulation on your test system by setting DBS Control general field 22, Target Level Emulation, to TRUE.

3. End of procedure.

Procedure To Enable Optimizer Cost Estimation Subsystem DIAGNOSTIC Statements With Target Level Emulation

To enable optimizer cost profile system DIAGNOSTIC statements in combination with Target Level Emulation, you must set the EnableCostProfileTLE field of DBS Control TRUE.

If the DBS Control field EnableSetCostProfile (see “Cost Profile Activation” on page 289) is set to 0, then this field is automatically set FALSE by the system and cannot be changed.

This field is initialized to FALSE. As long as the value for EnableCostProfileTLE is set FALSE, attempts to use combined Optimizer Cost Estimation Subsystem (OCES) and TLE DIAGNOSTIC syntax abort and return an error to the requestor.

1. Enable TLE on your system (see “Procedure To Enable Target Level Emulation”).

2. Run the DBS Control utility (see Utilities).

3. Ensure that the EnableSetCostProfile field is set to a value other than 0.

4. Set the EnableCostProfileTLE field TRUE.

5. End of procedure.
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 follows:

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.

3. **End of process.**

You can use the Teradata System Emulation Tool from a Windows client to perform this data capture and application. See *Teradata System Emulation Tool User Guide* for details.

If you do not use Teradata System Emulation Tool, 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 Teradata System Emulation Tool**

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 is a high level overview of this procedure:

1. Depending on your requirements, perform one or both of the following steps:
   a. Populate `SystemFE.Opt_Cost_Table` and `SystemFE.Opt_DBSCtl_Table` on the target system by running the DIAGNOSTIC DUMP COSTS statement (see SQL Data Manipulation Language).
      This operation extracts the Optimizer environmental cost value and DBS Control 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.
   b. Populate `SystemFE.Opt_RAS_Table` on the target system by running the DIAGNOSTIC DUMP SAMPLES statement (see SQL Data Manipulation Language).
      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.
   c. End of subprocedure.

2. Log onto Teradata System Emulation Tool 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 and DBS Control information to be captured for loading onto the test system.

4. Collect the Optimizer cost value information or random AMP statistics, the DBS Control information, 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 Teradata System Emulation Tool 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 Teradata System Emulation Tool 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` and to `SystemFE.Opt_DBSCtl_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.
7 Depending on your requirements, perform one or both of the following steps:
   a Run the DIAGNOSTIC SET COSTS SQL statement (see SQL Data Manipulation Language) 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 693 for an illustration.
   b Run the DIAGNOSTIC SET SAMPLES SQL statement (see SQL Data Manipulation Language) 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 693 for a comparable illustration.
   c End of subprocedure.

8 End of procedure.

Note that you can also use Teradata System Emulation Tool 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 Using SQL DIAGNOSTIC Statements**

The following list provides a high level procedure for capturing Optimizer environmental cost information from a production system and applying that information to a test system:

1 Depending on your requirements, perform one or both of the following steps:
   a Populate SystemFE.Opt_Cost_Table and SystemFE.Opt_DBSCtl_Table on the target system by running the DIAGNOSTIC DUMP COSTS statement (see SQL Data Manipulation Language).
      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.
   b Populate SystemFE.Opt_RAS_Table on the target system by running the DIAGNOSTIC DUMP SAMPLES statement (see SQL Data Manipulation Language).
      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.
   c End of subprocedure.
2 Depending on your requirements, perform one or both of the following steps:
   a 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
      • CLlv2
      • BTEQ SQL script
   b 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
      • CLlv2
      • BTEQ SQL script
   c End of subprocedure.
3 Depending on your requirements, perform one or both of the following steps:
   a Import the rows into SystemFE.Opt_Cost_Table on the test system using an appropriate utility from the following list:
      • FastLoad
      • MultiLoad
      • TPump
      • CLlv2
      • BTEQ SQL script
   b Import the rows into SystemFE.Opt_RAS_Table on the test system using an appropriate utility from the following list:
      • FastLoad
      • MultiLoad
      • TPump
      • CLlv2
      • BTEQ SQL script
   c End of subprocedure.
4 Import the rows into `SystemFE.Opt_DBSCtl_Table` on the test system using an appropriate utility from the following list:
   - FastLoad
   - MultiLoad
   - TPump
   - CLIv2
   - BTEQ SQL script

5 Populate the test system with appropriate test data imported from the production system for which testing is to be performed.

6 If your test suite requires, manipulate the captured statistics using the Teradata Manager tool STATs.

7 Depending on your requirements, perform one or both of the following steps:
   a Run the DIAGNOSTIC SET COSTS SQL statement (see SQL Data Manipulation Language) 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 693 for an illustration.
   b Run the DIAGNOSTIC SET SAMPLES SQL statement (see SQL Data Manipulation Language) 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 693 for a comparable illustration.
   c End of subprocedure.

8 End of procedure.
Target Level Emulation and Cost Profiles

You can use Target Level Emulation to obtain all necessary target system cost profile data for either Type 1 or Type 2 profiles. The general procedure is to use Teradata System Emulation Tool to save a copy of the $DBC.CostProfiles$ and $DBC.ConstantValues$ rows for the active cost profile on the target system, and then to install these as variable profiles on the emulation system (see “Procedure To Capture and Install Costing Information Including the Active Cost Profile” on page 691).

Target system cost profile information is associated with the TLE cost segment row in $SystemFE.Opt_Cost_Table$.

This information consists of the following data:

- Profile type.
- Profile name.
- Profile category.
- Any rows for that profile from the $DBC.ConstantValues$ table.

You can capture Profile identity information by submitting the following request:

```
DIAGNOSTIC DUMP COSTS targ-sys-name
```

The profile type, name, and category are stored in the cost table row. Teradata System Emulation Tool looks for this profile during export and, if found, captures the result of the following macro, which produces a set of SQL statements that create the actual cost profile:

```
EXEC DBC.ShowCostProfileAs('profile-name'
, 'targ-sys-name_Profile'
, ');
```

The Teradata System Emulation Tool Import function installs the cost segment row in the $SystemFE.Opt_Cost_Table$ table on the emulated system and then runs the sequence of SQL requests to create a variable cost profile named $targ-sys-name_Profile$. It then inserts the constant value rows collected from the target system.

The following diagnostic request installs the cost segment and activates $targ-sys-name_Profile$ at the specified emulation level:

```
DIAGNOSTIC SET COSTS targ-sys-name,
PROFILE targ-sys-name_Profile ON FOR level;
```

where $level$ is one of the following keywords:

- REQUEST
- IFP
- SESSION
- SYSTEM
Procedure To Capture and Install Costing Information Including the Active Cost Profile

There are two ways to capture Target Level Emulation data:

- Use the Teradata System Emulation Tool Export function (see Teradata System Emulation Tool User Guide).
- Use SQL DIAGNOSTIC statements (see SQL Data Manipulation Language).

The procedure defined here uses the second method.

1. Determine the name of the session profile:
   
   ```sql
   DIAGNOSTIC HELP PROFILE;
   ```
   
   Assume the request returns the name `prof-name` for the session profile.

2. Capture the cost segment and SQL to generate the emulation cost profile:
   
   ```sql
   DIAGNOSTIC DUMP COSTS Wellington;
   .EXPORT FILE=ProfileSQL
   EXEC DBC.ShowCostProfileAs ( 'prof-name' , 'Wellington_Profile' , 'Active cost profile for Wellington')
   .EXPORT reset
   ```

3. Load the file named `ProfileSQL`.
   
   To do this, execute the following installation command:
   
   ```sql
   .RUN FILE=ProfileSQL
   ```
   
   This installs `Wellington_Profile` as a variable Cost Profile.

4. Insert the cost segment row for Wellington into `SystemFE.Opt_Cost_Table` and his DBS Control and internal parameter rows into `SystemFE.Opt_DBSCtl_Table` using DIAGNOSTIC DUMP COSTS (see “Procedures to Capture Target System Optimizer Cost Values and Random AMP Statistical Samples for a Test Environment” on page 685).

5. Insert the internal parameter row for Wellington into `SystemFE.Opt_RAS_Table` using DIAGNOSTIC DUMP SAMPLES (see “Procedures to Capture Target System Optimizer Cost Values and Random AMP Statistical Samples for a Test Environment” on page 685).
6 Run the appropriate SQL DIAGNOSTIC requests to install the Target Level Emulation data:

```
DIAGNOSTIC SET COSTS Wellington
  , PROFILE Wellington_Profile
  ON FOR scope_level;
```

where `scope_level` indicates one of the following options:

- REQUEST
- SESSION
- IFP
- SYSTEM

See “DIAGNOSTIC SET COSTS” in SQL Data Manipulation Language for details.

```
DIAGNOSTIC SET SAMPLES Wellington
  ON FOR scope_level;
```

where `scope_level` indicates one of the following options:

- SESSION
- SYSTEM

See “DIAGNOSTIC SET SAMPLES” in SQL Data Manipulation Language for details.

7 This step is optional.

Remove the profile:

```
EXEC DBC.DeleteCostProfile ('Wellington_Profile');
```

8 End of procedure.
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 8: Target Level Emulation
Mapping Target System Files to a Test System Optimizer Table and GDO
This chapter describes how Teradata Database processes transactions.

Information about how the Archive/Recovery Utility processes archival and recovery transactions, which is often different in important ways, can be found in *Teradata Archive/Recovery Utility Reference*.
Database Transactions

Definition: Transaction

All processing in Teradata Database is transaction-based whether you realize it or not. The principle purpose of transaction management is to optimize concurrency: to ensure that as many sessions as possible can access the information in Teradata Database concurrently without compromising the consistency of the data.

Transaction management in relational database management systems has its origins in the work undertaken by Jim Gray and his colleagues in support of the development of the System R prototype RDBMS created within IBM Corporation in the 1970s. A summary of their work on the System R recovery management system is presented in Gray, et al. (1981).

Gray and Reuter (1993, page 7) make the following interesting analogy regarding database transactions: “The transaction concept is the computer equivalent to contract law. Imagine a society without contract law. That is what a computer system would be like without transactions. if nothing ever goes wrong, contracts are just overhead. But if something doesn’t quite work, the contract specifies how to clean up the situation”

A transaction is a sequence of \( n \) actions that preserve consistency irrespective of the actions taken. A database object is defined to be consistent when it satisfies all its propositions (see Database Design in the section that describes relation variables for a discussion of propositions, predicates, and related issues). Another way of saying this is that any two operations that relate to the same database object must appear to execute in some serial order (see “Serializability” on page 708). Integrity violations such as lost updates and inconsistent analyses do not satisfy the propositions of any database, and as a result leave it in an inconsistent state.

In the course of an update operation, a database object becomes transiently inconsistent as it undergoes a change to a new consistent state. Transactions were devised to enforce consistency in the face of potential transient inconsistency. Beside its common definitions as a unit of work and a unit of recovery, a transaction can also be seen as a unit of consistency (Gray et al., 1976), because it takes the database from one consistent state to another consistent state. All transient inconsistencies are isolated within the transaction and are never seen by the database.

As noted, a transaction is both a unit of work and a unit of recovery, though for the purposes of this chapter, it is viewed mostly as a unit of work (see Teradata Archive/Recovery Utility Reference for information about the Teradata approach to database 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. This is the so-called atomic property of transactions. 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 rollback operations to be performed on a database in order to change its current consistent state back to an earlier consistent state.
The system maintains the before-update copies of rows updated within a transaction in the Transient Journal (see “Transient Journal” on page 699). After-update images, as well as all of the following items, are contained within the file system Write Ahead Log, or WAL:

- Images of updates made to data blocks
- Images of updates made to cylinder indexes
- Images of updates made to File Information Blocks (FIBs)
- Instructions for where and how to use all these change images.

These WAL images are called redo records. After the system applies the appropriate set of WAL log redo records to the data on disk, then the data blocks, cylinder indexes, and FIB images appear as if the updated copies of those blocks that had really only been in memory, had actually been written to disk. In other words, the redo records apply their updates to older versions of those blocks.

The TJ records in the WAL log are undo records. After the system finishes processing the redo records, the data is in a consistent state, which permits the processing of the undo records.

During file system startup, and before the AMPs begin to come up, the file system handles any redo records in the WAL log that need to be processed. After that, the file system finishes its part of the startup process and the database software goes into normal recovery mode, where it processes any applicable TJ records in the same way they have always been processed, the only difference being that where in the past they were stored in the system table DBC.TransientJournal, they are now stored in the WAL log.

The following table briefly describes the conditions in which the system rolls back a transaction:

<table>
<thead>
<tr>
<th>Response Code Type</th>
<th>Session Mode</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>ANSI</td>
<td>Rolls back the error-generating request only. Does not release locks placed on behalf of the rolled back request.</td>
</tr>
<tr>
<td></td>
<td>Teradata</td>
<td>Not applicable²</td>
</tr>
<tr>
<td>Failure</td>
<td>ANSI</td>
<td>Rolls back the entire transaction that contains the error-generating request.</td>
</tr>
<tr>
<td></td>
<td>Teradata</td>
<td></td>
</tr>
</tbody>
</table>

². The system does not return Error codes in Teradata session mode.

Note that ANSI mode transactions are not always atomic because they do not roll back the entire transaction when an error response occurs, only the individual request that caused the error response. As a result, they do not support the A property of ACID transactions (see “The ACID Properties of Transactions” on page 698) in all circumstances.
To ensure that your transactions are always handled as intended, it is critical to code your applications with logic to handle any situations that only roll back an error-generating request rather than the entire transaction of which it is a member.

Also notice that the system does not release any locks placed for a request that is rolled back because of an Error response. All such locks remain in effect until the transaction either commits or rolls back.

In the case of system failures, you can use archived data (and the Before and After row images optionally recorded for data tables in the Permanent journal for their containing databases) to recover the database by either rolling back or rolling forward any number of previously performed actions. See *Teradata Archive/Recovery Utility Reference* for details.

### The ACID Properties of Transactions

The general concept of transaction processing is encapsulated by their so-called ACID properties. The term ACID was originally coined by Haerder and Reuter (1983). ACID is an acronym for the following set of properties that characterize any correct database transaction:

- Atomicity
- Consistency
- Isolation
- Durability

The specific meanings of these expressions in terms of database transactions are defined in the following table:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>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.</td>
</tr>
<tr>
<td>Consistency</td>
<td>A transaction transforms one consistent database state into another. Intermediate inconsistencies in the database are not permitted. Date (2003, 2004) argues that if database constraints are enforced properly, consistency is not an interesting property and, from a logical perspective, is trivial. Instead, Date contends, transaction managers should have the ultimate goal of enforcing database correctness. See “The Date critique of the use of consistency in the ACID initialism” on page 805 for a more complete statement of this position and its unenforceability.</td>
</tr>
<tr>
<td>Isolation</td>
<td>The operations of any transaction are concealed from all other transactions until that transaction commits. Isolation is a synonym for serializable (see “Serializability” on page 708).</td>
</tr>
<tr>
<td>Durability</td>
<td>Once a commit has been made, the new consistent state of the database survives even if the underlying system crashes. Durability is a synonym for persistent.</td>
</tr>
</tbody>
</table>
It should be clear that these four factors are not orthogonal, and the degree of shared variance among them varies considerably. As defined by Haerder and Reuter, for example, Atomicity and Consistency are very close to being subtle restatements of one another, and neither is possible without Isolation.

Furthermore, the importance of the various ACID guarantees depends on whether a transaction is read-only or if it also performs write operations. In the case of a read-only transaction, for example, Atomicity and Durability are irrelevant, but Isolation remains critically important (see “Levels of Isolation” on page 710 for why this is true).

Note that transactions are not always atomic in ANSI mode because when a request within a transaction fails with an Error response, only that request, not the entire transaction, is rolled back. The remainder of the transaction continues until it either commits or rolls back (see “Definition: Transaction” on page 696).

For more details about the ACID properties of database transactions, consult the following references: Bernstein and Newcomer (1997), Gray and Reuter (1993), Weikum and Vossen (2001).

**Transient Journal**

Because transactions sometimes fail and must be rolled back to a previous state, a mechanism must exist for preserving the before-change row set of a transaction. Preservation of the before-change row images for a transaction is the task of the Transient Journal, which is maintained automatically by the Write Ahead Logging component of the Teradata database management software. The system maintains a separate Transient Journal in the WAL log\(^1\) for each individual database transaction whether it runs in ANSI or Teradata session mode. Transient Journal rows and WAL records are interleaved within the same WAL log.

When a transaction fails for any reason, the system rolls back its updates by rolling forward the before-change copies of any rows touched by the failed transaction. It does this by writing the appropriate Transient Journal before-change rows over the updated after-change rows.

In other words, if a transaction failure occurs, the system acts as follows:

1. Processes WAL redo records (see “Definition: Transaction” on page 696) in order to complete the updates that must be performed before the Transient Journal undo records can be processed in step 2.

2. Retrieves the Transient Journal undo record images from the WAL log and applies them to the updates.
   
   To do this, it examines the transaction number and log record type field of each row to determine which rows to process, which to hide from the database management system, and which to return to the caller. This process continues until the system has completed the rollback operation.
   
   This has the effect of rolling back whatever updates the transaction had performed by writing the before-change images over those updates.

3. End of process.

---

1. Transient Journal records were written to the dictionary table DBC.TransientJournal in the past, but no longer are. Writing a Transient Journal record does not produce any other WAL log records.
Transaction Schedulers and Transaction Histories

A transaction scheduler is the software that controls the concurrent execution of interleaved transactions by restricting the order in which the various read, write, commit, and rollback operations of that interleaved set execute.

The purpose of transaction schedulers is to ensure that all transactions in the system are correct in the sense they are both serializable and recoverable (see “Serializability” on page 708). One of the principal tasks of a transaction scheduler is to avoid deadlocks. It does this by not permitting transactions that have conflicting data access requirements to run concurrently. In other words, the scheduler ensures that no two transactions lock the same database object in conflicting modes.

A transaction history is formed by interleaving the read and write operations of a set of transactions. Transaction histories are a model of what the transaction scheduler sees.

Concurrency is obviously a good thing in a multiuser environment, and interleaving the steps of transaction sets is an optimal way to achieve concurrency. By running transactions concurrently, it is possible to attain more optimal efficiencies. It makes no sense to execute just one transaction at a time, and there should always be another transaction ready to perform when a running transaction becomes blocked and enters an I/O wait state.

What is sometimes not fully understood is that the consistency of the database is even more desirable than attaining maximal concurrency. As a result, the operations undertaken by transaction interleavings must always be harmless to the consistency of the database.

A correct transaction history is a serializable transaction history (see “Serializability” on page 708), meaning that the read and write operations of a set of transactions can be reordered until the read and write operations of each transaction are together without affecting the values they read.

Serializability is said to be strict when transactions that are already in serial order in a history remain in the same relative order. For example, if transaction Tx1 writes before transaction Tx2 reads, then Tx1 must be serialized before Tx2. Another way of saying this is that the system must guarantee that all operations of any transaction in a history must have the same order as the actual transaction they model.

A complete transaction history is a sequence of operations that reflects the execution of multiple transactions, including a transaction terminating commit or rollback for each transaction in the history.

2. Serializable is used here in the broadest sense of the term. This does not mean that part of the job of a transaction scheduler is to terminate transaction sets running at non-serializable isolation levels if the system determines it is valid for them to do so.
Writing Transaction Histories Symbolically

It is often useful to write a transaction history in shorthand notation. Suppose you have the following sequence of operations occurring with two concurrently running transactions:

Assume the following sequence of actions for the concurrently running transactions Tx₁ and Tx₂:

1. Transaction 2 reads data item \( x \).
2. Transaction 2 writes a new value for data item \( x \).
3. Transaction 1 reads data item \( x \).
4. Transaction 1 reads data item \( y \).
5. Transaction 1 commits.
6. Transaction 2 reads data item \( y \).
7. Transaction 2 writes a new value for data item \( y \).
8. Transaction 2 commits.

This can be written in transaction history notation as follows:

\[ H: r_2(x) \ w_2(x) \ r_1(x) \ r_1(y) \ c_1 \ r_2(y) \ w_2(y) \ c_2 \]

where:

<table>
<thead>
<tr>
<th>Syntax element …</th>
<th>Specifies …</th>
</tr>
</thead>
<tbody>
<tr>
<td>H:</td>
<td>that the following set of symbols represent a transaction history</td>
</tr>
<tr>
<td>( r )</td>
<td>a read operation</td>
</tr>
<tr>
<td>( w )</td>
<td>a write operation</td>
</tr>
<tr>
<td>( (x) )</td>
<td>data item ( x )</td>
</tr>
<tr>
<td>( (y) )</td>
<td>data item ( y )</td>
</tr>
<tr>
<td>( c_n )</td>
<td>commit transaction ( n )</td>
</tr>
<tr>
<td>( r_n )</td>
<td>roll back transaction ( n )</td>
</tr>
</tbody>
</table>

Some of the later sections in this chapter use this syntax to notate transaction histories.
Transactions, Requests, and Statements

Introduction

Depending on the current session mode, any of the following definitions can represent a transaction:

- An individual SQL request.
  This is true only in Teradata session mode.
- A set of SQL requests of arbitrary length terminated by a COMMIT statement.
- A set of SQL requests of arbitrary length enclosed within explicit BEGIN TRANSACTION/END TRANSACTION boundaries.
  This is true only in Teradata session mode.
- A multistatement request.
- A macro.

The restrictions that apply to the items in this list are described when the items themselves are described.

Statement Processing

Statements are an SQL syntactic construct. They have the following properties:

- They require locks on the database objects they access in order to ensure serializability (see “Database Locks, Two-Phase Locking, and Serializability” on page 706).
- They can be submitted individually as requests.
  In this case, an SQL statement is functionally equivalent to a Teradata request.
  Depending on the circumstances, an SQL statement can also be functionally equivalent to a database transaction.
- They can be submitted as components of a multistatement request.
  Depending on the circumstances, a multistatement request can also be functionally equivalent to a database transaction.
- They can be submitted as components of a macro.
- They can be submitted as components of a stored procedure.

3. The validity of a multistatement request depends on the underlying API supporting the application that handles it, not the transaction semantics of the session in which it is submitted. Applications such as BTEQ, which use the CLIv2 API, handle multistatement requests correctly. Applications such as SQL Assistant, which uses the ODBC API, do not.

4. This is only true for applications that run under CLIv2. Applications like SQL Assistant, which runs under ODBC, run each request within a macro as an individual transaction.
Request Processing

A request is an input data stream composed of one or more SQL statements. Requests are a semantic concept and are the units that are processed by the Teradata PE.

The Teradata RDBMS architecture, and the PE in particular, is designed to process requests, not SQL statements per se. If a request contains more than one SQL statement, it is referred to as a multistatement request.

The following segments of system input analysis (see Chapter 1: “Request Parsing” for details) are all performed at the level of the request:

- SQL statement syntax checking (see “Syntaxer” on page 30).
- SQL statement syntax annotation (see “Resolver” on page 33).
- Access rights checking (see “Security Checking” on page 35).
- Request optimization (see “Optimizer” on page 59 and Chapter 2: “Query Rewrite and Optimization”).

The Lock Manager acquires the most restrictive locks required by the SQL statements in a request as early as possible. This is fundamental to the two-phase locking protocol (see “Database Locks, Two-Phase Locking, and Serializability” on page 706). Locks are never upgraded or released within a request.

For example, consider the following multistatement request:

```sql
SELECT *
FROM employee
;UPDATE employee
SET salary_amount = salary_amount * 1.1;
```

The SELECT statement in this request requires only a READ lock, but the UPDATE statement requires a WRITE lock. The WRITE lock is the most restrictive lock required by the request, so the system applies it to the request for both the READ and the UPDATE statements.

Multistatement Requests

A Teradata request terminates with a SEMICOLON character at the end of a line. You can think of this functionality as being equivalent to the ASCII EOT (End Of Transaction) character. A semicolon placed at any other point in the request does not terminate it.\(^5\) You can use this property to stack multiple SQL statements within a single request either by placing intermediate semicolons at the beginning of a subsequent line or in the middle of a line.

For example, both of the following requests are valid multistatement requests:

```sql
SELECT * FROM employee; UPDATE employee SET salary_amount=salary_amount * 1.1;

SELECT * FROM employee
;UPDATE employee SET salary_amount=salary_amount * 1.1;
```

---

\(^5\) This is not true for applications that use the ODBC API, such as SQL Assistant. ODBC-compliant applications do not recognize requests formatted in this manner as being multistatement requests.
Multistatement requests have the following properties:

- They can only include DML statements.
  This property distinguishes them from macros, which can contain a single DDL statement as long as it is the last statement in the request.
- The system performs the statements within a multistatement request in the order they are specified, with knowledge of dependencies.
- The most exclusive locks required by any individual statement within the request are held for the entire request.
- Like a transaction, the outcome of a multistatement request is all or nothing. If one statement in the request fails, the entire request fails and the system rolls it back.
- They can be committed either implicitly (Teradata mode only) or explicitly (using COMMIT in ANSI mode or END TRANSACTION in Teradata mode).

Transaction Processing

The most commonly used example of a transaction is a debit-credit transaction undertaken by means of a bank ATM. Suppose you withdraw 10 dollars from your checking account and deposit it in your savings account. This is a two-part transaction: withdrawal of money from the checking account (debit phase) and depositing it in the savings account (credit phase). Suppose the debit phase of the transaction completes successfully, but the credit phase does not. Do those 10 dollars just disappear? Without a proper transaction management system, they just might. In the scenario presented here, the transaction manager rolls back the withdrawal when the deposit fails so that no money is lost. This transaction is atomic because it is all-or-nothing. It cannot perform only part of its work.

Unlike ANSI session mode transaction semantics, the system automatically rolls back transactions in Teradata session mode whenever the system returns an error. You can also explicitly roll back the work performed by a transaction using either the ABORT or ROLLBACK statements.

Teradata session mode processing has the following advantages:

- You can move from implicit to explicit processing and back within the same script.
- You can nest explicit transactions within other explicit transactions.

There is no practical advantage to doing this, however, because failure of any nested transaction causes the entire transaction in which it is nested to be rolled back. Similarly, if the containing transaction fails for any reason, the results of any nested transactions it contains are also rolled back.

Keep in mind that each BEGIN TRANSACTION statement must have a matching END TRANSACTION statement, and that this type of processing can get very complicated very quickly.

---

6. This is an oversimplification. The classic debit-credit transaction has numerous components, but from a user perspective, it is a withdrawal of funds from one account and their deposit into another.
Statement and Request processing are essentially identical when operating in Teradata or ANSI modes, implicitly or explicitly. The conditions under which changes are applied is what differentiates the two modes from one another.

The operations of committing or rolling back changes to data are what constitute transaction processing.

**Hierarchical Relationship of Transactions, Requests, and SQL Statements**

The hierarchy of transactions, requests, and SQL statements is as follows:

1. A transaction contains one or more requests.
   - In the degenerate case, a transaction is a single-statement request.
2. A request contains one or more SQL statements.
   - The following table describes the specific details of these relationships:

<table>
<thead>
<tr>
<th>Submission Category</th>
<th>Minimum Number of Requests or SQL Statements per Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Teradata Mode Implicit Transaction</td>
</tr>
<tr>
<td>Request</td>
<td>1(^{a})</td>
</tr>
<tr>
<td>SQL Statement</td>
<td>1</td>
</tr>
</tbody>
</table>

- a. Each individual SQL statement and each multistatement request is treated as a single request. Note that ODBC-compliant applications do not support multistatement requests.
- b. This assumes a multistatement request. If the BEGIN TRANSACTION, END TRANSACTION, and interleaved SQL action statement are configured as separate SQL statements, then the minimum number of requests is 3. Note that ODBC-compliant applications do not support multistatement requests.
- c. The minimum statements are: BEGIN TRANSACTION, an SQL action statement, and END TRANSACTION.
- d. This assumes a multistatement request that includes a COMMIT or ROLLBACK statement as its last statement. If the request is not multistatement, then the minimum number of requests is 2: an SQL action statement followed by a COMMIT or ROLLBACK statement. Note that ODBC-compliant applications do not support multistatement requests.
- e. The minimum statements are: an SQL action statement followed by a COMMIT or ROLLBACK statement.
Database Locks, Two-Phase Locking, and Serializability

Database Locks

A lock is a device, usually implemented as a form of semaphore, that relational database management systems use to manage concurrent access to database objects by interleaved transactions running in multiple parallel sessions. Among the information contained in a lock is the identity of the database object it is locking, the identity of the transaction holding it, and its level and severity. The level and severity of a lock can be thought of as guarantees made to the transaction, assuring it that the objects for which it has requested locks are isolated from illegal interventions that might otherwise be made by other concurrently running transactions on those objects.

Locks cannot be placed on the individual partitions of an SLPPI or MLPPI table, only on either the entire table or on individual row hash values.

Two-Phase Locking

The solution to the illegal intervention problems is two-phase locking.7

The two-phase locking protocol (Eswaran et al., 1976) is the heart of modern transaction processing in relational database management systems. A locking protocol is defined as two-phase if it does not request additional locks for a transaction after it releases the locks it already holds.

This locking protocol is the foundation of serializability. Without two-phase locking, serializability cannot exist.

The two phases of 2PL are:

• The growing phase, during which locks on database objects are acquired.
• The shrinking phase, during which the previously acquired locks are released.

This is sometimes called the Two-Phase Rule:

<table>
<thead>
<tr>
<th>Phase Number</th>
<th>Action Taken By Transaction Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Take locks.</td>
</tr>
<tr>
<td>2</td>
<td>Release locks.</td>
</tr>
</tbody>
</table>

In practice, the shrinking phase is a point rather than an epoch; the system drops all locks held by a transaction at the moment that transaction commits or finishes rolling back.

7. Better put, two-phase locking is a solution to concurrency problems. Quoting Fekete et al. (2005, page 493), “Many database researchers think of concurrency control as a solved problem, since there exists a proven set of sufficient conditions for serializability. The problem is that those sufficient conditions can lead to concurrency control bottlenecks … [The issue of lower isolation level concurrency settings] … poses a new task for the theorist: to discover how to guarantee correctness at the least cost for such lower isolation levels.
Depending on the current session mode and the API supporting the application, the Lock Manager releases the locks held by a transaction at the point any of the following events occurs:

- A commit, whether implicit or explicitly specified.
  The following are all equivalent to a transaction commit:
  - The system encounters a SEMICOLON character at the end of a line of SQL text.
    Although the SQL language does not enforce any line-oriented structural rules, recall that Teradata Database recognizes only requests, not SQL statements per se, and a multistatement request terminates only when a SEMICOLON character is the last character in a line (see “Transactions, Requests, and Statements” on page 702).
  - The system encounters either of the following SQL statements:
    - COMMIT
    - END TRANSACTION
  - The system successfully performs a macro or stored procedure.
- A rollback or abort, whether implicit or explicitly specified.
  The following are both equivalent to a roll back or abort operation:
  - The system encounters either of the following SQL statements:
    - ROLLBACK
    - ABORT
  - The system aborts the transaction for reasons external to the transaction itself.

Unlike database locks, which the system releases automatically, HUT locks are released only when you explicitly specify the RELEASE LOCK option with your Archive/Recovery-related utility command or if your job successfully submits a RELEASE LOCK utility command after the Archive/Recovery-related operation has completed. See *Teradata Archive/Recovery Utility Reference* for details.

If a transaction must roll back, the 2PL protocol provides the advantage that the locks it holds on the objects whose updates must be undone need not be reacquired because, by definition, they are held until the transaction either commits or completes its rollback.

This protocol is formally referred to as strict two-phase locking, because the locks are released only when a transaction commits or rolls back. This action is always undertaken by the system because there are no SQL statements for releasing locks.

---

8. Whether the COMMIT or END TRANSACTION statement also requires a terminating SEMICOLON character depends on the API used by the application (see “Multistatement Requests” on page 703).

9. Whether the ROLLBACK or ABORT statement also requires a terminating SEMICOLON character depends on the API used by the application (see “Multistatement Requests” on page 703).
Computing methods tend to be optimizations rather than absolutes, and 2PL is no exception to that generalization. The principal concurrency problems that 2PL prevents are elaborated by five classic problems of transaction processing, usually named as follows:

- The lost update problem (see “The Lost Update Phenomenon” on page 713).
- The uncommitted dependency problem (see “The Uncommitted Dependencies (Dirty Read) Phenomenon” on page 715).
- The inconsistent analysis problem (see “The Inconsistent Analysis Phenomenon” on page 716).
- The dirty read problem (see “The Unrepeatable Read Phenomenon” on page 718).
- The deadlock problem (see “Deadlock” on page 753 and “Pseudo Table Locks” on page 750).

- Increased system overhead to administer locking.
- Decreased concurrency.

The impact of the lower concurrency that locks introduce is reduced greatly in a system like Teradata Database that supports multiple levels of locking granularity.

**Serializability**

The property of concurrent database accesses by transactions such that any arbitrary serial execution of those transactions preserves the integrity of the database is called serializability. The following definition of serializability is equivalent: although a given set of transactions executes concurrently, it appears to each transaction T in the set that the other member transactions executed either before T, or after T, but not both (paraphrased slightly from Gray and Reuter, 1993). Teradata Database ensures serializability as long as the current isolation level for the session is SERIALIZABLE (see “The ACID Properties of Transactions” on page 698 and the SET SESSION CHARACTERISTICS statement in SQL Data Definition Language for details).

For example, suppose table A is a checking accounts table and table B is a savings accounts table. Suppose one transaction, Tx₁, needs to move 400.00 USD from the checking account of a bank customer to the savings account of the same customer. Suppose another concurrently running transaction, Tx₂, performs a credit check on the same bank customer.

For the sake of illustration, assume the three states of the two accounts seen in the following table:

<table>
<thead>
<tr>
<th>State</th>
<th>Checking Account Amount</th>
<th>Savings Account Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$900.00</td>
<td>$100.00</td>
</tr>
<tr>
<td>2</td>
<td>$500.00</td>
<td>$100.00</td>
</tr>
<tr>
<td>3</td>
<td>$500.00</td>
<td>$500.00</td>
</tr>
</tbody>
</table>
State 1 depicts the initial state of the accounts.

State 2 depicts an intermediate condition.

Tx₁ has withdrawn $400.00 from the checking account, but has not yet deposited the funds in the savings account.

State 3 depicts the final state of the accounts.

Tx₁ has deposited the $400.00 withdrawn from the checking account into the savings account.

Without two-phase locking, Tx₂ can read the two accounts at state 2 and come to the conclusion that the customer balance is too low to justify permitting the purchase for which the credit check was determining the viability. But the reality is that this customer still has $1000 in the two accounts and should have qualified.

This is an example of the dirty read phenomenon (see “The Uncommitted Dependencies (Dirty Read) Phenomenon” on page 715).

The condition that 2PL ensures is serializability. When serializability is in force, the effect of these concurrently running transactions is the same as what would occur if they ran one after the other in series.

The two possibilities are as follows:

1. Tx₁ runs.
2. Tx₂ runs.

   In this scenario, Tx₂ only sees state 3, so the customer passes the credit check.

1. Tx₂ runs.
2. Tx₁ runs.

   In this scenario, Tx₂ only sees state 1, so the customer passes the credit check.

It makes no difference which scenario actually takes place, even in a distributed system, as long as the order is the same everywhere and both result in a consistent state for the database. The important thing to understand from this is that serializability ensures only consistent states for the database, not some particular ordering of transaction execution.

Two-phase locking of database objects (see “Two-Phase Locking” on page 706) is sufficient, but not necessary, to ensure serializability.

The term *serializable* is a synonym for the ACID property known as Isolation (see “The ACID Properties of Transactions” on page 698, “Levels of Isolation” on page 710, and the SET SESSION CHARACTERISTICS statement in *SQL Data Definition Language*).

Serializability describes a correct transaction schedule, meaning a schedule whose effect on the database is the same as that of some arbitrary serial schedule.
Levels of Isolation

The ANSI SQL-2003 standard defines isolation level as follows: “The isolation level of an SQL-transaction defines the degree to which the operations on SQL-data or schemas in that SQL-transaction are affected by the effects of and can affect operations on SQL-data or schemas in concurrent SQL-transactions.”\(^\text{10}\) Note that isolation level is a concept related to concurrently running transactions and how well their updates are protected from one another as a system processes their respective transactions.

Serializability defines transaction isolation. A transaction is either isolated from other concurrently running transactions or it is not. If you can achieve greater concurrency at the expense of imperfect isolation by using a lower isolation level, while at the same time being certain that you can avoid concurrency errors, then there is no reason not to run under that isolation level. The result is that you use CPU resources more effectively, while still guaranteeing serializable execution for the specific workload implemented in these transactions.

The ANSI SQL standard formalizes what it refers to as four isolation “levels” for transactions. To be precise, this section of the standard defines isolation (called SERIALIZABLE\(^\text{11}\)) and three weaker, non-serializable isolation levels that permit certain prohibited operation sequences to occur.

The standard collectively refers to these prohibited operation sequences as phenomena. Note that the ANSI isolation levels are defined in terms of these phenomena, not in terms of locking, even though all commercial RDBMSs implement transaction isolation using locks.\(^\text{12}\)

The three defined phenomena are dirty read, nonrepeatable read, and phantom read, respectively.

The non-serializable isolation levels ANSI defines are the following:

- Read Uncommitted
- Read Committed
- Repeatable Read


\(^{11}\) Date and Darwen (1997) point out that in the context this usage was first introduced into ANSI SQL, it was inappropriate because it applied only to single transactions, while serializability, as defined in the database management research literature, is a property of the interleaved execution of a set of concurrent transactions.

\(^{12}\) The O’Neils call this isolation level snapshot isolation. It does not exhibit any of the ANSI-defined phenomena, but is also not truly serializable (see Berenson et al., 1995).
The following table is taken from the ANSI SQL standard with slight modification. It specifies the phenomena that are or are not possible for each isolation level:

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Read Committed</td>
<td>Not possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
</tbody>
</table>

Teradata Database does not support the isolation levels READ COMMITTED and REPEATABLE READ. READ UNCOMMITTED is implemented using an ACCESS level lock (see “ACCESS” on page 725).

Berenson et al. (1995) have published a thorough criticism of this scheme, including the fact that it permits a “dirty write,” though it does not even mention this possibility. Furthermore, their paper notes that in spite of the fact that the intent of developing these isolation levels was to define them in an implementation-free manner, the definitions rely entirely on known locking behaviors.

Another problem with the ANSI isolation level definitions is that while the first three levels are defined in terms of phenomena, the SERIALIZABLE level also adds the following condition: not only must a transaction history not allow the three phenomena, but it must also be serializable. This is not insignificant, because, as Berenson et al. note, the O’Neils had discovered a new form of isolation that does not have any of the ANSI phenomena problems, but which is also not serializable.  

The authors have formalized a reinterpretation of the language in the standard in an attempt to express what they think its authors really meant, though later thought suggests this is an unachievable task. O’Neil (2004) notes that there seem to be an “infinite number of Isolation Levels, and this demonstrates there is no finite set of phenomena that can characterize them all.”

He reaches this conclusion after having spent years working on the problem and after having thought he had correctly redefined the ANSI isolation levels in such a way that not only were the previously noted problems with the definitions remedied, but the levels themselves had been defined in such a way that they were no longer tied to locking implementations, but could also be implemented in terms of optimistic and multiversion concurrency control schemes (Adya et al., 2000).

13. This new form of isolation is commonly referred to as Snapshot Isolation.
Changing the Transaction Isolation Level for Read-Only Operations

Sometimes you might be willing to give up a level of transaction isolation insurance in return for better performance. While this makes no sense for operations that write data, it can sometimes make sense to permit dirty read operations, particularly if you are only interested in gaining a general impression of some aspect of the data rather than obtaining consistent, reliable, repeatable results. The mechanism for this is to downgrade the default lock for read-only operations from a severity of READ to a severity of ACCESS.

Teradata Database provides methods for allowing the possibility of dirty reads at two different levels: the individual request and the session.

<table>
<thead>
<tr>
<th>TO set the default read-only locking severity for this level ...</th>
<th>USE this method ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>individual request</td>
<td>LOCKING request modifier. See SQL Data Manipulation Language for details of the syntax and usage of this statement modifier.</td>
</tr>
<tr>
<td>session</td>
<td>SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement. See SQL Data Definition Language for details of the syntax and usage of this statement.</td>
</tr>
</tbody>
</table>

Note that the global application of ACCESS locking for read operations when SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is set to READ UNCOMMITTED depends on the setting of the DBS Control flag AccessLockForUncomRead.

When the flag is set FALSE, SELECT operations within INSERT, DELETE, MERGE, and UPDATE requests set READ locks, while when the flag is set TRUE, the same SELECT operations set ACCESS locks. See “SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL” in SQL Data Definition Language and Utilities for details.

14. This is a very important consideration, and it should not be taken lightly. The overall qualitative workload of the session must be examined carefully before making the determination of whether to default to ACCESS-level locking for read-only operations or not. For example, consider a session in which a MultiLoad import job is running. Because of the way MultiLoad updates table rows during its acquisition phase (see Teradata MultiLoad Reference), using ACCESS locks to query the target table of the MultiLoad job during an acquisition phase can produce extremely inaccurate result sets. In this case, the results probably would not provide even a reasonable impression of the table data.
Canonical Concurrency Problems

Introduction

Without two-phase locking (2PL) and serializability of interleaved transactions, there are a number of classic data integrity phenomena, sometimes referred to as canonical concurrency problems, that can emerge. Four of the most frequently mentioned classic phenomena are the following:

- The lost update phenomenon.
- The uncommitted dependencies phenomenon.
- The inconsistent analysis phenomenon.
- The unrepeatable read phenomenon.

When full transaction isolation is in force (see “Levels of Isolation” on page 710), these phenomena cannot occur. Each of these phenomena is described in some detail in the following pages.

The Lost Update Phenomenon

The lost update phenomenon describes a situation in which two transactions update the same field value in a row before either transaction commits. As a result, because the second transaction updates the field based on its original value rather than its updated, but uncommitted, value as changed by the first transaction, the update made by the first transaction is never seen, and the result is an inconsistent database.

Note that this result cannot be achieved if the transaction history is serializable because the first transaction would have locked the object being updated, thus preventing simultaneous updating of the object by any other concurrently running transaction. The second transaction could only see the object after it had been updated by the first, so that update is not lost to the system, and can be replicated by any arbitrary serial execution of those same transactions.

The consistency of the database is not affected by whether transaction 1 runs first or transaction 2 runs first. The critical point is that any updates must be properly committed before they can be updated again. In other words, it makes no difference to the consistency of the database whether transaction 1 updates a value from 10 to 20 and then transaction 2 updates the same value from 20 to 10 (or vice versa) as long as each transaction is properly committed.

In transaction history notation, this partial history is written as follows:

\[ H: r_1(x) \ r_2(x)w_1(x)w_2(x) \]

\( Tx_2 \) has written over the update of \( x \) made by \( Tx_1 \) without either transaction having committed. As a result, the update \( w_1(x) \) is lost.

---

15. Partial because the history does not include either a commit or rollback operation.
In the following graphic, the first transaction is called Tx1 and the second is called Tx2.

<table>
<thead>
<tr>
<th>Tx1</th>
<th>Time</th>
<th>Tx2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read x</td>
<td>t1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read x</td>
</tr>
<tr>
<td></td>
<td>t2</td>
<td></td>
</tr>
<tr>
<td>Write x</td>
<td>t3</td>
<td>Write x</td>
</tr>
<tr>
<td></td>
<td>t4</td>
<td></td>
</tr>
</tbody>
</table>

The stages in the transaction history are as follows:

1. At time t₁, Tx₁ retrieves row₁.
   The value of the field in question is value₀.
2. At time t₂, Tx₂ also retrieves row₁.
   The value of the field at this time is still value₀.
3. At time t₃, Tx₁ updates the field value in row₁.
   The value of the field is now value₁.
4. At time t₄, Tx₂ updates the same field value in row₁.
   The value of the field is now value₂.

The problem with this result is that the update was based on the initial field value of value₀ rather than the correct, updated value, which is value₁. Tx₂ did not see value₁ before writing value₂ over it.
The Uncommitted Dependencies (Dirty Read) Phenomenon

The uncommitted dependency phenomenon, also commonly called the dirty read problem, describes a situation in which one transaction is permitted to retrieve, or even update, a value that has been updated by a concurrently running second transaction that has not yet committed. The problem arises because the second transaction might never commit. If the second transaction were to roll back, or be rolled back by the system for some reason external to the transaction itself, the database would be left in an inconsistent state.

The first example demonstrates the case where the first transaction reads a value that has been updated by a second transaction, and the second transaction then rolls back. As a result, the uncommitted update that occurred at time $t_1$ is never actually seen by the system, so as far as the database is concerned, it never occurred. The danger in this is that the first transaction can use the phantom update value to produce misleading reports or, worse, update the phantom update value.

In transaction history notation, this history is written as follows:

$$H: w_2(x)r_1(x)r_2$$

In the following graphic, the first transaction is named $Tx_1$ and the second is named $Tx_2$.

<table>
<thead>
<tr>
<th>$Tx_1$</th>
<th>$Tx_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>Write x</td>
</tr>
<tr>
<td>Read $x$</td>
<td>$t_2$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>Rollback $Tx_2$</td>
</tr>
</tbody>
</table>

The stages in the transaction history are as follows:

1. At time $t_1$, $Tx_2$ updates $value_0$ to $value_1$ in row $1$.
2. At time $t_2$, $Tx_1$ retrieves row $1$ where the value is now $value_1$.
3. At time $t_3$, $Tx_2$ rolls back, setting the field value back to $value_0$.

Meanwhile, $Tx_1$ continues processing, using the phantom update value $value_1$ instead of the correct value, which is $value_0$.

Any reports generated by $Tx_1$, or updates $Tx_1$ does to this column after $t_3$ is based on $value_1$ which, because of the rollback operation on $Tx_2$, never existed as far as the database is concerned. As a result, the state of the database is not consistent.

16. This action is referred to as an uncommitted update.
The second example combines an uncommitted dependency with a lost update. The uncommitted dependency occurs when Tx₁ updates value₁ to value₂ after Tx₂ had updated value₀ to value₁, but without committing. Tx₁ becomes dependent on Tx₂ committing, but because Tx₂ instead rolls back at time t₃, setting the dependent update value back to value₀, the update operation made by Tx₁ is not only not valid, but also lost. Either outcome leaves the database in an inconsistent state.

In transaction history notation, this history is written as follows:

\[ H : w₂(x)w₁(x)r₂ \]

In the following graphic, the first transaction is named Tx₁ and the second is named Tx₂.

1. At time t₁, Tx₂ updates a field in row₁ from value₀ to value₁.
2. At time t₂, Tx₁ updates the same field in row₁ from value₁ to value₂.
3. At time t₃, Tx₂ rolls back, setting the field value back to value₀.

The Inconsistent Analysis Phenomenon

In the following scenario, one transaction is reporting the quantity of a particular part P₁01 on hand in three different warehouses. A different, interleaved transaction is simultaneously updating the respective quantities of that part in the three warehouses to reflect the fact that a quantity of the part had been shipped from warehouse 3, which had an overstock, to warehouse 1, which was nearly out of the part. To make the calculation demonstrate the inconsistent analysis problem more clearly, a quantity of the part from a third warehouse, warehouse 2, is also introduced.

The initial quantities of the part in each of the three warehouses at time t₀ are as follows:

<table>
<thead>
<tr>
<th>Warehouse 1</th>
<th>Warehouse 2</th>
<th>Warehouse 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qty_P₁₀₁</td>
<td>Qty_P₁₀₁</td>
<td>Qty_P₁₀₁</td>
</tr>
<tr>
<td>30</td>
<td>750</td>
<td>1000</td>
</tr>
</tbody>
</table>
Assume that no parts are shipped or deleted during the course of the example transactions. In transaction history notation, this partial\(^{17} \) history is written as follows:

\[ H: r_1(x - 40) r_1(y - 50) r_2(z - 30) w_2(z - 20) r_2(x - 40) w_2(x - 50) c r_1(z - 20) \]

In the following graphic, the first transaction is named Tx\(_1\) and the second is named Tx\(_2\).

<table>
<thead>
<tr>
<th>Tx(_1)</th>
<th>Time</th>
<th>Tx(_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (x)</td>
<td>(t_1)</td>
<td></td>
</tr>
<tr>
<td>(x = 30)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM = 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read (y)</td>
<td>(t_2)</td>
<td></td>
</tr>
<tr>
<td>(y = 750)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM = 780</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read (z)</td>
<td>(t_3)</td>
<td>Read (z)</td>
</tr>
<tr>
<td>(z = 1000)</td>
<td></td>
<td>(z = 750)</td>
</tr>
<tr>
<td>Read (x)</td>
<td>(t_4)</td>
<td>Read (x)</td>
</tr>
<tr>
<td>(x = 30)</td>
<td></td>
<td>(x = 30)</td>
</tr>
<tr>
<td>Write (x)</td>
<td>(t_5)</td>
<td>Write (x)</td>
</tr>
<tr>
<td>(x = 280)</td>
<td></td>
<td>(x = 280)</td>
</tr>
<tr>
<td>Commit Tx(_2)</td>
<td>(t_6)</td>
<td>Commit Tx(_2)</td>
</tr>
<tr>
<td>Read (z)</td>
<td>(t_7)</td>
<td></td>
</tr>
<tr>
<td>(z = 750)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM = 1530</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM should be 1780</td>
<td>(t_8)</td>
<td></td>
</tr>
</tbody>
</table>

The stages in the transaction history are as follows:

1. At time \(t_1\), Tx\(_1\) retrieves row\(_{WH1}\), where the value of \(P_{101}\) is 30, and sets the value of SUM to 30.
2. At time \(t_2\), Tx\(_1\) retrieves row\(_{WH2}\), where the value of \(P_{101}\) is 750, and increments the value of SUM to 780.
3. At time \(t_3\), Tx\(_3\) retrieves row\(_{WH3}\), where the value of \(P_{101}\) is 1000.
4. At time \(t_4\), Tx\(_2\) updates row\(_{WH3}\), decrementing the value of \(P_{101}\) from 1000 to 750.
5. At time \(t_5\), Tx\(_2\) retrieves row\(_{WH1}\), where the value of \(P_{101}\) is still 30.
6. At time \(t_6\), Tx\(_2\) updates row\(_{WH1}\), incrementing the value of \(P_{101}\) from 30 to 280.

17. Partial because the history does not specify either a commit or a rollback of Transaction 1.
At time $t_7$, $Tx_2$ commits.

According to the database, the quantities of P101 in each warehouse are 280, 750, and 750 for warehouse 1, warehouse 2, and warehouse 3, respectively.

At time $t_8$, $Tx_1$ retrieves row $WH_3$, which now has the P101 value 750.

When $Tx_1$ increments the value of SUM, the value is 1530, but the actual sum across all warehouses is the same it was at time $t_0$, or 1780, because no parts were shipped or deleted during the course of these two transactions.

**The Unrepeatable Read Phenomenon**

Depending on the application, the unrepeatable read phenomenon might not be a concern. In fact, the Teradata ACCESS-level locking severity is specifically designed to permit unrepeatable reads.

An unrepeatable read occurs when a transaction reads a row that is then updated by another concurrently running transaction. If the first transaction re-reads the row in question, it sees yet another value for the field in the same row without having itself updated it.

In transaction history notation, this partial\(^{18}\) history is written as follows:

$$H: r_1(x)w_2(x)r_1(x)$$

In the following graphic, the first transaction is named $Tx_1$ and the second is named $Tx_2$.

1. At time $t_1$, $Tx_1$ retrieves row $1$.
   $Tx_1$ reads a particular field in row $1$ and sees value $0$.

2. At time $t_2$, $Tx_2$ retrieves row $1$.
   $Tx_2$ updates the field previously read by $Tx_1$ and updates its value to value $1$.

3. At time $t_3$, $Tx_1$ again retrieves row $1$.
   $Tx_1$ reads the updated field and sees value $1$ instead of the expected value $0$.

---

\(^{18}\) Partial because no commit or rollback is specified for either transaction.
Lock Manager

Introduction

Any number of users and applications can simultaneously access data stored in a Teradata database.

The Lock Manager imposes concurrency control by locking the database object being accessed by each transaction and releasing those locks when the transaction either commits or rolls back its work. This control ensures that the data remains consistent for all users. Note that with the exception of pseudo table locks, locks in Teradata Database are not managed globally, but by each AMP individually.

While the Lock Manager imposes locks automatically, you can upgrade locks explicitly by using the SQL LOCKING request modifier (see SQL Data Manipulation Language).

Locking Levels

The Lock Manager implicitly locks an object at the following levels:

<table>
<thead>
<tr>
<th>This lock level …</th>
<th>Locks …</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most Restrictive</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td>all rows of all tables in the database and their associated secondary index subtables.</td>
</tr>
<tr>
<td>Table</td>
<td>all rows in the base table and in any secondary index and fallback subtables associated with it.</td>
</tr>
<tr>
<td>View</td>
<td>all underlying tables accessed by the view.</td>
</tr>
<tr>
<td>Row Hash</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>• A row hash lock is applied whenever a table is accessed using a unique primary index (UPI).</td>
</tr>
<tr>
<td></td>
<td>• For an update that uses a unique secondary index (USI), the appropriate row of the secondary index subtable is also locked.</td>
</tr>
<tr>
<td></td>
<td>• It is not necessary to lock the fallback copy of the row, nor any associated row, of a nonunique secondary index (NUSI) subtable.</td>
</tr>
<tr>
<td>Least Restrictive</td>
<td></td>
</tr>
</tbody>
</table>
Locking Considerations

Teradata Database always makes an effort to lock database objects at the least restrictive level and severity possible to ensure database integrity while at the same time maintaining maximum concurrency.\(^{19}\) 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.

If an application begins a transaction and then performs a large number of single-row updates without closing that transaction, one or more AMP lock tables eventually fills to capacity.\(^{20}\) Such a lock table overflow also affects other transactions because no request can begin without first acquiring one or more locks on its underlying database objects. After awhile, transactions that are unable to acquire any locks abort, and can even cause the system to crash.

You can control the maximum number of row hash locks that the Lock Manager allows for each transaction using the DBS Control flag \text{MaxRowHashBlocksPercent}\ (see \textit{Utilities} for details). The default for this flag is 50 percent of the total number of control blocks that an AMP lock table can support. You can set the value for \text{MaxRowHashBlocksPercent} to as large a percentage as you need to accommodate the number of row hash locks required for your system workloads.

Teradata Database automatically aborts a transaction when the number of row hash locks it acquires exceeds the threshold defined by \text{MaxRowHashBlocksPercent}. No other transactions are affected when this occurs. This control over the number of row hash locks held by a transaction applies only to ANSI session mode and explicit Teradata session mode transactions. Implicit Teradata mode transactions are not affected because each implicit Teradata mode request is a transaction in itself, and the system releases its locks immediately after it commits or rolls back.

\(^{19}\) This is not strictly true for HUT locks, where the same locking levels and severities are always used for a given Archive/Recovery utility command. That does not mean the locking levels and severities for HUT locks are not optimal, but rather that the optimum locks for those operations are predetermined.

\(^{20}\) This typically occurs for, but is not limited to, transactions in ANSI session mode.
You can review all the active locks and determine which other user locks are blocking your transactions using the Lock Display (lokdisp) or Locking Logger (dumplocklog) utilities (see Utilities).

**Releasing Locks**

The Lock Manager releases all locks held by a transaction under the following conditions:

- An implicit transaction commits (Teradata session mode).
  
  All ANSI mode transactions are explicit.
- A two-phase commit transaction commits (Teradata session mode).
  
  2PC is not valid in ANSI mode.
- An explicit transaction commits by issuing its outermost END TRANSACTION statement (Teradata session mode).
  
  Explicit transactions are not valid in ANSI mode.
- A transaction issues a COMMIT statement (ANSI session mode).
  
  The COMMIT statement is not valid in Teradata mode.
- A transaction issues a ROLLBACK or ABORT statement (all session modes)

Unlike the case for database transaction locks, you must release HUT locks explicitly. This is done either by specifying the RELEASE LOCK option as part of a given Archive/Recovery-related utility command or by issuing a separate RELEASE LOCK command in your job script that is placed sequentially to execute following the completion of the command that set those HUT locks.

Unreleased HUT locks persist after job completion, user logoff, and even after system restarts, so you must be careful to ensure that any HUT locks set by an Archive/Recovery-related job are explicitly released after that job completes.

See the descriptions of the various Archive/Recovery utility commands in *Teradata Archive/Recovery Utility Reference* for details.
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 session 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 session mode.
  - COMMIT statement in ANSI mode.
- The transaction aborts.
- The system restarts.

21. The exception to this is the WRITE locks placed by a SELECT AND CONSUME operation on a queue table. The system grants WRITE locks on a queue table only when one or more rows exist in the table. The system does not grant locks at the time it receives the SELECT AND CONSUME request.
Teradata Database Locking Levels and Severities

Introduction

Teradata Database locks have two orthogonal dimensions: level and severity.\(^{22}\) The level of a lock refers to its scope or granularity: the type and, by inference, the size of the object locked. For example, a database lock is a higher, less finely grained level lock than a row hash lock,\(^{23}\) which operates at a lower level and finer granularity.

The selection of lock granularity is always a tradeoff between the conflicting demands of concurrency and overhead. For example, concurrency increases as the choice of locking level becomes increasingly granular. Exerting a row-hash level lock permits more users to access a given table than exerting a table-level lock on the same table. This is why the system provides multiple levels of locking granularity. See “Locking Levels” on page 723 and “Lock Manager” on page 719 for a description of locking levels.

The severity of a lock refers to its degree of restrictiveness or exclusivity, such as WRITE lock being more restrictive than an ACCESS lock, or an EXCLUSIVE lock being more restrictive than a READ lock. See “Locking Severity and Relative Restrictiveness” on page 724 and “Default Lock Assignments and Lock Upgradeability” on page 739 for more information about locking severities.

Locking Levels

The hierarchy of locking levels for a database management system is a function of the available granularities of locking, with database-level locks having the lowest (coarsest) granularity and row hash-level locks having the highest (finest) granularity. Depending on the request being processed, the system places a certain default lock level on the object of the request, which can be one of the following database objects:

- Database
- Table\(^{24}\)
- 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 719) combine to exert various locking granularities. The less granular the combination, the greater the impact on concurrency and system performance, and the greater the delay in processing time.

---

22. There is a small, but unimportant, correlation between locking levels and locking severity. For example, it makes no sense to apply a row hash-level lock with EXCLUSIVE severity because the row hash level is atomic and so cannot be shared.

23. Note that Teradata Database locks rows at the level of row hash, not the individual row, which means that a row hash-level lock typically locks more than one row.

24. See “Pseudo Table Locks” on page 750 for a description of a special category of table-level locking.
**Locking Severity and Relative Restrictiveness**

A locking scheme that provides multiple levels of locking severity, such as the scheme implemented by Teradata Database, is said to be multigranular. Eswaran et al. (1976) introduced the concept of multigranular locking to provide an optimal method of ensuring database consistency while simultaneously ensuring the highest possible concurrency.

The available lock severities, from most restrictive to least restrictive, are as follows:

<table>
<thead>
<tr>
<th>Lock Severity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Most Restrictive</strong></td>
<td></td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td>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, copied, rolled back, rolled forward, or built by an Archive/Recovery utility command, in which case a HUT EXCLUSIVE lock is placed on the resource. An EXCLUSIVE lock restricts access to the object by any other user. You can also place this lock explicitly using the LOCKING request modifier (see SQL Data Manipulation Language).</td>
</tr>
<tr>
<td>HUT EXCLUSIVE</td>
<td></td>
</tr>
<tr>
<td>WRITE</td>
<td>Placed in response to an INSERT, UPDATE, or DELETE request or when a set of database objects is being recovered or restored by an Archive/Recovery utility command. A WRITE lock restricts access by other users (except for applications that are not concerned with data consistency and choose to override the automatically applied WRITE lock by specifying a less restrictive ACCESS lock). You can also place this lock explicitly using the LOCKING request modifier (see SQL Data Manipulation Language).</td>
</tr>
<tr>
<td>HUT WRITE</td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>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 LOCKING request modifier (see SQL Data Manipulation Language). Archive requests can also place HUT READ and HUT GROUP READ locks on database resources. You must release any HUT READ or HUT GROUP READ locks you set either by submitting the RELEASE LOCK Archive utility command after the command that set those locks completes, or by specifying the RELEASE LOCK option in the command text itself. See Teradata Archive/Recovery Utility Reference for syntax and usage details.</td>
</tr>
<tr>
<td>SHAREa</td>
<td></td>
</tr>
<tr>
<td>HUT READ</td>
<td></td>
</tr>
<tr>
<td>HUT GROUP READ</td>
<td></td>
</tr>
</tbody>
</table>
The CHECKSUM, ACCESS, and HUT ACCESS locking severities are all at the same level in the restrictiveness hierarchy.

### CHECKSUM
- Placed in response to a user-specified LOCKING FOR CHECKSUM modifier (see SQL Data Manipulation Language) 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 772, SQL Data Manipulation Language, and SQL Stored Procedures and Embedded SQL.

### ACCESS
- Placed in response to a user-defined LOCKING FOR ACCESS modifier (see SQL Data Manipulation Language), or by setting the session default isolation level to READ UNCOMMITTED using the SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement (see SQL Data Definition Language).
- Permits a user to have READ access to an object that might already be locked for READ or WRITE. An ACCESS lock does not restrict access by another user except when an EXCLUSIVE lock is required; therefore it is sometimes referred to as a dirty READ lockb.
- A user requesting an ACCESS lock 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.
- Note that the GROUP READ lock set by the Archive utility also places a HUT ACCESS lock internally on its subject table for the duration of the GROUP READ lock.
- See Teradata Archive/Recovery Utility Reference for details.

### Least Restrictive

- SHARE is a deprecated synonym for READ.
- Note that the global application of ACCESS locking for read operations when SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is set to READ UNCOMMITTED depends on the setting of the DBS Control flag AccessLockForUncomRead.

When the flag is set FALSE, SELECT operations within INSERT, DELETE, MERGE, and UPDATE requests set READ locks, while when the flag is set TRUE, the same SELECT operations set ACCESS locks.
Compatibility Among Locking Severities

No locking severity is compatible with all other locking severities. There are two basic types of locking severity for any relational database management system: Read and Write. The transaction literature refers to these two basic types as Access locking and Exclusive locking, though the terms are not defined in the same way as they are used in Teradata Database.

Read locks always conflict with Write locks, while Write locks always conflict with other Write locks.

The Teradata Lock Manager controls three types of Read lock:

- ACCESS (see “ACCESS” on page 725).
- CHECKSUM (see “CHECKSUM” on page 725).
- READ (see “READ” on page 724).

The Teradata Lock Manager also controls two types of Write lock:

- WRITE (see “WRITE” on page 724).
- EXCLUSIVE (see “EXCLUSIVE” on page 724).

Various commands of the Archive/Recovery utility use three types of Read lock:

- HUT ACCESS
- HUT READ
- HUT GROUP READ

Various commands of the Archive/Recovery utility use only one type of Write lock:

- HUT EXCLUSIVE

See Teradata Archive/Recovery Utility Reference for descriptions of the various HUT locks.

The following table summarizes the compatibilities and incompatibilities among the various locking severities used by the Teradata Lock Manager. Note that the system enforces the identical compatibilities, where relevant, for HUT locks.
<table>
<thead>
<tr>
<th>This locking severity ...</th>
<th>Is compatible with this locking severity ...</th>
<th>But is not compatible with this locking severity ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ACCESS</td>
<td>• ACCESS</td>
<td>• EXCLUSIVE</td>
</tr>
<tr>
<td>• HUT ACCESS</td>
<td>• HUT ACCESS</td>
<td>• HUT EXCLUSIVE</td>
</tr>
<tr>
<td>• CHECKSUM</td>
<td>• CHECKSUM</td>
<td></td>
</tr>
<tr>
<td>• READ</td>
<td>• ACCESS</td>
<td>• WRITE</td>
</tr>
<tr>
<td>• SHARE\textsuperscript{a}</td>
<td>• HUT ACCESS</td>
<td>• EXCLUSIVE</td>
</tr>
<tr>
<td>• HUT READ</td>
<td>• CHECKSUM</td>
<td>• HUT EXCLUSIVE</td>
</tr>
<tr>
<td>• HUT GROUP READ</td>
<td>• READ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SHARE\textsuperscript{a}</td>
<td>• HUT EXCLUSIVE</td>
</tr>
<tr>
<td></td>
<td>• HUT READ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• HUT GROUP READ</td>
<td></td>
</tr>
</tbody>
</table>

\textsuperscript{a} SHARE is a deprecated synonym for READ.
The following notation is used to describe the locking severity compatibilities:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q$</td>
<td>A locking queue associated with a database object.</td>
</tr>
<tr>
<td>$L$</td>
<td>A list of locks currently held.</td>
</tr>
<tr>
<td><code>lock(&lt;object&gt;,&lt;lock requested&gt;)</code></td>
<td>A database object-locking severity requested pair.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>IF <code>lock(&lt;object&gt;,&lt;lock requested&gt;)</code> is ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>queued for any lock in $L$</td>
<td>the transaction is placed in $Q$ and waits there as long as <code>lock(&lt;object&gt;,&lt;lock requested&gt;)</code> is queued. A request in this state is said to be blocked (see “Blocked Requests” on page 746).</td>
</tr>
<tr>
<td>granted</td>
<td><code>lock(&lt;object&gt;,&lt;lock requested&gt;)</code> is added to $L$ with <code>&lt;lock requested&gt;</code> and the transaction resumes processing.</td>
</tr>
</tbody>
</table>

After the transaction finishes with an object by either committing or rolling back, its lock is removed from $L$.

The table on the following page summarizes the action taken when a requested locking severity competes with an existing locking severity. Note that the system enforces the identical compatibilities with other locks for HUT ACCESS, HUT READ, HUT GROUP READ, and HUT EXCLUSIVE locks as it does for the comparably named database locks.
<table>
<thead>
<tr>
<th>Severity of Requested Lock</th>
<th>None</th>
<th>ACCESS</th>
<th>READ</th>
<th>WRITE</th>
<th>EXCLUSIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>HUT ACCESS</td>
<td>HUT READ</td>
<td>HUT GROUP READ</td>
<td>HUT EXCLUSIVE</td>
</tr>
<tr>
<td>ACCESS</td>
<td></td>
<td>Lock Granted</td>
<td>Lock Granted</td>
<td>Lock Granted</td>
<td>Request Queued&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>HUT ACCESS</td>
<td></td>
<td>Lock Granted</td>
<td>Lock Granted</td>
<td>Request Queued&lt;sup&gt;b&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHECKSUM</td>
<td></td>
<td>Lock Granted</td>
<td>Request Queued&lt;sup&gt;b&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td></td>
<td>Lock Granted</td>
<td>Request Queued&lt;sup&gt;b&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HUT READ</td>
<td></td>
<td>Request Queued&lt;sup&gt;b&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HUT GROUP READ&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
<td>Request Queued&lt;sup&gt;b&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE</td>
<td></td>
<td>Request Queued&lt;sup&gt;b&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td></td>
<td>Request Queued&lt;sup&gt;b&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HUT EXCLUSIVE</td>
<td></td>
<td>Request Queued&lt;sup&gt;b&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

<sup>a</sup> HUT GROUP READ locks are implemented internally as RANGE READ locks. This lock place a HUT ACCESS lock on the subject table for the duration of the Read operation and a series of rolling HUT READ locks on the component row sets of the table as rows are read in sequence from a set of data blocks ranging between system-determined starting and ending rowhash values. See “HUT Lock Severities” on page 733.

<sup>b</sup> If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.
Because other client utilities such as BTEQ, FastExport, FastLoad, MultiLoad, Teradata Parallel Data Pump, and Teradata Parallel Transporter use standard database locks, the interactions of those locking severities with those of other database locks are identical. See the following manuals for details of the locks set by those utilities:

- Basic Teradata Query Reference
- Teradata FastExport Reference
- Teradata FastLoad Reference
- Teradata MultiLoad Reference
- Teradata Parallel Data Pump Reference
- Teradata Parallel Transporter Reference

A queued request is in an I/O wait state and is said to be blocked (see “Blocked Requests” on page 746).

A WRITE or EXCLUSIVE lock on a database, table, or view restricts all requests or transactions except the one holding the lock from accessing data within the domain of that object.

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.

Table-level WRITE locks on dictionary tables prevent contending tasks from accessing the dictionary, so the Lock Manager attempts to lock dictionary tables at the row hash level whenever possible.

**The NOWAIT Option for the SQL LOCKING Request Modifier**

When you specify the NOWAIT option for the SQL LOCKING request modifier, the system aborts a transaction that makes a lock request that cannot be fulfilled immediately. For details on how to use the NOWAIT option with the LOCKING request modifier, see *SQL Data Manipulation Language*.

The system uses a slight variation of this code internally to avoid blocking tactical queries while DDL operations are occurring. Instead of aborting the request, the system instead downgrades its lock severities from READ to ACCESS. See “DDL and DCL Statements, Dictionary Access, and Locks” on page 766.
AMP-Based Utilities and Logging

There are two AMP-based utilities related to logging:

- **Locking Logger**
  This utility produces a report of miscellaneous database lock information that you can use to detect blocked transactions and global deadlocks. The data reported by the Locking Logger is extracted from various database transaction logs.
  To enable the logging of locks, the LockLogger flag of the DBS Control utility must be set to TRUE (see *Utilities* for details).

- **Show Locks**
  This utility produces a report on the various HUT locks (see *Teradata Archive/Recovery Utility Reference* and “Client Utility Locks and Teradata Database” on page 732) currently in place on various Teradata Database resources.

Both of these utilities are documented in the second volume of *Utilities*. 
Client Utility Locks and Teradata Database

Introduction

The Teradata Tools and Utilities programs that perform tasks such as bulk data loads (FastLoad, MultiLoad, BTEQ), bulk data exports (FastExport, BTEQ), stream data loads (Teradata Parallel Data Pump), both data loads and data exports (Teradata Parallel Transporter), and archiving, recovering, or restoring tables and databases (Archive/Recovery) also place locks at different levels and severities on Teradata Database resources (see “Locking Levels” on page 723 and “Locking Severity and Relative Restrictiveness” on page 724 for details of locking levels and severities).

The client utilities place two basic types of locks on Teradata Database resources:

- **Standard database locks**
  
  All of the client data loading and exporting utilities except Archive/Recovery use standard database transaction locks (see “Database Locks, Two-Phase Locking, and Serializability” on page 706).

  For details about when the various database locks that client utilities place and release on resources in Teradata Database, see the following Teradata Tools and Utilities manuals:

  - *Basic Teradata Query Reference*
  - *Teradata FastExport Reference*
    
    The Teradata Parallel Transporter uses the FastExport protocol for its EXPORT operator (see *Teradata Parallel Transporter Reference* for details).
  
  - *Teradata FastLoad Reference*
    
    The Teradata Parallel Transporter uses the FastLoad protocol for its LOAD operator (see *Teradata Parallel Transporter Reference* for details).

  - *Teradata MultiLoad Reference*
    
    The Teradata Parallel Transporter uses the MultiLoad protocol for its UPDATE operator (see *Teradata Parallel Transporter Reference* for details).

  - *Teradata Parallel Data Pump Reference*
    
    The Teradata Parallel Transporter uses the Teradata Parallel Transporter protocol for its STREAM operator (see *Teradata Parallel Transporter Reference* for details).

  - *Teradata Parallel Transporter Reference*

- **Host Utility (HUT) locks**

  HUT locks are the locks set by the Archive/Recovery utility and the various commands related to it. Unlike database locks, you must remove HUT locks explicitly by specifying the RELEASE LOCK command in your ARCHIVE, RESTORE, ROLLBACK, or ROLLFORWARD job or by specifying the RELEASE LOCK option with the Archive/Recovery-related command you submit because the system does not release HUT locks automatically, even retaining them after a command aborts, a job terminates, or a system restart.
Another difference between HUT locks and database locks is that for Archive operations, HUT locks are placed only on the tables or databases (or both) on the set of AMPs being archived, while database transaction table- and database-level locks are placed on those objects across all AMPs in the system.

Finally, HUT locks are associated with the user who submits the Archive/Recovery job, not with the job or with a transaction, while database locks are always associated exclusively with a transaction.

See Teradata Archive/Recovery Utility Reference for details.

**HUT Lock Levels**

The Archive/Recovery utility places HUT locks at the following levels only:

- **Table**
- **Database**

Note that the system does not place HUT locks at the row hash level for any of the operations performed by the Archive/Recovery utility.

Whether the locks are set at the Table or Database level, or both, depends on what you specify is to be archived, checkpointed, restored, or recovered.

**HUT Lock Severities**

The Archive/Recovery utility places HUT locks with the following locking severities only:

- **READ** or **GROUP READ** for ARCHIVE operations.

Note that a GROUP READ lock sets a HUT ACCESS lock on its subject table internally for the duration of the Archive operation on that table as well as placing a series of rolling HUT READ locks on successive blocks of rows demarcated by a system-determined range of rowhash values. GROUP READ locks are implemented internally as ROW RANGE locks (see “GROUP READ Locks” on page 736 for details).

GROUP READ locks are implemented internally as Row Range locks, and have the same level of severity as a standard READ lock (see “Locking Severity and Relative Restrictiveness” on page 724).

- **EXCLUSIVE** for the following operations:
  - **BUILD**
    - Locks are placed at the following levels:
      - **TABLE** for a table-level build.
      - **DATABASE** for a database-level build.
  - **COPY**
    - Locks are placed at the TABLE level.
• **RESTORE**  
  Locks are placed at the following levels for an all-AMP restore, depending on both the ARCHIVE and RESTORE command specifications, as follows:

<table>
<thead>
<tr>
<th>IF the ARCHIVE is done at this level ...</th>
<th>AND the RESTORE is done at this level ...</th>
<th>THEN the system places an EXCLUSIVE HUT lock at this level ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>database</td>
<td>DATABASE</td>
</tr>
<tr>
<td>database</td>
<td>table</td>
<td>TABLE</td>
</tr>
<tr>
<td>table</td>
<td>database</td>
<td>TABLE on each of the tables being restored, one table at a time.</td>
</tr>
<tr>
<td>table</td>
<td>table</td>
<td>TABLE on each of the tables being restored, one table at a time.</td>
</tr>
</tbody>
</table>

• **REVALIDATE REFERENCES**  
  Locks are placed at the TABLE level.

• **ROLLBACK**  
  Locks are placed at the TABLE level on the journal.

• **ROLLFORWARD**  
  Locks are placed at the TABLE level on the journal.

• **WRITE for CHECKPOINT operations.**  

Note that the system does not place HUT locks at the ROW hash level or the CHECKSUM severity.

The following Archive/Recovery utilities place *database* locks, not HUT locks:

<table>
<thead>
<tr>
<th>Utility</th>
<th>Database Locks Placed</th>
<th>Objects Locked</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE DATABASE</td>
<td>EXCLUSIVE at database level.</td>
<td>Database being deleted</td>
</tr>
<tr>
<td>WRITE at table level</td>
<td></td>
<td>Dictionary tables such as the following when necessary:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DBC.TVM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DBC.TVFields</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DBC.Indexes</td>
</tr>
<tr>
<td>LOGGING ONLINE ARCHIVE OFF</td>
<td>none.</td>
<td>Table or database being logged</td>
</tr>
<tr>
<td>ROWHASH.</td>
<td></td>
<td>• DBC.RCEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DBC/archiveLoggingObjsTbl</td>
</tr>
</tbody>
</table>
HUT Lock Interactions With Database Transaction Locks

HUT ACCESS, HUT READ, HUT GROUP READ, and HUT EXCLUSIVE locks interact with locks placed by Teradata Database in exactly the same manner as database locks of the same severity interact with other database locks.

For example, the exclusion of other lock requests on the same database resource enforced by a HUT EXCLUSIVE lock placed during a Restore operation is identical to the exclusion of other lock requests enforced by an EXCLUSIVE locked placed by an operation within Teradata Database (see the table near the end of the topic “Compatibility Among Locking Severities” on page 726 for details).

From a user perspective, there are only two important differences between HUT locks and database transaction locks: their duration and their scope.

<table>
<thead>
<tr>
<th>Lock Type</th>
<th>Duration</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database transaction</td>
<td>Held only for the time it takes for a transaction to either commit or roll back. Database transaction locks are released automatically as soon as the transaction that set them completes.</td>
<td>Session-level, meaning that the lock applies to any request submitted by the session running the locking transaction. Another session running against the same set of objects must set its own locks, even if it is running under the same user.</td>
</tr>
</tbody>
</table>
**GROUP READ Locks**

The HUT GROUP READ lock is a hybrid type that combines an ACCESS lock on the entire subject table with a moving READ lock that locks successive sets of system-determined ranges of rowhash values. The internal name for the HUT GROUP READ lock is a ROW RANGE lock.

The following graphic illustrates how a GROUP READ lock works.

<table>
<thead>
<tr>
<th>Lock Type</th>
<th>Duration</th>
<th>Scope</th>
</tr>
</thead>
</table>
| HUT       | A HUT lock is only released if you issue an explicit RELEASE LOCK command, or if the batch locking operation placing the lock completes successfully and also specifies the RELEASE LOCK command. HUT locks remain even after any of the following events occur:  
  • The utility command that placed the HUT lock fails.  
  • The ARCHIVE job in which the HUT lock was placed terminates.  
  • Teradata Database experiences a restart.  
  Because there is only one lock, a RELEASE LOCK command against a set of HUT-locked database objects releases all existing HUT locks, even if other jobs submitted by that user continue to access the previously HUT-locked object.  
  See *Teradata Archive/Recovery Utility Reference* for warnings about releasing HUT locks while other jobs are running. Such actions can lead to an inconsistent archive. | User-level, meaning that the lock applies to any ARCHIVE operation a user is performing on the object.  
  A user can only place one HUT lock on a database object at a time. An exerted lock allows any ARCHIVE operation from that user to access the database object set on which it holds locks, but blocks other users from acquiring conflicting HUT locks on that object set.  
  HUT locks block all conflicting transaction lock attempts on the object set, even if they are requested by the same user. |
In terms of their interaction with database locks, GROUP READ LOCKS combine the properties of table-level ACCESS locks on the AMP currently being archived throughout the duration of their application, and rowhash-level READ locks as the system successively locks, reads, archives, and releases the READ lock on each range of row hash values on that AMP.

The process is as follows:

1. Lock the subject table for ACCESS on the first AMP that contains rows to be archived.
2. Place a rowhash-level READ lock on the first block of rows belonging to the system-determined range of rowhash values in the subject table on the current AMP.
3. Read that block of rows for Archive.
4. Write the block of rows to the archive medium.
5. Release the READ lock on the first block of rows.
6. Place a READ lock on the second block of rows belonging to the system-determined range of rowhash values in the subject table on the current AMP.
7 Make the following check:

<table>
<thead>
<tr>
<th>IF all the rows in the subject table on the current AMP …</th>
<th>THEN …</th>
</tr>
</thead>
<tbody>
<tr>
<td>have been archived</td>
<td>move to the next AMP in the archive sequence and lock the subject table for ACCESS.</td>
</tr>
<tr>
<td>have not been archived</td>
<td>place a READ lock on the next block of rows belonging to the system-determined range of rowhash values in the subject table on the current AMP.</td>
</tr>
</tbody>
</table>

8 Iterate stages 2 through 7 until all the rows in the subject table on all AMPS have been archived.

9 Release the ACCESS lock on the subject table.

10 End of process.
Default Lock Assignments and Lock Upgradeability

Introduction

The Lock Manager assigns locking levels and severities to SQL requests by default. 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 before the system commits the transaction.

Default Lock Assignments

The following table lists some of the default lock assignments the Lock Manager applies for various SQL requests and access strategies:

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>Access Type</th>
<th>Default Lock Type Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UPI or USI</td>
<td>NUSI or FTS</td>
</tr>
<tr>
<td>CREATE DATABASE</td>
<td>Not applicable</td>
<td>Database</td>
</tr>
<tr>
<td>DROP DATABASE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODIFY DATABASE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>Not applicable</td>
<td>Table</td>
</tr>
<tr>
<td>DROP TABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALTER TABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>Row hash</td>
<td>Table</td>
</tr>
<tr>
<td>INSERT</td>
<td>Row hash</td>
<td>Not applicable</td>
</tr>
<tr>
<td>MERGE</td>
<td>Row hash</td>
<td>• Not applicable for INSERT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Table for UPDATE</td>
</tr>
<tr>
<td>SELECT</td>
<td>Row hash</td>
<td>Table</td>
</tr>
<tr>
<td>SELECT AND CONSUME</td>
<td>Row hash</td>
<td>Row hash</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Row hash</td>
<td>Table</td>
</tr>
</tbody>
</table>

a. If a SELECT operation is part of a tactical query, the system might downgrade its READ lock to an ACCESS lock to avoid blocking. See “DDL and DCL Statements, Dictionary Access, and Locks” on page 766 for further information.

b. The system does not grant the lock if the request is delayed because there are no rows in the queue table. As soon as a row is inserted into the table, the system grants the lock, and transaction processing resumes.

25. The Optimizer determines what locking levels and severities are necessary to service a request, but it does not actually assign locks.
The following table lists the default lock assignments for various Archive/Recovery utility commands. A lock severity not preceded by the keyword HUT is a database transaction lock.

<table>
<thead>
<tr>
<th>FOR this Archive/Recovery command …</th>
<th>The system places this lock by default …</th>
<th>At this level …</th>
<th>Which can be downgraded to this lock …</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ARCHIVE</td>
<td>HUT READ</td>
<td>• Table for a table-level archive</td>
<td>HUT GROUP READ\textsuperscript{b}</td>
</tr>
<tr>
<td>• DUMP\textsuperscript{a}</td>
<td></td>
<td>• Database for a database-level archive</td>
<td></td>
</tr>
<tr>
<td>FOR this Archive/Recovery command ...</td>
<td>The system places this lock by default ...</td>
<td>At this level ...</td>
<td></td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>---------------------------------</td>
<td>-----------------</td>
<td></td>
</tr>
<tr>
<td>BUILD</td>
<td>HUT EXCLUSIVE</td>
<td>• Table for a table-level build</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Database for a database-level build</td>
<td></td>
</tr>
<tr>
<td>COPY</td>
<td></td>
<td>Table</td>
<td></td>
</tr>
<tr>
<td>RESTORE</td>
<td></td>
<td>• Table for a table-level restore</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Table for a partition-level restore</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Database for a database-level restore</td>
<td></td>
</tr>
<tr>
<td>REVALIDATE REFERENCES</td>
<td></td>
<td>Table</td>
<td></td>
</tr>
<tr>
<td>ROLLBACK</td>
<td></td>
<td>Table</td>
<td></td>
</tr>
<tr>
<td>ROLLFORWARD</td>
<td></td>
<td>Table</td>
<td></td>
</tr>
<tr>
<td>LOGGING ONLINE ARCHIVE OFF</td>
<td></td>
<td>Rowhash</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• none on the logged object set</td>
<td>rowhash</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• WRITE on $DBC.RCEvent$ and $DBC.ArchiveLoggingObjTbl$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGGING ONLINE ARCHIVE ON</td>
<td>READ on the logged object set</td>
<td>Depends either on the locked object set and the aggregate number of tables per request or the number of tables contained per user or database. The level is always one of the following:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Table</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Database</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HUT ACCESS</td>
<td>Database</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE on $DBC.RCEvent$ and $DBC.ArchiveLoggingObjTbl$</td>
<td>Rowhash</td>
<td></td>
</tr>
<tr>
<td>DELETE DATABASE</td>
<td>EXCLUSIVE</td>
<td>Database</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE on DBC tables as necessary for dropping tables in the database being deleted.</td>
<td>Table</td>
<td></td>
</tr>
</tbody>
</table>

a. DUMP is a deprecated synonym for ARCHIVE.

b. A HUT GROUP READ lock is a downgrade from a HUT READ lock because it sets a HUT ACCESS lock at the table level, then applies a rolling HUT READ lock on a sequence of row sets within the ACCESS-locked table.
Note that no HUT lock can be upgraded, but several can be downgraded in selected situations.

**Changing Lock Assignments With LOCKING Request Modifier**

Depending on the assigned lock and the individual SQL request, you can change some default lock assignments using the LOCKING request modifier (see *SQL Data Manipulation Language* for syntax and usage details). You can upgrade any lower severity lock to a higher severity lock, but the only downgrade permitted is from a READ lock to an ACCESS lock.

The following table provides a summary of the allowable changes for database lock upgrades:

<table>
<thead>
<tr>
<th>A change from this default assignment lock ...</th>
<th>TO this user-specified lock ...</th>
<th>IS ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS^a</td>
<td>ACCESS</td>
<td>redundant, but valid.</td>
</tr>
<tr>
<td>READ^b SHARE</td>
<td>READ SHARE</td>
<td>a valid upgrade.</td>
</tr>
<tr>
<td>WRITE</td>
<td>WRITE</td>
<td></td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td>EXCLUSIVE</td>
<td></td>
</tr>
<tr>
<td>ACCESS</td>
<td>READ SHARE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EXCLUSIVE</td>
<td></td>
</tr>
<tr>
<td>READ SHARE</td>
<td>WRITE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EXCLUSIVE</td>
<td></td>
</tr>
<tr>
<td>WRITE</td>
<td>EXCLUSIVE</td>
<td>a valid downgrade.</td>
</tr>
<tr>
<td>READ</td>
<td>ACCESS</td>
<td></td>
</tr>
</tbody>
</table>

a. Even though a CHECKSUM lock is essentially identical to an ACCESS lock, you can only specify CHECKSUM locks explicitly using the LOCKING request modifier; you cannot upgrade them to a higher severity lock. This is because Teradata Database never specifies CHECKSUM as a default lock severity.

b. SHARE is a deprecated synonym for READ.
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:

<table>
<thead>
<tr>
<th>This LOCKING request modifier severity specification</th>
<th>Is available for this SQL DML statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCLUSIVE</td>
<td>DELETE, INSERT, MERGE, SELECT, UPDATE</td>
</tr>
<tr>
<td>WRITE</td>
<td>SELECT, SELECT AND CONSUME</td>
</tr>
<tr>
<td>READ</td>
<td>SELECT</td>
</tr>
<tr>
<td>ACCESS</td>
<td>SELECT</td>
</tr>
</tbody>
</table>

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 because doing so would compromise the integrity of the database. Because the SELECT statement does not update data, and therefore its actions cannot compromise database integrity, you are permitted to change its default locking severity to any other severity. This option does not extend to its SELECT AND CONSUME variant, for which the severity can only be upgraded to WRITE or EXCLUSIVE.

Note that you cannot upgrade any of the HUT locks, but you can downgrade several of them (see “Default Lock Assignments” on page 739 for details).

**Rules for Upgrading Locks**

Upgrading system locks helps to minimize deadlocks, but lessens concurrency, which can downgrade system performance if not done selectively. 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 (typically from READ to WRITE) can occur when conflicting locks are required by multiple statements in a single transaction.

You can determine the current status of operations such as request blocking and transaction aborts for a particular session using the Query Session utility (see Utilities for details on how to use Query Session).
Releasing Locks

Except for HUT locks, users have no control over when locks are released. You must explicitly release any HUT locks set by Archive/Recovery processing by specifying either a RELEASE LOCK command after an Archive/Recovery-related command completes its processing or by specifying the RELEASE LOCK option for the applicable Archive/Recovery command (see Teradata Archive/Recovery Utility Reference for details).

The Lock Manager releases locks upon completion of any of the following:

- Implicit (Teradata mode) or two-phase commit (2PC mode) transaction.
- Outermost END TRANSACTION statement of an explicit transaction (Teradata mode).
- COMMIT of an ANSI mode transaction.
- ROLLBACK/ABORT of any transaction.

This lock release occurs regardless of when you receive the response to a request because the spool file might exist after the end of the transaction.

Each of these actions also drops the Transient Journal and closes any open cursors.

Guidelines for Changing Default Lock Assignments and Changing Intratransaction Request Orders to Maximize Concurrency

The following set of guidelines set out some rules of thumb for maximizing concurrency with your database transactions.

- Use ACCESS locks in place of READ locks whenever the application can tolerate dirty reads (see “The Uncommitted Dependencies (Dirty Read) Phenomenon” on page 715).
- 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 (see “LOCKING” in SQL Data Manipulation Language for information about how to do this).
- If the CREATE INDEX or ALTER TABLE … FALLBACK LOCKING request modifier specifies WRITE (or if there is no LOCKING request modifier specified), then specify an ACCESS lock in your SELECT statement to permit concurrency (see “LOCKING” in SQL Data Manipulation Language for information about how to do this).
  
  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 also result in deadlocks, you should consider running only read-only transactions (or access-only if a LOCKING FOR WRITE clause is specified) concurrently with ALTER TABLE … FALLBACK or CREATE INDEX statements. See “LOCKING” in SQL Data Manipulation Language for information about how to do this.
- 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.
- You should always place SELECT AND CONSUME statements as early as possible in a transaction to avoid conflicts with other database resources. This is to minimize the likelihood of a situation where the SELECT AND CONSUME TOP 1 statement would enter a delayed state while holding locks on resources that might be needed by other requests.
Blocked Requests

Introduction

A request that is waiting in a lock queue (see “Compatibility Among Locking Severities” on page 726) is considered to be blocked (see “Compatibility Among Locking Severities” on page 726). If you suspect that a request is blocked, you can use the Query Session utility (see Utilities) to confirm or refute your suspicions about the status of the session.

Any incompatibility with an EXCLUSIVE lock can result in a queue of several blocked requests, all of which must wait until the system releases the blocking EXCLUSIVE lock.

The blocking behavior of HUT locks is identical to that of database transaction locks, as is the interactive blocking behavior of HUT and database transaction locks.

Blocking and Transaction Throughput

Blocking due to lock conflicts occurs in any system that uses locking. Tay et al. (1985) found that blocking imposes the upper bound on transaction throughput. In other words, of all the factors that affect transaction performance, lock contention has the greatest potential to be the most severe mitigating effect. When system performance becomes affected enough to be a problem, the condition is referred to as thrashing.

Because of this potentially critical performance issue, some have advocated replacing locking systems with optimistic concurrency control (OCC) methods. The general idea of OCC methods is analogous to instruction pipelining in a CPU or the old CSMA/CD Ethernet LAN protocol: go ahead and try, and if you fail, you just try again. To grossly oversimplify the concept, the premise is that for any given set of concurrently running transactions, the likelihood that any of their individual operations will violate the isolation principle is small. Therefore, it should enhance transaction throughput to dispense with locking and just go ahead and hope that isolation is not violated. The check is to determine whether conflicting transactions have violated the isolation of other transactions only at commit time. If a violation is found, then the offending transaction is rolled back.

Despite their theoretical attractiveness and their demonstrated power in some controlled situations (Kung and Robinson, 1981), OCC methods have not proven to be practical for most production database management environments (Framszek and Robinson, 1985; Haerder, 1984; Mohan, 1992 - see Thomasian (1998) for a review).

26. A consume mode SELECT statement that has not been granted a lock because it is in a delay state is not considered to be blocked. However, if such a request is awakened and is placed into a lock queue, it is then considered to be blocked. See the footnote to “Explicit Transactions” on page 722.

27. The problem was originally called “the convoy phenomenon” (Blasgen et al., 1979), but thrashing is now more commonly used. Tay (1987) also notes that the convoy problem was actually reported for what he refers to as resource locks, not data locks. The lock contention reported for the convoy phenomenon were on the System R buffer pool, recovery log, and system entry and exits.
Blocking and Deadlock Are Not The Same Thing

When the Lock Manager places a request in a lock queue, it is known that the request will execute as soon as it arrives at the head of the queue. Blocked requests do not time out; they remain in the queue as long as it takes to reach its head, at which time they are granted the locks they are waiting for and execute.

The only requests the Lock Manager never enqueues are those explicitly specified with a LOCKING request modifier and the NOWAIT option (see SQL Data Manipulation Language for information about the LOCKING request modifier). Such a request aborts immediately if it cannot acquire the specified lock, and it is the responsibility of the user to resubmit the aborted request.

The root word “dead” in deadlock is used with good cause. A deadlock is a lock contention situation that cannot be solved without external intervention of some kind (see “Deadlock” on page 753). For Teradata Database, the external intervention is accomplished by:

1 Aborting the younger request in the deadlock.
2 Placing it back into the lock queue to be executed at a later time.

Problem: Single Request Transactions

When several requests that compete for the same table are submitted as separate, single request transactions, the Lock Manager resolves their locking requirements as illustrated by the following 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.

28. The opposite of deadlock is livelock (see Ullman, 1982), a term that has no meaning for locking schemes that use first come, first served locking strategies. Suppose there are three concurrently running transactions, Tx1, Tx2, and Tx3. Suppose Tx1 has object A locked, and Tx2 is waiting to gain access to A. If Tx1 drops its lock on A, but Tx3 then acquires that lock before Tx2 can, the situation is referred to as livelock, because Tx2 could conceivably wait forever to acquire the lock on A, but it is serendipity that prevents it from doing so, not a deadlocked contention for resources among transactions. The duration of the wait by Tx2 on A cannot be determined.
6  Job_6 requires an ACCESS lock.
   This conflicts with the queued EXCLUSIVE lock, so job_6 is queued behind job_5.

7  End of process.

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:

<table>
<thead>
<tr>
<th>Stage</th>
<th>This transaction number</th>
<th>Has this request</th>
<th>That requires this lock</th>
<th>With this result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>SELECT ... FROM table_a</td>
<td>READ</td>
<td>the READ lock on table_a is granted and the SELECT request begins processing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INSERT INTO table_a</td>
<td>WRITE</td>
<td>the WRITE lock on table_a is not granted and the INSERT into table_a request is blocked pending completion of the SELECT request.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>INSERT INTO table_a</td>
<td>WRITE</td>
<td>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.</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>SELECT ... FROM table_a</td>
<td>None. The READ lock is released.</td>
<td>processing of the SELECT request from Transaction 1 completes.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>INSERT INTO table_a</td>
<td>WRITE</td>
<td>the Lock Manager upgrades the lock on table_a for the Transaction 1 INSERT request.</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>INSERT INTO table_a</td>
<td>WRITE</td>
<td>transaction 2 remains queued and inactive, waiting for its WRITE lock request on table_a to be granted.</td>
</tr>
</tbody>
</table>
Pseudo Table Locks

Introduction

You can think of a pseudo table as an alias for the physical table it represents. The purpose of pseudo tables is to provide a mechanism for queueing table locks in order to avoid the global deadlocking that can otherwise occur in response to a full-table scan request in a parallel system (see “Deadlock” on page 753).

When you make an all-AMP request for a READ, WRITE, or EXCLUSIVE lock, the system automatically imposes pseudo table locking. You can think of pseudo table locks as intention locks, though not in the same sense the term is used in the transaction literature.29

Properties of Pseudo Table Locks

Pseudo table locks enable sequential locking on database objects that span multiple AMPs in a parallel database architecture. Without pseudo table locking, if multiple users simultaneously submit an all-AMP request on the same table, a deadlock is almost certain to occur because if multiple requests on that table are sent in parallel, they are likely to arrive at the various AMPs holding the table rows in different sequential orders, and each request will lock the rows belonging to that table on different AMPs, creating a deadlock situation.

For example, suppose a request from user_1 locks table rows on AMP3, while user_2 locks the table rows on AMP4 first. When the user_1 request attempts to lock table rows on AMP4 (or when the user_2 request attempts to lock table rows on AMP3), a global deadlock occurs. Pseudo table locks prevent such deadlocks from occurring.

Pseudo table locking has the following properties:

- Each physical table, whether user or dictionary, has a system-assigned table ID hash code alias. This hash code constitutes a pseudo table.
- The system places pseudo table locks on this table ID hash code whether the lock to be placed is at the table- or row hash-level. Row hash pseudo table locks are placed only on data dictionary tables with fallback.
- The table ID hash codes are evenly distributed across the AMPs.
- Each AMP is a gatekeeper for the tables to which it has been assigned table ID hash codes.
- Any all-AMP requests for READ, WRITE, or EXCLUSIVE locks are always dispatched to the relevant gatekeeper AMP for pseudo table lock processing.

29. An intention lock is an indicator placed on the ancestors of a locked entity in a multigranular locking scheme. For example, if you were to place a row-level lock, the lock manager would also place an intention lock on the table that contained that row and also, perhaps, on the database that contained the table. Intention locks allow a lock scheduler to determine whether there are locks on the ancestors of an entity it seeks to lock that would implicitly lock that entity in a conflicting mode. Refer to any of the texts listed in “References” on page 801 for a detailed explanation of intention locks.
With some exceptions, the system places pseudo table locks only at table-level.

The exceptions to the rule about pseudo table locks being placed at only table-level are the following DDL statements, for which pseudo table locks are placed on various data dictionary fallback tables at the row hash-level:

- CREATE HASH INDEX
- CREATE JOIN INDEX
- CREATE TABLE
- DROP HASH INDEX
- DROP JOIN INDEX
- DROP MACRO
- DROP PROCEDURE
- DROP TABLE
- DROP VIEW

**Example Scenario**

For example, consider the following scenario:

1. *User_1* submits an all-AMPs request.
2. The request-originating PE sends a message to the gatekeeper AMP for the table.
3. The gatekeeper AMP places a lock on the pseudo table (table ID hash).
4. Because the table is not currently locked, the *user_1* request obtains the requested lock and proceeds.
5. Meanwhile, *user_2* submits an all-AMP request for the same table.
6. The request-originating PE sends a message to the gatekeeper AMP for the table.
7. The gatekeeper AMP places a lock on the pseudo table hash ID.
8. Because *user_1* already has the table locked, the request from *user_2* must wait in a queue until the request submitted by *user_1* releases its locks on the table by either committing or aborting its transaction. Because the *user_2* request was the next in sequence to place a pseudo table lock on the table, it is next in the queue to lock the table for processing.

Teradata Database handles all-AMP lock requests as follows:

1. The PE that processes an all-AMPs request determines the table ID hash for an AMP to manage the all-AMP lock request similar to how a PE is assigned to an active queue table (see “Queue Table FIFO Cache Management” on page 71) and to how an AMP is selected for collecting a single-AMP random statistical sample (see “Random AMP Sampling” on page 178).
2. The Lock Manager places a pseudo table lock on the table.
3. The first request to place a pseudo table lock acquire a lock on the relevant table across all AMPs.
The following graphic illustrates the process:

First request

Second request

Determine Table ID hash

AMP

AMP

AMP

AMP

101A313
Deadlock

Introduction

A problem that can occur with two-phase locking (see “Two-Phase Locking” on page 706) is the situation in which two requests lock database resources the other needs to continue processing its transaction. All the transactions neutralized by this outcome are in a simultaneous wait state, with each waiting for another to release a locked resource before it can continue. If such a stalemate cannot be resolved without explicit intervention, it is referred to as a deadlock, or deadly embrace.

This is an entirely different situation than the case in which individual requests are waiting for a lock to be released. Note that Teradata lock requests do not time out, so a request waits in the lock queue until it is granted the lock it needs.30

Detecting and Handling Deadlock for Database Locks

Teradata Database supports deadlock detection and handling at two different levels:

• AMP-local
  The AMP-local deadlock detection software checks for deadlocks at a fixed 30 second interval.

• Global
  Global deadlock detection runs at a user-defined period set by the DBS Control utility using the DeadLockTimeOut field. The global deadlock detection software runs by default with a period of four minutes, but there are times when you might want to set it to a shorter interval.

Whenever the system detects a deadlock, it rolls back the most recently initiated (newest, or youngest) transaction of the two in the deadlock.

Detecting and Handling Deadlock for Archive/Recovery HUT Locks and Other Client Utilities

The global deadlock detector does not detect HUT deadlocks.

Because the other client utilities all use database locking, the global deadlock detector does know when they become trapped in a deadlock. To resolve the deadlock, the system aborts the deadlocked transaction, rolls back its updates, and returns a retryable 2631 error to the requesting application or utility.

30. This is true unless you specify the NO WAIT option in a LOCKING clause that modifies the request, in which case it aborts immediately and rolls back any updates made by the containing transaction.
The following table summarizes what happens to a transaction submitted from a client utility or application when a retryable 2631 error occurs, and then provides a very basic solution for the problem.

<table>
<thead>
<tr>
<th>IF the client utility or application is ...</th>
<th>AND ...</th>
<th>THEN ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTEQ</td>
<td>.SET RETRY is active</td>
<td>even though the entire transaction has been rolled back, BTEQ retries only the failed request and any subsequent requests in the transaction, not the entire transaction. If this occurs, the updates made by requests in the transaction prior to the failed request are lost, as is the contextual information indicating that a transaction is in progress.</td>
</tr>
</tbody>
</table>
| anything else                             |         | you must code the application to:  
1 Check for error code 2631.  
2 If error code 2631 is detected, then resubmit the entire aborted transaction. |

**Local Deadlocks**

Each request in a transaction places its own locks as it runs and continues to hold the locks it acquires until the transaction either commits or rolls back.

The disadvantage of this is that it can result in deadlocks when multiple concurrent accesses against the same database objects occur. The application-level solution to this potential problem is to apply all the explicit database- and table-level locks you need in a single request at the beginning of a transaction.
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.

Retrievals With ACCESS Lock Severities

Because an ACCESS lock is compatible with all locking severities 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 application holding an ACCESS lock while that same data is simultaneously being modified by an application holding a WRITE lock. 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 LOCKING 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 LOCKING request modifier in your SQL statements.

How You Can Use The LOCKING Modifier To Help Prevent Deadlocks

The LOCKING request 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 LOCKING 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 LOCKING Modifier To Enhance Concurrency

To make more efficient use of system resources, you can use the LOCKING request 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 LOCKING request modifier:

- The LOCKING request modifier is not ANSI SQL-2003-compliant.
- The LOCKING request modifier can precede any SQL request or it can be used alone, without modifying an SQL statement. The LOCKING clause is typically used as a modifier, in which case it precedes the statement. For example, the following statement is valid:
  
  LOCKING TABLE tablename FOR READ;

- More than one LOCKING request modifier can be specified in the same request.
- The CREATE VIEW and REPLACE VIEW statements allow the LOCKING request modifier to be specified as part of the view definition.
- You can use the LOCKING request 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 LOCKING with the NOWAIT option to abort a transaction if a lock cannot be granted immediately.
- LOCKING 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\(^\text{31}\).

31. Locks for row hash operations are not documented by EXPLAIN reports.
The following example uses the LOCKING phrase as both a modifier and as a statement:

```
LOCKING TABLE table_a FOR READ
LOCKING TABLE table_b FOR READ
SELECT ... FROM table_a, table_b WHERE ...;
LOCKING 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 session mode):

<table>
<thead>
<tr>
<th>User</th>
<th>SQL Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BEGIN TRANSACTION; SELECT y FROM table_a WHERE pi = 1; UPDATE table_a SET y = 0 WHERE pi = 1; END TRANSACTION;</td>
</tr>
<tr>
<td>B</td>
<td>BEGIN TRANSACTION; SELECT z FROM table_a WHERE pi = 1; UPDATE table_a SET z = 0 WHERE pi = 1; END TRANSACTION;</td>
</tr>
</tbody>
</table>

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 request 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 request 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 LOCKING ROW [FOR] WRITE/EXCLUSIVE phrase. This phrase does not override any lock already held on the target table.

LOCKING 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:

<table>
<thead>
<tr>
<th>User</th>
<th>SQL Text</th>
</tr>
</thead>
</table>
| A    | BEGIN TRANSACTION;  
|      | LOCKING 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;  
|      | LOCKING ROW WRITE  
|      | SELECT z  
|      | FROM table_a  
|      | WHERE USI = 1;  
|      | UPDATE table_a  
|      | 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 LOCKING ROW ... request is granted only after the UserA END TRANSACTION statement has been processed.

**The LOCKING Modifier and the NOWAIT Option**

If your request cannot wait in a lock queue, you can specify the LOCKING request modifier with the NOWAIT option.

NOWAIT specifies that the entire transaction, even in ANSI session 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.
The LOCKING Modifier in View Definitions

The LOCKING request 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 LOCKING 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 LOCKING request modifier can be used to override the default READ lock with an ACCESS lock. For example:

```
LOCKING TABLE tablename FOR ACCESS
SELECT ...
FROM tablename ...;
```

Be aware that the effect of LOCKING FOR ACCESS is that of reading while writing, so dirty reads can occur with this lock. The best approach to specifying ACCESS locks is to use them only when you are interested in a broad, statistical snapshot of the data in question, not when you require precise results.

ACCESS locking can result in incorrect or inconsistent data being returned to a requestor, as detailed in the following points:

- 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 LOCKING ROW modifier cannot be used to lock multiple row hashes. If LOCKING ROW FOR ACCESS is specified with multiple row hashes, the declaration implicitly converts to LOCKING 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 `QualifyAcnt` is defined as a secondary index, the following request:

```sql
LOCKING TABLE AccntRec FOR ACCESS
SELECT AcctNo, QualifyAcnt
FROM AccntRec
WHERE QualifyAcnt = 1587;
```

could return the following result:

```
<table>
<thead>
<tr>
<th>AcctNo</th>
<th>QualifyAcnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1761</td>
<td>4214</td>
</tr>
</tbody>
</table>
```

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.
Example: Transaction Without Deadlock

Introduction

The following example demonstrates how to optimize concurrency using the guidelines from “Releasing Locks” on page 744.

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 concurrently running transactions:

<table>
<thead>
<tr>
<th>Transaction Number</th>
<th>SQL Text</th>
</tr>
</thead>
</table>
| 1                  | LOCKING 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:

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`.
5. End of process.
Example: Transaction With 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 concurrently:

<table>
<thead>
<tr>
<th>Transaction Number</th>
<th>Request Number</th>
<th>SQL Text</th>
</tr>
</thead>
</table>
| 1                  | 1              | LOCKING table_1 FOR READ
                        CREATE INDEX (column_3, column_4) ON table_1; |
| 2                  | 2              | BEGIN TRANSACTION; |
| 3                  |                | SELECT * 
                        FROM table_1; |
| 4                  | 4              | UPDATE table_1 
                        SET column_1 = <value-1>
                        WHERE column_1 = <value-2>; |
| 5                  | 5              | END TRANSACTION; |

Problem Transactions
Assume the actions in a transaction are taken in the following order:

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.

8 End of process.

**Problem Resolution**

To avoid this deadlock, change your SQL in either of the following ways:

- Add the modifier LOCKING `table_1` FOR WRITE to the second transaction after request 2 as follows.
  ```sql
  BEGIN TRANSACTION;
  LOCKING table_1 FOR WRITE
  SELECT *
  FROM table_1;
  UPDATE table_1
  SET column_1 = value_1
  WHERE column_1 = value_2;
  END TRANSACTION;
  ```

- Remove the LOCKING request modifier from the first transaction.
  ```sql
  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 concurrently, with the first having started prior to the second:

<table>
<thead>
<tr>
<th>Transaction Number</th>
<th>SQL Text</th>
</tr>
</thead>
</table>
| 1                  | LOCKING 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.

<table>
<thead>
<tr>
<th>Transaction Number</th>
<th>SQL Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOCKING table_1 FOR EXCLUSIVE</td>
</tr>
<tr>
<td></td>
<td>CREATE INDEX (column_3, column_4) ON table_1;</td>
</tr>
<tr>
<td>2</td>
<td>SELECT *</td>
</tr>
<tr>
<td></td>
<td>FROM table_1</td>
</tr>
<tr>
<td></td>
<td>WHERE column_3 = 124</td>
</tr>
<tr>
<td></td>
<td>AND column_4 = 93;</td>
</tr>
</tbody>
</table>

The upgraded LOCKING 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, Dictionary Access, and Locks

The performance of a DDL or DCL statement causes the 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. Depending on the dictionary table, the system sometimes downgrades READ lock requests made by read-only tactical queries on the dictionary to ACCESS locks if the query would otherwise be blocked by WRITE locks placed on those tables by ongoing DDL operations.

If these read-only queries are not blocked, then they use the standard READ locks.

The following dictionary views and tables are affected by this locking downgrade on a blocked READ lock request:

- `DBC.AccLogRuleTbl`
- `DBC.ConstraintNames`
- `DBC.Indexes`
- `DBC.TableConstraints`
- `DBC.TextTbl`
- `DBC.TriggersV`
- `DBC.TVFields`
- `DBC.TVM`
- `DBC.UDFInfo`

The only SQL statements eligible for a dictionary access READ lock-to-ACCESS lock downgrade upon being otherwise blocked are the following:

- `SELECT`
- `HELP COLUMN`
- `HELP CONSTRAINT`
- `HELP INDEX`
- `HELP STATISTICS`
- `SHOW FUNCTION/HASH INDEX/JOIN INDEX/MACRO/METHOD/PROCEDURE/REPLICATION GROUP/TABLE/TRIGGER/TYPE/VIEW`

These are system-initiated lock downgrades: they are not specified as part of the request.
DML Requests and Locks

When it processes DML requests, the system accesses the information it needs from 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.

The default locks the system applies as for DML requests are listed in the following table:

<table>
<thead>
<tr>
<th>DML Request</th>
<th>Updated Columns</th>
<th>Selection Criteria</th>
<th>Object Locked</th>
<th>Locking Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>Not applicable</td>
<td>UPI or USI&lt;sup&gt;c&lt;/sup&gt;</td>
<td>Row hash</td>
<td>READ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NUPI&lt;sup&gt;d&lt;/sup&gt;</td>
<td>Set of rows</td>
<td>READ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any other&lt;sup&gt;3&lt;/sup&gt;</td>
<td>Table</td>
<td>READ&lt;sup&gt;e&lt;/sup&gt;</td>
</tr>
<tr>
<td>SELECT AND CONSUME</td>
<td>Not applicable</td>
<td>Any</td>
<td>Row hash</td>
<td>WRITE</td>
</tr>
<tr>
<td>DELETE&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Not applicable</td>
<td>UPI or USI</td>
<td>Row hash</td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NUPI</td>
<td>Set of rows</td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any other</td>
<td>Table</td>
<td>WRITE</td>
</tr>
<tr>
<td>INSERT … SELECT</td>
<td>Not applicable</td>
<td>Select table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPI or USI</td>
<td>Row hash</td>
<td>READ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NUPI</td>
<td>Set of rows</td>
<td>READ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any other</td>
<td>Table</td>
<td>READ</td>
</tr>
<tr>
<td>INSERT … [VALUES]&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Primary row</td>
<td>WRITE</td>
</tr>
<tr>
<td>UPDATE&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Neither UPI nor USI</td>
<td>UPI or USI</td>
<td>Row hash</td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td>Neither NUPI nor USI</td>
<td>NUPI</td>
<td>Set of rows</td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td>Any other</td>
<td>Table</td>
<td></td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td>USI</td>
<td>USI</td>
<td>Table</td>
<td>WRITE</td>
</tr>
<tr>
<td>MERGE (Update)&lt;sup&gt;1b&lt;/sup&gt;</td>
<td>Neither UPI nor USI</td>
<td>UPI or USI</td>
<td>Row hash</td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td>Neither NUPI nor USI</td>
<td>NUPI</td>
<td>Set of rows</td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td>Any other</td>
<td>Table</td>
<td></td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td>USI</td>
<td>USI</td>
<td>Table</td>
<td>WRITE</td>
</tr>
<tr>
<td>MERGE (Insert)&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Primary row</td>
<td>WRITE</td>
</tr>
</tbody>
</table>
a. 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.

b. 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.

c. If a join or hash index includes the specified columns, then the entire index is READ-locked and the base table is not accessed.

d. 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.

e. When the SELECT is a tactical query, the system downgrades the lock request from READ to ACCESS if the operation would otherwise be blocked.
Several locking issues apply to consume mode SELECT operations against a queue table, as detailed in the following bulleted list:

- The default lock assignment for a consume mode SELECT operation against a queue table has WRITE severity at the row hash level.
  You cannot lower the severity of this lock assignment.
- The default lock assignment for a consume mode SELECT operation against the target table of an INSERT … SELECT AND CONSUME operation has WRITE severity at the table level.
  Although you can specify a LOCKING request modifier, it has no effect on the behavior of the SELECT AND CONSUME operation: the system still retrieves rows in the same order and the query enters a delay state when the table is empty.
- The system does not grant a row hash-level WRITE lock on a queue table for a consume mode SELECT operation unless there is at least one row in the table to be consumed. This is unlike all other SQL requests, where the system grants locks at the time the it receives the request.

As soon as a row is inserted into the subject queue table, the following things happen:

- The system grants the WRITE lock on the first row hash in the queue.
- Transaction processing resumes.
- Rows are consumed until the queue is empty.
- End of process.

- You can include only one consume mode SELECT statement in a multirequest transaction in Teradata session mode unless you place all such consume mode SELECT statements between explicit BEGIN TRANSACTION and END TRANSACTION statements.

For example, the following multirequest transaction is valid:

```
BEGIN TRANSACTION;

SELECT AND CONSUME TOP 1 *
FROM myqueue;

SELECT AND CONSUME TOP 1 *
FROM myqueue;

SELECT AND CONSUME TOP 1 *
FROM myqueue;

END TRANSACTION;
```

32. Consume mode SELECT operations are also known as SQL SELECT AND CONSUME TOP 1 statements.
The following consume mode SELECT statements are also valid in Teradata session mode, but each is an individual implicit transaction (see “Transactions, Requests, and Statements” on page 702):

```
SELECT AND CONSUME TOP 1 col_1_qits, qsn
FROM myqueue;

SELECT AND CONSUME TOP 1 col_1_qits, USER, CURRENT_TIMESTAMP
FROM myqueue;

SELECT AND CONSUME TOP 1 *
FROM myqueue;
```

- There are no special considerations for specifying consume mode SELECT statements in ANSI session mode.

The following is a valid ANSI session mode example that specifies two consume mode SELECT requests:

```
SELECT AND CONSUME TOP 1 *
FROM myqueue;

SELECT AND CONSUME TOP 1 *
FROM myqueue;

COMMIT;
```

**Caution:** Every request you submit in ANSI session mode is treated as part of the same transaction until you submit an explicit COMMIT or ROLLBACK statement.

If you submit a ROLLBACK statement, then all the work that was done in the current session from the time the transaction began is rolled back and the work is lost.

- You should place your consume mode SELECT statements as early in a Teradata session mode transaction as possible to avoid conflicts with other database resources.

- You should avoid the following practices when issuing SELECT AND CONSUME statements:
  - Coding *any* SELECT AND CONSUME statements within explicit transactions.
  - Coding large numbers of SELECT AND CONSUME statements within a transaction, especially if there are also DELETE and UPDATE operations made on the same queue table as SELECT AND CONSUME statements.
    
    When the system performs a SELECT AND CONSUME operation on a queue table, it then also performs a row collection operation each time it does a delete or update operation on that same queue table, which has a performance impact.

  - You should place any action taken based on consuming a row from a queue table in the same transaction as the consume mode SELECT operation on that same queue table. This ensures that both the row consumption and the action taken on that queue table are committed together, so no row or action for that queue table is lost.

If no action is to be taken, then you should isolate any SELECT AND CONSUME statement as the only statement in a transaction.
• The system aborts any transaction in which the limit on the number of SELECT AND CONSUME statements, 2,210, is exceeded.

• You cannot code any of the following statements that operate on the same queue table as a SELECT AND CONSUME statement within the same multistatement request:
  • DELETE
  • MERGE
  • UPDATE

• You should avoid coding large numbers of DELETE and UPDATE operations on queue tables because of the negative effect on performance.

• You should restrict DELETE, MERGE, or UPDATE operations on queue tables to exceptional conditions because of the negative effect on performance. The critical factor is not how many such operations you code, but how frequently those operations are performed. You can ignore this admonition if, for example, you run an application that performs many DELETE, MERGE, or UPDATE operations only under rarely occurring, exceptional conditions. Otherwise, because of the likely performance deficits that result, you should code DELETE, MERGE, and UPDATE operations only sparingly, and these should never be frequently performed operations.

The reason for this advisory is that UPDATE, MERGE, and DELETE operations on a queue table are more costly than the same operations performed against a non-queue table because each such operation forces a full-table scan in order to rebuild the FIFO cache on the affected PE.

• Teradata Database uses the Teradata Workload Manager software to manage all deferred requests (transactions in a delayed state) against a queue table. As a result, you should optimize your respective use of the two features because a large number of deferred requests against a queue table can have a negative effect on the ability of the Teradata Workload Manager to manage not just delayed consume mode queries, but all queries, optimally.

• The system returns an error to the requestor when more than 20 percent of the sessions on a PE are already in a delayed state. Assuming the number of sessions is set for the default value of 120, the threshold number of delayed state sessions is 24.

• A SELECT AND CONSUME request can consume any rows inserted into a queue table within the same transaction.

33. The Teradata Workload Manager client application software does not need to be enabled to be used by the system to manage delayed consume mode requests.
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 Positioned Cursors

A SELECT statement performed when its associated cursor is opened causes Teradata Database 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 request modifier. The LOCKING request modifier is supported for updatable cursors, and the CHECKSUM locking severity is designed especially for use with LOCKING.

Example

This example uses LOCKING with CHECKSUM on table_1:

```
EXEC SQL 
REPLACE macro macro_2 AS 
(LOCKING TABLE table_1 FOR CHECKSUM 
SELECT i, text 
FROM table_1); 
```

How Cursors and Locks Interact

When a cursor is opened, the system 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 Database to UPDATE or DELETE the data row when the application specifies such an action against WHERE CURRENT OF cursor_name.
### Cursor Locking Modes

<table>
<thead>
<tr>
<th>IF the locking modifier is ...</th>
<th>THEN the system ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS</td>
<td>does <em>not</em> 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.</td>
</tr>
<tr>
<td>CHECKSUM</td>
<td>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 <em>not</em> 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.</td>
</tr>
</tbody>
</table>
**Transaction Semantics: Operating in ANSI or Teradata Session Modes**

**Introduction**

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

- ANSI
- Teradata

**Default Session Mode**

Teradata session mode ensures compatibility with legacy applications, although it introduces a few caveats:

- When you upgrade an existing Teradata Database, you might want to set the system default to Teradata session mode to provide compatibility for existing users and applications.
- New customers should consider setting the system default to ANSI session mode.

The default session 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 the system with the connect or the logon/run parcel sequence.

**Session Option: Changing Session Modes**

To change the mode for a session using client software based on the CLIv2 API, do any of the following:

<table>
<thead>
<tr>
<th>FOR this client software ...</th>
<th>USE these commands or options ...</th>
<th>TO switch to this session mode ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTEQ</td>
<td>.[SET] SESSION TRANSACTION ANSI</td>
<td>ANSI</td>
</tr>
<tr>
<td></td>
<td>.[SET] SESSION TRANSACTION BTET</td>
<td>Teradata</td>
</tr>
<tr>
<td></td>
<td>Log off prior to entering this command. The command does not take effect until the next logon. See <em>Basic Teradata Query Reference</em> for more detail on using BTEQ commands.</td>
<td></td>
</tr>
<tr>
<td>Preprocessor2</td>
<td>TRANSACT (ANSI)</td>
<td>ANSI</td>
</tr>
<tr>
<td></td>
<td>TRANSACT (BTET)</td>
<td>Teradata</td>
</tr>
<tr>
<td></td>
<td>TRANSACT (2PC)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRANSACT (COMMIT)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See <em>Teradata Preprocessor2 for Embedded SQL Programmer Guide</em> for more detail on setting Preprocessor options.</td>
<td></td>
</tr>
</tbody>
</table>

---

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

An application-initiated asynchronous abort causes full transaction rollback in both ANSI and Teradata session 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 session 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, the Transient Journal is dropped, and any open cursors are closed.

ANSI session mode only recognizes termination of a transaction by the explicit performance of a COMMIT or ABORT/ROLLBACK statement from the application. This means that statement failures do not cause a rollback of the transaction, only of the request that causes them. The system does not arbitrarily close a transaction unless its termination is required to preserve the integrity of the database.

In ANSI session 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 session modes:

<table>
<thead>
<tr>
<th>These SQL statements ...</th>
<th>Are not valid in this session mode ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>• BEGIN TRANSACTION</td>
<td>ANSI</td>
</tr>
<tr>
<td>• BT</td>
<td></td>
</tr>
<tr>
<td>• END TRANSACTION</td>
<td></td>
</tr>
<tr>
<td>• ET</td>
<td></td>
</tr>
<tr>
<td>COMMIT [WORK]</td>
<td>Teradata</td>
</tr>
</tbody>
</table>

Session Pool Manager

The Session Pool Manager permits setting the Teradata or ANSI session mode for pooled sessions.

Two-Phase Commit Protocol

The 2PC protocol (see Introduction to Teradata for a brief description of 2PC) is supported for Teradata session mode, but not for ANSI session mode.

If you attempt to use the 2PC protocol while in ANSI session mode, the logon process aborts and an error is reported.
ANSI Session Mode

Definition

ANSI session mode is a state in which transaction processing adheres to the rules defined by the ANSI SQL-2003 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 session modes.

Rules

The following rules are enforced in ANSI session mode:

- A transaction is initiated when:
  - No transaction is currently active.
  - A request is submitted.
- Transactions are always explicit.
  More accurately, each ANSI transaction is implicitly initiated, but always explicitly completed.

A transaction begins with the first request submitted in a session and continues until the system encounters either an explicit COMMIT statement or an explicit ROLLBACK statement, at which point it ends, releasing all the locks it held, discarding the Transient Journal (see “Transient Journal” on page 699), and closing any open cursors.

Statement failures do not cause a rollback of the transaction, only of the request that causes them.

If you do not submit an explicit COMMIT statement, then all requests submitted are considered to be a continuation of the same transaction, and no work is committed.

- A transaction is opened by the first request executed in a session or by the next request performed in the session following the close of a transaction.
- Either an explicitly submitted COMMIT statement or an explicitly submitted ROLLBACK statement terminates a transaction.

You can use the Teradata SQL ABORT statement in place of the ANSI SQL ROLLBACK statement. ABORT is a Teradata extension to the ANSI SQL-2003 standard.

- The most exclusive locks (READ, WRITE, EXCLUSIVE) are retained at the highest level (Rowhash and Table) and are not released until a transaction is committed.
- The system checks the number of control blocks used for Rowhash locks in a transaction and aborts it if the number exceeds the threshold set for the DBS Control flag MaxRowHashBlocksPercent (see Utilities for details).
- Each transaction consists of one or more requests, each of which can consist of one or more SQL statements (see “Transactions, Requests, and Statements” on page 702).
Chapter 9: Locking and Transaction Processing
ANSI Session Mode

- Multistatement requests are treated as a single atomic transaction; either all the work done by all the statements in a multistatement request is committed or none of it is (see “Transactions, Requests, and Statements” on page 702).

- Error responses (see SQL Fundamentals for information about success, warning, error, and failure responses) roll back only the request that evokes them and not the entire transaction.

This means that ANSI mode transactions are not universally atomic because they do not roll back the entire transaction when an error response occurs. As a result, they do not support the A property of ACID transactions (see “The ACID Properties of Transactions” on page 698) in all circumstances.

To ensure that your transactions are always handled as intended, it is critical to code your applications with logic to handle any situation that only rolls back an error-generating request rather than the entire transaction of which it is a member.

The system does not release any locks placed for a request that is rolled back because of an Error response.

- Only one DDL statement is permitted per transaction, and it must be the last action statement specified in the transaction.

If you attempt to perform another DDL statement before you issue a COMMIT statement, the system returns an Error for the non-valid statement, but does not roll back the transaction.

Although they are technically DCL statements, Teradata Database treats the DATABASE and SET SESSION statements as DDL statements for the purposes of handling transactions.

- If you log off prior to committing your work, then the system rolls back all your transaction requests.

- Control of character truncation of trailing non-blank characters causes errors.

Use the SUBSTRING function to prevent such errors (see SQL Functions, Operators, Expressions, and Predicates).

- 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.

- The two-phase commit (2PC) protocol is not valid (see Introduction to Teradata and “2PC” on page 851) for brief descriptions of 2PC).

Because only explicit transactions are valid, you cannot mix implicit and explicit transactions within the same script.
ANSI Mode Transaction Processing Case Studies

The following set of ANSI mode transactions provides case studies you can use to compare with analogous Teradata mode transactions (see “Teradata Mode Transaction Processing Case Studies” on page 785).

The set contains the following case studies:

- “ANSI Mode Successful Transaction” on page 779
- “ANSI Mode Errors and Failures” on page 780
- “ANSI Mode Transactions and DDL: DDL Request Placement Within the Transaction” on page 781
- “ANSI Mode Transactions and DDL: Multistatement Requests” on page 782
- “ANSI Mode DELETE Performance for Different Transaction Structures” on page 782

ANSI Mode Successful Transaction

The following transaction is an example of a successful ANSI mode transaction:

```
INSERT INTO employee
SELECT *
FROM employee;
*** Insert completed. 26 rows added.

UPDATE employee
SET department_number = 400
WHERE department_number = 401
;DELETE FROM employee
WHERE department_number = 401;
*** Update completed. 7 rows changes.
*** Delete completed. No rows removed.

SELECT *
FROM employee
WHERE department_number = 401;
*** Query completed. No rows found.

COMMIT;
*** COMMIT done.
*** Total elapsed time was 1 second.
```

This single statement/single request begins an ANSI mode explicit transaction.
WRITE locks are held on employee.

This multiresponse request is composed of two statements, but is a single request.
WRITE locks are still held on employee.

This single statement/single request is a check to ensure that all employees in department_number 401 were deleted in the previous request.
Notice that even though this is a SELECT statement, which only requires a READ (or ACCESS) lock, the transaction continues to hold a WRITE lock on employee.

This statement terminates the explicit transaction by committing all changes.
All locks are released and the Transient Journal is dropped from the dictionary.
The next request entered begins another explicit ANSI transaction.
ANSI Mode Errors and Failures

When ANSI transaction semantics are in effect, SQL Error responses do not cause a rollback, while Failure responses do.

The following example shows how Error responses do not roll back a transaction:

```
INSERT INTO employee
SELECT *
FROM customer_service.employee;
*** Insert completed. 26 rows added.

SELECT *
FROM employee;
*** Error 3706 Syntax error; SELECT * must have a FROM clause.

SELECT *
FROM employee;
*** Query completed. 26 rows found. 9 columns returned.

SELECT *
FROM employee
WHERE emp_num = 1010;
*** Error 5628 Column emp_num not found in Employee.

SELECT *
FROM employee;
*** Query completed. 26 rows found. 9 columns returned.

INSERT INTO employee
SELECT *
FROM customer_service.employee;
*** Insert completed. 26 rows added.

CREATE TABLE tbl_1 (
    col_1, 
    col_2 INTEGER);
*** Error 3739 The user must give a data type for .

SELECT *
FROM employee;
*** Query completed. 26 rows found. 9 columns returned.
```

This INSERT … SELECT request begins an ANSI mode explicit transaction.

Syntaxer problems are errors, not failures. The inserts from the previous statement remain, and all locks continue to be in force.

Correcting the SELECT syntax produces a successful statement that indicates the inserted rows from the first statement remain in place. The transaction is still not committed.

Resolver problems are neither always errors, nor always failures. In this case, the problem evokes an Error response, not a Failure response. The inserts from the first statement remain, and all locks continue to be in force.

Repeating the selection of all columns from the table proves that the inserted rows from the first statement remain. The transaction is still not committed.

This INSERT … SELECT request begins an ANSI mode explicit transaction.

Syntaxer problems are errors, not failures. The inserts from the previous statement remain, and all locks continue to be in force.

Selecting all columns from the table proves that the inserted rows from the first statement remain. The transaction is still not committed.
Like Teradata mode transactions, there can only be one DDL request in a transaction, and it must be the last sequential request.

Unlike Teradata mode transactions, an ANSI mode transaction does not roll back if you attempt to submit another DDL request before committing the transaction. Instead, it continues to respond with Error responses until the requestor either issues a COMMIT statement or a ROLLBACK/ABORT statement.
ANSI Mode Transactions and DDL: Multistatement Requests

Like Teradata mode transactions, you cannot mix DDL and DML statements with a single request in ANSI session mode.

The following macro and multistatement request, which have the identical semantics, both fail. An ANSI mode Failure response rolls back the transaction.

```
SELECT *
FROM table_1
;SELECT *
FROM table_2
;CREATE TABLE table_5 (
  col_1 INTEGER);

*** Failure 3576 Data definition not valid unless solitary.
Statement# 1, Info =0

CREATE MACRO mac_1 AS (
  SELECT *
  FROM table_1;
  SELECT *
  FROM table_2;
  CREATE TABLE table_5 (
    col_1 INTEGER);
);

*** Failure 3576 Data definition not valid unless solitary.
Statement# 1, Info =0
```

ANSI Mode DELETE Performance for Different Transaction Structures

Depending on how you structure a transaction that contains a DELETE statement, it can either create a Transient Journal entry for each row deleted from a table or create only one Transient Journal entry for the entire transaction.

The following DELETE statement is a single explicit transaction. It writes a Transient Journal entry for each deleted row, so its performance is poor, particularly for large tables.

```
DELETE FROM table_1;
COMMIT;
```

The following explicit transaction is not valid because the BEGIN TRANSACTION and END TRANSACTION statements are not legal in ANSI session mode.

```
BEGIN TRANSACTION;
DELETE FROM table_1;
END TRANSACTION;
```

The following multistatement request contains the same two statements as the first transaction, but because they are packaged as a multistatement request, they are treated as if it were an implicit transaction. The system does not write a Transient Journal entry for each row deleted from the table, and its performance is very good.

```
DELETE FROM table_1
;COMMIT;
```
Teradata Session Mode

Definition

Teradata, or BTET, session mode is a state in which transaction processing follows a set of rules defined by Teradata over years of evolution.

Teradata session 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 session modes.

Rules

The following rules are enforced in Teradata session mode:

- A transaction is initiated when no transaction is currently active and either of the following occurs:
  - An SQL request is executed. This is an implicit Teradata mode transaction.
  - A BEGIN TRANSACTION statement is encountered. This marks the beginning of an explicit Teradata mode transaction.

  Note that any explicit transaction proceeds until it encounters one of the following SQL statements:
  - END TRANSACTION
    In the case of nested transactions, it is not the first END TRANSACTION statement encountered that terminates the transaction, but the last.
  - ROLLBACK or ABORT

- Transactions can be implicit or explicit. Unless bounded by explicit BEGIN TRANSACTION (BT) and END TRANSACTION (ET) statements, the system treats each request as an implicit transaction (see “Transactions, Requests, and Statements” on page 702).

- Explicit transaction boundaries are specified using BEGIN TRANSACTION (BT) and END TRANSACTION (ET) statements (see SQL Data Manipulation Language for information about the BEGIN TRANSACTION and END TRANSACTION statements). An implicit transaction terminates when it either completes successfully (Success response) or causes a Failure response (see SQL Fundamentals for information about Success, Warning, Error, and Failure responses).

- When a transaction commits, the system discards its Transient Journal (see “Transient Journal” on page 699) and closes any open cursors.
When a transaction fails, the system first rolls it back automatically, discards its Transient Journal, and closes any open cursors. There is no need to perform an ABORT or ROLLBACK statement to make the rollback occur.

Statement Failure responses roll back the entire transaction, not just the request that evokes them.

The most exclusive locks (READ, WRITE, EXCLUSIVE) are retained at the highest level (Row hash, Table) and are not released until a transaction is committed.

The system checks the number of control blocks used for Rowhash locks in a transaction and aborts it if the number exceeds the threshold set for the DBS Control flag MaxRowHashBlocksPercent (see Utilities for details).

Each transaction consists of one or more requests, each of which can consist of one or more SQL statements (see “Transactions, Requests, and Statements” on page 702).

Error responses (see SQL Fundamentals for information about success, warning, error, and failure responses) evoked by a bad request roll back the entire transaction. The system releases all locks placed for a request that is rolled back.

Only one DDL statement is permitted per transaction, and it must be the last action statement specified in the transaction. If you attempt to perform another DDL statement before you commit the transaction, the system returns an Error for the non-valid statement, and then rolls back the transaction. Although they are technically DCL statements, Teradata Database treats the DATABASE and SET SESSION statements as DDL statements for the purposes of handling transactions.

Multistatement requests are treated as a single atomic transaction; either all the work done by all the statements in a multistatement request is committed or none of it is (see “Transactions, Requests, and Statements” on page 702).

If you log off prior to committing your work, then the system rolls back all your transaction requests.

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.

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.

The two-phase commit (2PC) protocol is valid (see Introduction to Teradata or “2PC” on page 851 for brief descriptions of 2PC).

Because both explicit and implicit transactions are valid, you can mix them within the same script.
Teradata Mode Transaction Processing Case Studies

The following set of Teradata mode transactions provides case studies you can use to compare with analogous ANSI mode transactions (see “ANSI Mode Transaction Processing Case Studies” on page 779).

The set contains the following case studies:

- “Teradata Mode Failure (ADD 3-23)” on page 785
- “Teradata Mode Requests (ADD 3-25 - 3-27)” on page 786
- “Mixing DDL and DML Within a Multistatement Request” on page 787
- “Teradata Mode DELETE Performance for Different Transaction Structures” on page 788

Teradata Mode Failure (ADD 3-23)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN TRANSACTION;</td>
<td>Beginning of an explicit transaction.</td>
</tr>
<tr>
<td>INSERT INTO employee SELECT * FROM customer_service.employee;</td>
<td>Single statement. Single request. WRITE locks held.</td>
</tr>
<tr>
<td>SELECT * FROM employee WHERE empnum = 401;</td>
<td>Syntax error is a failure. Transaction rolled back. All previous requests in transaction are also rolled back. All locks released.</td>
</tr>
<tr>
<td>SELECT * FROM employee;</td>
<td>Single statement. Single request. Implicit transaction.</td>
</tr>
<tr>
<td>END TRANSACTION;</td>
<td>Request to end the transaction causes a failure response because the transaction begun with the BEGIN TRANSACTION statement had already rolled back.</td>
</tr>
</tbody>
</table>

**BTEQ -- Enter your DBC/SQL request or BTEQ command:**

- BTEQ -- Enter your DBC/SQL request or BTEQ command: BEGIN TRANSACTION;
  *** Begin transaction accepted.***

- BTEQ -- Enter your DBC/SQL request or BTEQ command: INSERT INTO employee SELECT * FROM customer_service.employee;
  *** Insert completed. 26 rows added.***

- BTEQ -- Enter your DBC/SQL request or BTEQ command: SELECT * FROM employee WHERE empnum = 401;
  *** Failure 3706 Syntax error; SELECT * must have a FROM clause.***

- BTEQ -- Enter your DBC/SQL request or BTEQ command: SELECT * FROM employee;
  *** Query completed. No rows found.***

- BTEQ -- Enter your DBC/SQL request or BTEQ command: END TRANSACTION;
  *** Failure 3510 Too many END TRANSACTION statements.***
### Teradata Mode Requests (ADD 3-25 - 3-27)

<table>
<thead>
<tr>
<th>SQL Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DELETE FROM table_1 WHERE PI_col=2;</strong></td>
<td>These three statements are separate requests. They are also implicit transactions. The implications of this are:</td>
</tr>
<tr>
<td><strong>INSERT INTO table_1 VALUES (2,3,4);</strong></td>
<td>• The requests are performed serially in the order they are received.</td>
</tr>
<tr>
<td><strong>UPDATE table_1 SET col_3=4;</strong></td>
<td>• Their locks are applied and released separately.</td>
</tr>
<tr>
<td><strong>BEGIN TRANSACTION;</strong></td>
<td>• The success or failure of each has no effect on the success or failure of the others.</td>
</tr>
<tr>
<td><strong>DELETE FROM table_1 WHERE PI_col=2;</strong></td>
<td>These five statements are separate requests within a single explicit transaction. The implications of this are:</td>
</tr>
<tr>
<td><strong>INSERT INTO table_1 VALUES (2,3,4);</strong></td>
<td>• The requests are performed serially in the order they are specified in the transaction.</td>
</tr>
<tr>
<td><strong>UPDATE table_1 SET col_3=4;</strong></td>
<td>• Locks are held, and possibly upgraded, throughout the duration of the transactions, only being released when either an END TRANSACTION statement commits the work or a ROLLBACK or ABORT statement rolls back the work.</td>
</tr>
<tr>
<td><strong>END TRANSACTION;</strong></td>
<td>• The success or failure of each has a direct effect on the success or failure of the others.</td>
</tr>
<tr>
<td><strong>DELETE FROM table_1 WHERE PI_col=2;</strong></td>
<td>These three statements form a single multistatement request. They are also implicitly a single transaction. The implications of this are:</td>
</tr>
<tr>
<td><strong>INSERT INTO table_1 VALUES (2,3,4);</strong></td>
<td>• The most restrictive lock held by the transaction, a table-level WRITE lock, is applied to table_1.</td>
</tr>
<tr>
<td><strong>UPDATE table_1 SET col_3=4;</strong></td>
<td>• The work done by the request is atomic: either it is all committed or it is all rolled back.</td>
</tr>
<tr>
<td><strong>CREATE MACRO mac_1 AS (</strong></td>
<td>This macro contains three separate statements. Because they are contained within the same macro, they behave identically to a multistatement request that contains the same three statements in the same order.</td>
</tr>
<tr>
<td><strong>DELETE FROM table_1 WHERE PI_col=2;</strong></td>
<td>The result of executing the macro is atomic in exactly the same way its equivalent multistatement request is atomic.</td>
</tr>
<tr>
<td><strong>INSERT INTO table_1 VALUES (2,3,4);</strong></td>
<td>The EXPLAIN reports generated for these two requests are identical.</td>
</tr>
<tr>
<td><strong>UPDATE table_1 SET col_3=4;</strong></td>
<td><strong>EXPLAIN EXEC mac_1</strong></td>
</tr>
<tr>
<td><strong>EXEC mac_1</strong></td>
<td><strong>EXPLAIN DELETE FROM table_1 WHERE PI_col=2;</strong></td>
</tr>
<tr>
<td><strong>DELETE FROM table_1 WHERE PI_col=2;</strong></td>
<td><strong>INSERT INTO table_1 VALUES (2,3,4);</strong></td>
</tr>
<tr>
<td><strong>;INSERT INTO table_1 VALUES (2,3,4);</strong></td>
<td><strong>UPDATE table_1 SET col_3=4;</strong></td>
</tr>
</tbody>
</table>
Mixing DDL and DML Within a Multistatement Request

You cannot mix DDL and DML statements within the same macro or multistatement request. An attempt to perform such a request results in a failure response.

For example, the following multistatement request fails because it mixes DML (two SELECT statements) with DDL (a CREATE TABLE statement):

```
SELECT *
FROM table_1
;SELECT *
FROM table_1
;CREATE TABLE table_33 (col_1 INTEGER);
```

*** Failure 3576 Data definition not valid unless solitary. Statement#1, Info =0

The equivalent macro text fails with the same error at the time you attempt to create the macro, not when you attempt to execute it.

```
CREATE MACRO mac_1 AS (SELECT *
FROM table_1;
SELECT *
FROM table_1;
CREATE TABLE table_33 (col_1 INTEGER);
);
```

*** Failure 3576 Data definition not valid unless solitary. Statement#1, Info =0

If you include a DDL statement within a Teradata session mode transaction, it must be the last request in the transaction. If it is not, the transaction fails and all its work is rolled back. For example:

```
BEGIN TRANSACTION;

*** Begin transaction accepted.
BTEQ -- Enter your DBC/SQL request or BTEQ command:
CREATE TABLE table_19 (col_1 INTEGER);

*** Table has been created.
BTEQ -- Enter your DBC/SQL request or BTEQ command:
INSERT INTO table_3 VALUES (1);

*** Failure 3932 Only an ET or null statement is legal after a DDL statement.
BTEQ -- Enter your DBC/SQL request or BTEQ command:
SHOW TABLE table_19;

*** Failure 3807 Table/view/trigger/procedure ‘table_19’ does not exist.
```
Teradata Mode DELETE Performance for Different Transaction Structures

Depending on how you structure a transaction that contains a DELETE request, it can either create a Transient Journal entry for each row deleted from a table or create only one Transient Journal entry for the entire transaction.

The following DELETE request is a single implicit transaction. It does not write a Transient Journal entry for each deleted row, so its performance is quite good.

```
DELETE FROM table_1;
```

The following explicit transaction contains only BEGIN TRANSACTION and END TRANSACTION statements in addition to the DELETE request. Because of the way it is structured, it writes a Transient Journal entry for each deleted row and performs poorly, especially for large tables.

```
BEGIN TRANSACTION;
DELETE FROM table_1;
END TRANSACTION;
```

The following multistatement request contains the same three statements as the previous transaction, but because they are packaged as a multistatement request, they are treated as an implicit transaction. The system does not write a Transient Journal entry for each row deleted from the table, and its performance is identical to that of the single-statement implicit transaction version.

```
BEGIN TRANSACTION
;DELETE FROM table_1
;END TRANSACTION;
```
## Comparison of Transaction Rules in ANSI and Teradata Session Modes

The following table compares the transaction rules for ANSI and Teradata session modes:

<table>
<thead>
<tr>
<th>Rule</th>
<th>ANSI Session Mode</th>
<th>Teradata Session Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction declaration</td>
<td>Always implicit.</td>
<td>Either implicit or explicit.</td>
</tr>
<tr>
<td>Transaction initiation and</td>
<td>Transactions initiate when no transaction is active</td>
<td>Opened by one of the following:</td>
</tr>
<tr>
<td>termination</td>
<td>and an SQL request is performed.</td>
<td>• Each request performed in a session.</td>
</tr>
<tr>
<td></td>
<td>Opened by one of the following:</td>
<td>• BEGIN TRANSACTION statement.</td>
</tr>
<tr>
<td></td>
<td>• First SQL request performed in a session.</td>
<td>If requests are grouped into explicit transactions in this way, they must be</td>
</tr>
<tr>
<td></td>
<td>• First request performed after the previous</td>
<td>terminated by a matching END TRANSACTION statement.</td>
</tr>
<tr>
<td></td>
<td>transaction terminates.</td>
<td>Transaction rollbacks are specified by an ABORT or ROLLBACK statement.</td>
</tr>
<tr>
<td></td>
<td>In both cases, the transaction terminates with either a</td>
<td>The COMMIT statement is not valid.</td>
</tr>
<tr>
<td></td>
<td>COMMIT [WORK] or ROLLBACK [WORK] (or ABORT) statement.</td>
<td></td>
</tr>
<tr>
<td>Cursors</td>
<td>Always positioned.</td>
<td>Never positioned.</td>
</tr>
<tr>
<td>Error behavior</td>
<td>Errors roll back only the request that causes them,</td>
<td>Errors roll back the entire transaction.</td>
</tr>
<tr>
<td></td>
<td>not the entire transaction.</td>
<td>All locks are released and the Transient Journal is first applied, then deleted from</td>
</tr>
<tr>
<td></td>
<td>Locks placed by erroneous statements are not released</td>
<td>the dictionary.</td>
</tr>
<tr>
<td></td>
<td>and the Transient Journal is not deleted from the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dictionary.</td>
<td></td>
</tr>
<tr>
<td>Statement failure behavior</td>
<td>Not applicable.</td>
<td>Statement failures roll back the entire transaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All locks are released and the Transient Journal is first applied, then deleted from</td>
</tr>
<tr>
<td>Default attribute for character</td>
<td>CASESPECIFIC</td>
<td>the dictionary.</td>
</tr>
<tr>
<td>comparisons</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control of character truncation</td>
<td>Records errors for truncation problems with trailing</td>
<td>Truncation problems do not cause errors.</td>
</tr>
<tr>
<td>errors</td>
<td>non-blank characters.</td>
<td></td>
</tr>
</tbody>
</table>
### Comparison of Transaction Rules in ANSI and Teradata Session Modes

<table>
<thead>
<tr>
<th>Rule</th>
<th>ANSI Session Mode</th>
<th>Teradata Session Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default table type semantics</td>
<td>MULTISET</td>
<td>SET</td>
</tr>
<tr>
<td>Default TRIM function behavior</td>
<td>Trims both leading and trailing pad characters as if TRIM (BOTH FROM expression) were explicitly specified.</td>
<td>Trims only trailing pad characters as if TRIM(TRAILING FROM expression) were explicitly specified.</td>
</tr>
<tr>
<td>DDL statement placement</td>
<td>Must be last statement in transaction.</td>
<td>Must be last statement in transaction.</td>
</tr>
<tr>
<td>Statements performed through a logon startup string</td>
<td>Not applicable.</td>
<td>Must follow standard rules for:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Transaction definition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Transaction termination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDL statement placement</td>
</tr>
</tbody>
</table>

a. Because each statement is an implicit transaction, there is nothing to roll back if a statement fails.
Rollback Processing

Introduction
This section describes the differences in rollback processing for Teradata and ANSI session modes.

Application-Initiated Asynchronous Abort
An application-initiated asynchronous ABORT causes full transaction rollback in both session modes. This occurs by means of a CLlv2 abort request or by the TDP when the application terminates without proper session cleanup.

Teradata Session 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 session mode:

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).
4. End of process.

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.

34. The “application” in this case is a component of the database management system, whether client-based or server-based.
ANSI Session Mode

ANSI only recognizes termination of a transaction by the performance of a COMMIT [WORK] or ABORT/ROLLBACK [WORK] statement performed by the application. The system 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 is necessary before the locks held by the transaction can be released.
- Rejected because the request was blocked and had specified a LOCKING 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.

See Database Design for information about how to design your databases in ways that facilitate mixed tactical and decision support query workloads.

Group AMP Locking Considerations

Group AMP operations use a series of row hash-level locks, one for each of the rows touched by the query. See Database Design for more information about group AMP operations.

One of the major benefits of group AMP processing is that it reduces the need to apply table-level locks significantly. Table-level locking, because it is at a higher level than row hash-level locking, exposes database objects to greater contention, so it is more likely to slow, or even block, other concurrently running operations. The Optimizer also applies table-level locks in a separate step, and all AMPs in the system are included in the step that gets and applies table-level locks.

In contrast to these operations with a negative impact on performance, group AMP operations can improve conditions for both read and update concurrency because the locks they place are not only set at lower levels, but also in fewer places. The larger the number of AMPs in the configuration, the greater the performance benefit obtained from group AMP operations, and the more likely the Optimizer is to specify group AMP-based query steps.
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;
```
Locking Issues With Tactical Queries

### 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;
```

### 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 request 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 the unique primary index "`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 request modifier in their view definitions.

### Row Hash-Level ACCESS Locks and Join Indexes

The Optimizer propagates row hash-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 hash-level ACCESS lock be applied to the customer table.

```sql
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.

Row Hash-Level ACCESS Locks With Group AMP Steps

One of the key advantages of group AMP steps is that they avoid placing table-level locks.
Instead of placing a table-level lock, group AMP operations exert row hash-level locks on each
row hash in the AMP group. The same performance-enhancing lock-level substitution also
occurs when you explicitly request row hash-level locking by specifying a LOCKING ROW
FOR ACCESS modifier with your SQL statement, as demonstrated by the EXPLAIN report
generated for the following SELECT statement. Notice the row hash-level locks with ACCESS
severity being applied in step 2 to the rows of both tables that are accessed by the query:

```
EXPLAIN
LOCKING ROW FOR ACCESS
SELECT p_name, p_type
FROM lineitem, parttbl
WHERE l_partkey = p_partkey
AND  l_orderkey = 5;
```

Explanation

1) First, we do a single-AMP RETRIEVE step from CAB.lineitem by way
of the primary index "CAB.lineitem.L_ORDERKEY = 5" with no
residual conditions locking row for access into Spool 2
(group_amps), which is redistributed by hash code to all AMPs.
Then we do a SORT to order Spool 2 by row hash. The size of Spool
2 is estimated with low confidence to be 1 row. The estimated
time for this step is 0.01 seconds.

2) Next, we do a group-AMPs JOIN step from Spool 2 (Last Use) by way
of a RowHash match scan, which is joined to CAB.parttbl locking
row of CAB.parttbl for access. Spool 2 and CAB.parttbl are
joined using a merge join, with a join condition of ("L_PARTKEY =
CAB.parttbl.P_PARTKEY"). The result goes into Spool 1
(group_amps), which is built locally on that AMP. The size of
Spool 1 is estimated with low confidence to be 1 row. The
estimated time for this step is 0.11 seconds.

3) Finally, we send out an END TRANSACTION step to all AMPs involved
in processing the request.
ACCESS Locks With Joins

When a query for which a row hash-level ACCESS lock has been requested also includes a join operation, the Optimizer applies the row hash-level ACCESS lock request to both tables if both are eligible.

The Optimizer plan generated for the following query is single-AMP because both tables are joined on their common orderkey UPI columns:

```sql
EXPLAIN
LOCKING ROW FOR ACCESS
SELECT l_quantity, l_partkey, o_orderdate
FROM lineitem, ordertbl
WHERE l_orderkey = o_orderkey
AND o_orderkey = 832094;
```

Explanation

1) First, we do a single-AMP JOIN step from CAB.ordertbl by way of the unique primary index "CAB.ordertbl.O_ORDERKEY = 832094" with no residual conditions, which is joined to CAB.lineitem by way of the primary index "CAB.lineitem.L_ORDERKEY = 382855" locking row of CAB.ordertbl for access and row of CAB.lineitem for access. CAB.ordertbl and CAB.lineitem are joined using a merge join, with a join condition of ("CAB.lineitem.L_ORDERKEY = CAB.ordertbl.O_ORDERKEY"). The input tables CAB.ordertbl and CAB.lineitem will not be cached in memory, but CAB.ordertbl is eligible for synchronized scanning. The result goes into Spool 1(one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with low confidence to be 35 rows. The estimated time for this step is 0.03 seconds.

Row Hash-Level ACCESS Locks Are Compatible With Aggregates

If aggregation can be performed as a single-AMP operation, the Optimizer honors explicit row hash-level ACCESS lock requests. An example might be the following query, where l_orderkey is the NUPI of the lineitem table and only a single row hash needs to be locked to satisfy the request:

```sql
EXPLAIN
LOCKING ROW FOR ACCESS
SELECT l_quantity, COUNT(*)
FROM lineitem
WHERE l_orderkey = 382855
GROUP BY l_quantity;
```

Explanation

1) First, we do a single-AMP SUM step to aggregate from CAB.lineitem by way of the primary index "CAB.lineitem.L_ORDERKEY = 382855" with no residual conditions, and the grouping identifier in field 1029 locking row for access. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The input table will not be cached in memory, but it is eligible for synchronized scanning. The size of Spool 3 is estimated with low confidence to be 35 rows. The estimated time for this step is 0.03 seconds.

2) Next, we do a single-AMP RETRIEVE step from Spool 3 (Last Use) by way of the primary index "CAB.lineitem.L_ORDERKEY = 382855" into Spool 1 (one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with low confidence to be 35 rows. The estimated time for this step is 0.04 seconds.

3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
If you explicitly request a row hash-level ACCESS lock and the query performs all-AMP aggregations, then the Optimizer upgrades the lock to a table-level ACCESS lock.

EXPLAIN
LOCKING ROW FOR ACCESS
SELECT SUM (l_quantity), SUM (l_extendedprice), COUNT(*)
FROM lineitem;

Explanation
---------------------------------------------------------------
1) First, we lock CAB.lineitem for access.
2) Next, we do an all-AMPs SUM step to aggregate from
   CAB.lineitem by way of an all-rows scan with no residual
   conditions. 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. The size of
   Spool 3 is estimated with high confidence to be 1 row. The
   estimated time for this step is 36 minutes and 56 seconds.
3) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of
   an all-rows scan into Spool 1 (group_amps), which is built locally
   on the AMPs. The size of Spool 1 is estimated with high
   confidence to be 1 row. The estimated time for this step is 0.67
   seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.

Some query tools make it difficult to include an ACCESS lock modifier in SQL statements. If
this is a problem at your site, you can instead enforce ACCESS locking by placing the
LOCKING request modifier in views through which the queries access the base tables.

**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 following update operation is a more complex version of the update operation presented in “Locking for Updates” on page 799. The following query updates the same non-indexed column (o_orderpriority) by accessing a single row using a UPI (o_orderkey), but also includes a join to lineitem rows on their common orderkey value.

If a complex update is single-AMP operation and there is an equality condition on the UPIs common to both joined tables (o_orderkey and l_orderkey in the example), then the generated query plan specifies a performant single-AMP merge update using row hash-level locking, as you can see in Step 1 of the EXPLAIN report for the following UPDATE statement:

```sql
EXPLAIN UPDATE ordertbl
FROM lineitem
SET o_orderstatus = 'OK'
WHERE l_orderkey = o_orderkey
AND l_shipdate = o_orderdate
AND o_orderkey = 5;
```

Explanation

1) First, we do a Single AMP MERGE Update to CAB.ordertbl from CAB.lineitem by way of a RowHash match scan.
2) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

Some complex updates might involve multiple all-AMP steps, which necessitate table-level locking. This is not an important performance issue if you only occasionally perform this type of complex update. However, if you must perform a significant number of such operations, the effects of their all-AMP operations combined with the collateral table-level WRITE locks they require are likely to impair scalability as a function of the increasing volume of all-AMP requests.
## References

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
</table>
A more introductory text than the Gray-Reuter and Weikum-Vossen books. |
At one time, this was what the Gray-Reuter and Weikum-Vossen volumes are today. The book is now out of print, but is available as a PDF file as one of the PDF book files on DVD 2 of the *ACM SIGMOD Anthology Silver Edition*, which is available from the Association for Computing Machinery, Inc.  
You can write or call the ACM offices at the following address and phone number:  
Association for Computing Machinery  
1515 Broadway, New York, New York 10036  
USA  
1-800-342-6626 (USA and Canada) or +212-626-0500 (Global)  
You can also purchase the item online at [http://www.acm.org](http://www.acm.org) or [http://www.sigmod.org](http://www.sigmod.org).  
If your public or corporate library has the CD-ROM version of these proceedings, which is no longer available for purchase, you can find the book on CD-ROM 1 in Volume 4 of the standard edition of the *ACM SIGMOD Anthology*. |
Together with the Weikum-Vossen volume, the ultimate reference for all things related to database transaction processing. |
This volume is mathematically rigorous, but at the same time exceedingly clear in its presentation. Numerous clever geometric proofs and illustrations of the concepts are included. This book is *not* just for the mathematically inclined.  
The book is now out of print, but is fairly easy to locate using any of the various used book search engines available on the World Wide Web. |
## References

### General transaction processing references (continued)

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An important extension of Gray and Reuter (Gray wrote the foreword to this book), providing a wider and more up-to-date perspective, albeit covering its subject matter with less depth.</td>
</tr>
<tr>
<td></td>
<td>The authors characterize this book as an updating and expansion of Bernstein, Hadzilacos, and Goodman (1987) in terms of its scope, which is an accurate assessment. Unlike Gray and Reuter, Weikum and Vossen deal strictly with the theoretical bases of transaction processing and do not review commercially available transaction processors and the like. In this sense, the book complements, rather than supersedes, the Gray and Reuter text.</td>
</tr>
</tbody>
</table>

### Origins of the concept of transaction management in relational database management systems

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>These two papers form the basis of the recovery model for System R developed by Gray et al. (1981). They should not be read as individual contributions, but rather as a unit.</td>
</tr>
<tr>
<td></td>
<td>Gray and his colleagues review the transaction management system (they call it a recovery manager) they designed for the System R relational database management system prototype developed by the IBM Corporation.</td>
</tr>
<tr>
<td></td>
<td>The design of the transaction management system was based on the earlier work of Bjork (1973) and Davies (1973).</td>
</tr>
<tr>
<td>Topic</td>
<td>Reference</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Topic</td>
<td>Reference</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>A classic paper on the subject.</td>
</tr>
<tr>
<td></td>
<td>A survey paper on optimistic replication.</td>
</tr>
<tr>
<td></td>
<td>This is perhaps the only place you will ever see livelock mentioned.</td>
</tr>
<tr>
<td></td>
<td>The book is more important for its simple and elegant proof that 2PL is a sufficient condition to ensure serializability. It is not, however, for those who are not mathematically inclined, because it is written in the formal style of a textbook designed for a course in the theoretical foundations of computer science.</td>
</tr>
</tbody>
</table>
The Date critique of the use of consistency in the ACID initialism


Date argues that consistency of the database is trivial because if integrity constraints are always checked immediately whenever data is transformed, then a transaction must always transform one consistent state of the database into another consistent state.

Instead, he argues, database correctness is what should be sought, not consistency. Of course, it is not possible to enforce correctness between the real world meaning of the information in a database and its internal representation of that information because the database cannot know the truth of the external world, but only its own version of that truth. While correctness is obviously something to be desired, it cannot be taken as a property. As a result, if the C in ACID represents consistency, it is trivial and if it represents correctness, it is unenforceable.

Date summarizes his argument as follows: "Correct implies consistent (but not the other way around) and inconsistent implies incorrect (but not the other way around)—where by correct we mean the database is correct if and only if it fully reflects the true state of affairs in the real world" (emphasis in original).

Having taken this position, Date then redefines the C in ACID as follows: "Any given transaction transforms a correct state of the database into another correct state, without necessarily preserving correctness at all intermediate points."


Date defends the position developed in the 8th edition of his textbook that the classically assigned meaning for the C in ACID as consistency is both trivial and uninteresting. Further, he argues that the unit of integrity in transaction processing should be the statement (as Date uses the term in his argument, it refers to what Teradata calls a request), not the transaction. He then goes on to posit what he calls multiple assignment as a remedy for this situation. The example of multiple assignment that Date provides is isomorphic with what Teradata calls a multistatement request (see “Multistatement Requests” on page 703), a construct that is unique to Teradata Database.

Note that even though this column is dated in late 2003, it is actually a response to a query raised by a reader of the 8th edition of *An Introduction to Database Systems*, which is dated 2004. Textbooks are often printed with a publication date that postdates their actual release.
<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Jim Gray, “A Transaction Model,” in: <em>Source Lecture Notes In Computer Science, Vol. 85: Proceedings of the 7th Colloquium on Automata, Languages and Programming</em>. Berlin: Springer-Verlag, 1980. Describes and analyzes the problems of long lived transactions and proposes that, while such transactions are “sleeping” (not performing database updates), transaction managers should enhance concurrency by not allowing them to hold locks. This means that updates of uncommitted transactions would be visible to other transactions.</td>
</tr>
<tr>
<td></td>
<td>Alan Fekete, Dimitrios Liarokapis, Elizabeth O’Neil, Patrick O’Neil, and Dennis Shasha, “Making Snapshot Isolation Serializable,” <em>ACM Transactions on Database Systems</em>, 30(2):492-528, 2005. Analyzes static dependencies between concurrently running applications to determine the conditions under which so-called “snapshot isolation” is also serializable. The work undertaken in this research is the first step toward developing a utility or set of utilities that can be used to determine the circumstances under which serializability can be relaxed without exposing the database to corruption (see the topic “The critique of the ANSI SQL standard transaction isolation levels” on page 807 for an important reference regarding this line of research.</td>
</tr>
<tr>
<td>Topic</td>
<td>Reference</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
This paper closely examines the isolation levels scheme presented by the ANSI SQL standard and finds it to be both ambiguous and incomplete. The authors provide alternative formalizations of the levels based on the possible interpretations of the written specification, then provide examples of how the standard could be greatly improved.  
The paper is also the initial exposition of the snapshot isolation level first defined by O’Neil and O’Neil. |
On the use of the term SERIALIZABLE in the ANSI SQL standard, the authors write in a footnote on page 59, “SERIALIZABLE is not an appropriate key word here. Serializability is a property of the interleaved execution of a set of concurrent transactions, not a property of any individual transaction considered independently. A better key word might have been just ISOLATED (perhaps FULLY ISOLATED).”  
Generally speaking, Date and Darwen echo the criticisms of the isolation levels section of the ANSI SQL standard made by Berenson et al. (1995), adding a few wry observations of their own along the way. |
A PDF file of these notes is available at [http://www.cs.umb.edu/cs734/](http://www.cs.umb.edu/cs734/).  
The titles of these notes is somewhat misleading because their content is devoted entirely to issues of transaction processing in relational database management systems. If you have any interest at all in this subject, this is a fascinating read. |
The critique of the ANSI SQL standard
transaction isolation levels (continued)


Among myriad other interesting ideas presented in this grant proposal, the O’Neils point out that while ANSI defines several weak levels of isolation that are not serializable, they do not define when it is safe to use any of them, nor do they suggest even the general situations in which the weak isolation levels might be useful.

Quoting them, “…lower isolation levels are normally used only in applications where they will not lead to inconsistent results: we want to use the lower isolation level and get better concurrency, but still avoid concurrency errors in the application. Clearly a certain amount of analysis of the application by a DBA is needed to guarantee this.

“But absolutely no support is given the DBA as to how to perform such analysis.”

The question is this: is it possible to tell if an application will run error-free under a given transaction isolation level? Unfortunately, the answer to this question is no.

Part of what the O’Neils propose is to develop a utility that DBAs could use to perform these analyses. Quoting them, “Our main deliverable will be a prototype mechanism for testing a database application to determine what errors, if any, will arise when the application is executed at various lower isolation levels.”
<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The paper that first established the basic concepts underlying all ensuing research on concurrency control.</td>
</tr>
<tr>
<td></td>
<td>Shows that blocking due to lock conflicts is the factor that imposes the upper bound on transaction throughput.</td>
</tr>
<tr>
<td></td>
<td>The implication of this is that, at least in some cases, non-locking optimistic concurrency control methods are more likely to increase throughput than they are to decrease it due to the necessity of aborting transactions that are found to produce anomalous outcomes.</td>
</tr>
<tr>
<td></td>
<td>The title says it all.</td>
</tr>
<tr>
<td></td>
<td>This book reorganizes and consolidates the research reported in two refereed journal articles (including the paper by Tay, Goodman, and Suri (1985)) and a Harvard University technical report from the mid-1980s, all of which were revisions of various parts of a Ph.D. dissertation submitted to Harvard University by the author.</td>
</tr>
<tr>
<td></td>
<td>The material is highly quantitative, but clearly reported and summarized.</td>
</tr>
<tr>
<td></td>
<td>Reviews a plethora of quantitative data about concurrency control. The coverage is not restricted to relational database management applications.</td>
</tr>
<tr>
<td></td>
<td>The first report of the potential for blocking due to lock contention to cause “thrashing,” or what the authors refer to as “the convoy phenomenon.”</td>
</tr>
<tr>
<td>Topic</td>
<td>Reference</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>The original paper to propose OCC as an alternative to locking mechanisms for transaction management.</td>
</tr>
<tr>
<td></td>
<td>A downward revision of the general applicability of OCC for transaction management from the IBM laboratory that had originally proposed the concept.</td>
</tr>
<tr>
<td></td>
<td>The first important paper to point out many of the problems with OCC in a realistic production environment.</td>
</tr>
<tr>
<td></td>
<td>Extends the observations of Haerder (1984) and introduces several additional concerns with OCC.</td>
</tr>
<tr>
<td></td>
<td>For anyone wondering what Mohan’s first name is, the riddle is solved by this explanation at his personal IBM web site:</td>
</tr>
<tr>
<td></td>
<td><strong>The first name puzzle:</strong> Those of you who are wondering what my first name is, please don’t worry, just call me Mohan. That is what my family calls me and that is the name my parents gave me, even though, given the way I write my name, Mohan appears to be my family name. In my part of India, there is no concept of a family name! We use as an initial the first letter of the father’s name or, in the case of a married woman, the husband’s name. In my case, I use “C” as an initial since my father’s name is Chandrasekaran.”</td>
</tr>
</tbody>
</table>
### References

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
</table>
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 813.
- 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 819.

Symbols from the predicate calculus are also used occasionally to describe logical operations. See page 817.

**Syntax Diagram Conventions**

**Notation Conventions**

<table>
<thead>
<tr>
<th>Item</th>
<th>Definition / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letter</td>
<td>An uppercase or lowercase alphabetic character ranging from A through Z.</td>
</tr>
<tr>
<td>Number</td>
<td>A digit ranging from 0 through 9. Do not use commas when typing a number with more than 3 digits.</td>
</tr>
</tbody>
</table>
Appendix A: Notation Conventions
Syntax Diagram Conventions

<table>
<thead>
<tr>
<th>Item</th>
<th>Definition / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>Keywords and variables.</td>
</tr>
<tr>
<td></td>
<td>- UPPERCASE LETTERS represent a keyword.</td>
</tr>
<tr>
<td></td>
<td>Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase.</td>
</tr>
<tr>
<td></td>
<td>- lowercase letters represent a keyword that you must type in lowercase, such as a UNIX command.</td>
</tr>
<tr>
<td></td>
<td>- lowercase italic letters represent a variable such as a column or table name. Substitute the variable with a proper value.</td>
</tr>
<tr>
<td></td>
<td>- lowercase bold letters represent an excerpt from the diagram. The excerpt is defined immediately following the diagram that contains it.</td>
</tr>
<tr>
<td></td>
<td>- UNDERLINED LETTERS represent the default value.</td>
</tr>
<tr>
<td></td>
<td>This applies to both uppercase and lowercase words.</td>
</tr>
<tr>
<td>Spaces</td>
<td>Use one space between items such as keywords or variables.</td>
</tr>
<tr>
<td>Punctuation</td>
<td>Type all punctuation exactly as it appears in the diagram.</td>
</tr>
</tbody>
</table>

**Paths**

The main path along the syntax diagram begins at the left with a keyword, 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.

**Continuation Links**

Paths that are too long for one line use continuation links. Continuation links are circled 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 reading.
**Required Entries**

Required entries appear on the main path:

![Diagram](SHOW FE0CA003)

If you can choose from more than one entry, the choices appear vertically, in a stack. The first entry appears on the main path:

![Diagram](SHOW CONTROLS VERSIONS FE0CA005)

**Optional Entries**

You may choose to include or disregard optional entries. Optional entries appear below the main path:

![Diagram](SHOW CONTROLS FE0CA004)

If you can optionally choose from more than one entry, all the choices appear below the main path:

![Diagram](READ SHARE ACCESS JC01A010)

Some commands and statements treat one of the optional choices as a default value. This value is UNDERLINED. It is presumed to be selected if you type the command or statement without specifying one of the options.

**Strings**

String literals appear in single quotes:

```
'msgtext'
```

![Diagram](JC01A004)
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:

Read loops from right to left.

The following conventions apply to loops:

<table>
<thead>
<tr>
<th>IF...</th>
<th>THEN...</th>
</tr>
</thead>
<tbody>
<tr>
<td>there is a maximum number of entries allowed</td>
<td>the number appears in a circle on the return path.</td>
</tr>
<tr>
<td></td>
<td>In the example, you may type <code>cname</code> a maximum of 4 times.</td>
</tr>
<tr>
<td>there is a minimum number of entries required</td>
<td>the number appears in a square on the return path.</td>
</tr>
<tr>
<td></td>
<td>In the example, you must type at least three groups of column names.</td>
</tr>
<tr>
<td>a separator character is required between entries</td>
<td>the character appears on the return path.</td>
</tr>
<tr>
<td></td>
<td>If the diagram does not show a separator character, use one blank space.</td>
</tr>
<tr>
<td></td>
<td>In the example, the separator character is a comma.</td>
</tr>
</tbody>
</table>
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 each side of the break. The name for the excerpted piece appears between the terminators in boldface type.

The boldface excerpt name and the excerpted phrase appears immediately after the main diagram. The excerpted phrase starts and ends with a plain horizontal line:

```
LOCKING | excerpt | A
A
HAVING con
where_cond
  ,
cname
  ,
col_pos
```

Multiple Legitimate Phrases

In a syntax diagram, it is possible for any number of phrases to be legitimate:

```
| dbname
| DATABASE
| tname
| TABLE
| VNAME
```

In this example, any of the following phrases are legitimate:

- `dbname`
- `DATABASE dbname`
- `tname`
Appendix A: Notation Conventions
Syntax Diagram Conventions

- TABLE tname
- vname
- VIEW vname

Sample Syntax Diagram

Diagram Identifier

The alphanumeric string that appears in the lower right corner of every diagram is an internal identifier used to catalog the diagram. The text never refers to this string.
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 *International Character Set Support*.

Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Encoding</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a–z</td>
<td>Any</td>
<td>Any single byte Latin letter or digit.</td>
</tr>
<tr>
<td>A–Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0–9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ぁ–ぞ</td>
<td>Unicode compatibility zone</td>
<td>Any fullwidth Latin letter or digit.</td>
</tr>
<tr>
<td>ぁ–ぞ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ｧ–ｦ</td>
<td>KanjiEBCDIC</td>
<td>Shift Out [SO] (0x0E).</td>
</tr>
<tr>
<td>ｧ–ｦ</td>
<td></td>
<td>Indicates transition from single to multibyte character in KanjiEBCDIC.</td>
</tr>
<tr>
<td>&lt;</td>
<td>KanjiEBCDIC</td>
<td>Shift In [SI] (0x0F).</td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td>Indicates transition from multibyte to single byte KanjiEBCDIC.</td>
</tr>
<tr>
<td>T</td>
<td>Any</td>
<td>Any multibyte character.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The encoding depends on the current character set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For KanjiEUC, code set 3 characters are sometimes preceded by “ss3”.</td>
</tr>
</tbody>
</table>
Appendix A: Notation Conventions
Character Shorthand Notation Used In This Book

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.

<table>
<thead>
<tr>
<th>Server Character Set</th>
<th>Pad Character Name</th>
<th>Pad Character Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LATIN</td>
<td>SPACE</td>
<td>0x20</td>
</tr>
<tr>
<td>UNICODE</td>
<td>SPACE</td>
<td>U+0020</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>IDEOGRAPHIC SPACE</td>
<td>U+3000</td>
</tr>
<tr>
<td>KANJI SJIS</td>
<td>ASCII SPACE</td>
<td>0x20</td>
</tr>
<tr>
<td>KANJI H1</td>
<td>ASCII SPACE</td>
<td>0x20</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>This symbol ...</th>
<th>Represents this phrase ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>iff</td>
<td>If and only if</td>
</tr>
<tr>
<td>∀</td>
<td>For all</td>
</tr>
<tr>
<td>∃</td>
<td>There exists</td>
</tr>
</tbody>
</table>
APPENDIX B  Content and Structure of XML Documents Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

This appendix contains the XML Plan Schema for the XML document, an example XML Optimizer Query Plan document produced by the XMLPLAN option for BEGIN QUERY LOGGING, and a sample Java program that can be used to process XML documents using the XML Schema.
The following table documents the information that is recorded in the XML plan produced by the BEGIN QUERY LOGGING … XMLPLAN option and the INSERT EXPLAIN and EXPLAIN … IN XML option.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query/@TimeStamp</td>
<td>Date and time when plan was logged.</td>
</tr>
<tr>
<td>Query/@DefaultDBName</td>
<td>Name of database where request was executed.</td>
</tr>
<tr>
<td>Query/@MachName</td>
<td>Name of system.</td>
</tr>
<tr>
<td>Query/@NumAMPs</td>
<td>Number of AMPs in system.</td>
</tr>
<tr>
<td>Query/@NumNodes</td>
<td>Number of nodes in system.</td>
</tr>
<tr>
<td>Query/@NumPEs</td>
<td>Number of PEs in system.</td>
</tr>
<tr>
<td>Query/@PENum</td>
<td>PE that processed request.</td>
</tr>
<tr>
<td>Query/@QueryID</td>
<td>System wide unique query identifier.</td>
</tr>
<tr>
<td>Query@QueryText</td>
<td>Text of request.</td>
</tr>
<tr>
<td>Query@ReleaseInfo</td>
<td>Database release information.</td>
</tr>
<tr>
<td>Query@StatementType</td>
<td>Type of statement executed.</td>
</tr>
<tr>
<td>Query@VersionInfo</td>
<td>Database version information.</td>
</tr>
<tr>
<td>QuerySteps/@Cardinality</td>
<td>Estimated cardinality of step.</td>
</tr>
<tr>
<td>QuerySteps/@Cost</td>
<td>Estimated cost of step.</td>
</tr>
<tr>
<td>QuerySteps/@EstCPUCost</td>
<td>Estimated CPU cost of step.</td>
</tr>
<tr>
<td>QuerySteps/@EstHRCost</td>
<td>Estimated heuristic cost of step.</td>
</tr>
<tr>
<td>QuerySteps/@EstIOCost</td>
<td>Estimated IO cost of step.</td>
</tr>
<tr>
<td>QuerySteps/@EstNetworkCost</td>
<td>Estimated network cost of step.</td>
</tr>
<tr>
<td>QuerySteps/@LockType</td>
<td>Severity of lock.</td>
</tr>
<tr>
<td>QuerySteps/@NoWaitFlag</td>
<td>Indicates whether the lock request should block.</td>
</tr>
<tr>
<td>QuerySteps/@ParallelKind</td>
<td>Indicates whether the step is a parallel step.</td>
</tr>
<tr>
<td>QuerySteps/@ParallelStepNum</td>
<td>Parallel step number.</td>
</tr>
<tr>
<td>QuerySteps/@RowHashFlag</td>
<td>Indicates whether the lock is a row hash lock.</td>
</tr>
<tr>
<td>QuerySteps/Relation</td>
<td>Source and target relations involved in step.</td>
</tr>
<tr>
<td>QuerySteps/@StepKind</td>
<td>Kind of step.</td>
</tr>
<tr>
<td>QuerySteps/@StepNum</td>
<td>Step number.</td>
</tr>
</tbody>
</table>
### Appendix B: Content and Structure of XML Documents Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

#### XML Plan Contents

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuerySteps/@StepText</td>
<td>Step text.</td>
</tr>
<tr>
<td>QuerySteps/@TriggerType</td>
<td>Describes trigger associated with step.</td>
</tr>
<tr>
<td>User_Database/@UDB_Name</td>
<td>Database name.</td>
</tr>
<tr>
<td>Relation/@Cached</td>
<td>Indicates whether relation is cached.</td>
</tr>
<tr>
<td>Relation/@Cardinality</td>
<td>Estimated cardinality of table.</td>
</tr>
<tr>
<td>Relation/@Confidence</td>
<td>Confidence level of estimated cardinality.</td>
</tr>
<tr>
<td>Relation/@GeogInfo</td>
<td>Geography of relation.</td>
</tr>
<tr>
<td>Relation/@Name</td>
<td>Table name.</td>
</tr>
<tr>
<td>Relation/@PartitionInfo</td>
<td>Indicates whether table has a partitioned primary index or not.</td>
</tr>
<tr>
<td>Relation/@RelationKind</td>
<td>Type of table.</td>
</tr>
<tr>
<td>Relation/@SortInfo</td>
<td>Indicates whether the relation is sorted or not.</td>
</tr>
<tr>
<td>Relation/@SortKey</td>
<td>Columns that make up the sort key.</td>
</tr>
<tr>
<td>Relation/@SortKind</td>
<td>Indicates the type of table.</td>
</tr>
<tr>
<td>Relation/@SpoolCompressedAllowed</td>
<td>Indicates whether spool is allowed to contain compressed columns.</td>
</tr>
<tr>
<td>Relation/@SpoolSize</td>
<td>Estimated spool size.</td>
</tr>
<tr>
<td>Relation/@SyncScan</td>
<td>Indicates whether table is eligible for synchronized scanning.</td>
</tr>
<tr>
<td>Relation/@TableName</td>
<td>Table name.</td>
</tr>
<tr>
<td>Relation/@Version</td>
<td>Table version.</td>
</tr>
<tr>
<td>Field/@FldAlias</td>
<td>Field alias.</td>
</tr>
<tr>
<td>Field/@JoinAccessFrequency</td>
<td>Number of times column is used in a join in the request.</td>
</tr>
<tr>
<td>Field/@Name</td>
<td>Field name.</td>
</tr>
<tr>
<td>Field/@RangeAccessFrequency</td>
<td>Number of times column is used in a range condition.</td>
</tr>
<tr>
<td>Field/@FieldID</td>
<td>Field ID.</td>
</tr>
<tr>
<td>Field/@ValueAccessFrequency</td>
<td>Number of times column is used in a value condition.</td>
</tr>
<tr>
<td>Predicate/@PredicateKind</td>
<td>Type of predicate.</td>
</tr>
<tr>
<td>Predicate/@PrediateText</td>
<td>Text of predicate condition.</td>
</tr>
<tr>
<td>IndexTable/@IndexName</td>
<td>Index name.</td>
</tr>
<tr>
<td>IndexTable/@IndexNum</td>
<td>Index number.</td>
</tr>
<tr>
<td>IndexTable/@IndexType</td>
<td>Type of index.</td>
</tr>
<tr>
<td>IndexTable/@OrderBy</td>
<td>Indicates whether index is ordered.</td>
</tr>
<tr>
<td>IndexTable/@RangeConstraint</td>
<td>Indicates whether plan has a range constraint on the value ordered index.</td>
</tr>
</tbody>
</table>
## XML Plan Contents

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexTable/@UniqueFlag</td>
<td>Indicates whether the index is unique or not.</td>
</tr>
<tr>
<td>IndexTable/@AccessInfo</td>
<td>Indicates whether the index access is covering or bitmap.</td>
</tr>
<tr>
<td>IndexTable/@Field1Only</td>
<td>Indicates whether the index is a join index and Field1 is the only part needed.</td>
</tr>
</tbody>
</table>
This section documents the content and structure of XML documents produced by the XMLPLAN option for BEGIN QUERY LOGGING using the standard W3C XML Schema language.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation>W3C Schema for logged or captured Teradata query Plans</xs:documentation>
  </xs:annotation>
  <xs:element name="QryPlanXML">
    <xs:annotation>
      <xs:documentation>This is the root element that can contain one or more nested queries.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Query" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Query">
    <xs:annotation>
      <xs:documentation>There is one Query element per logged query. The QueryID attribute value can be used to retrieve any additional information logged in DBQL tables for this query (or alternatively to retrieve any additional captured information in QCD tables).</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Plan" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="PlanObjects" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="QueryID" type="xs:int"/>
      <xs:attribute name="User_DatabaseId" type="xs:IDREF"/>
      <xs:attribute name="MachName" type="xs:string"/>
      <xs:attribute name="NumAMPs" type="xs:int"/>
      <xs:attribute name="NumPEs" type="xs:int"/>
      <xs:attribute name="NumNodes" type="xs:int"/>
      <xs:attribute name="ReleaseInfo" type="xs:string"/>
      <xs:attribute name="VersionInfo" type="xs:string"/>
      <xs:attribute name="PENum" type="xs:int"/>
      <xs:attribute name="QueryText" type="xs:string"/>
      <xs:attribute name="DateTimeStamp" type="xs:date"/>
      <xs:attribute name="StatementType" type="xs:string"/>
      <xs:attribute name="DefaultDBName" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
Appendix B: Content and Structure of XML Documents Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

XML Schema Used for the XMLPLAN Option

```xml
<xs:element name="Plan">
    <xs:annotation>
        <xs:documentation>
            The Plan element represents the Optimizer's execution plan including QuerySteps, Predicates, and the objects they reference.
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="QuerySteps" minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="QuerySteps">
    <xs:annotation>
        <xs:documentation>
            Each QuerySteps element references Relations and potentially applies Predicates. Each referenced Relation is represented by a nested RelationRef element whose AccessKind attribute value denotes if it's a Source (S) or Target (T). Nested StepAttribute elements may be contained each representing additional information about a step. Nested IndexRef elements represent the use of one or more Indexes during the execution of the step.
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="StepAttribute" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="RelationRef" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="Predicate" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="IndexRef" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="StepNum" type="xs:int"/>
        <xs:attribute name="ParallelStepNum" type="xs:int" default="0"/>
        <xs:attribute name="StepKind" type="xs:string"/>
        <xs:attribute name="ParallelKind" type="xs:string" default="S"/>
        <xs:attribute name="AmpUsage" type="xs:string" default="A"/>
        <xs:attribute name="TriggerType" type="xs:string" default="N"/>
        <xs:attribute name="EstCPUCost" type="xs:float" default="0"/>
        <xs:attribute name="EstIOCost" type="xs:float" default="0"/>
        <xs:attribute name="EstNetworkCost" type="xs:float" default="0"/>
        <xs:attribute name="EstHRCost" type="xs:float" default="0"/>
        <xs:attribute name="Cost" type="xs:float" default="0"/>
        <xs:attribute name="LockType" type="xs:string"/>
        <xs:attribute name="RowHashFlag" type="xs:string" default="F"/>
        <xs:attribute name="NoWaitFlag" type="xs:string"/>
        <xs:attribute name="Cardinality" default="0">
            <xs:simpleType>
                <xs:restriction base="xs:decimal">
                    <xs:totalDigits value="18"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:element>
```
Appendix B: Content and Structure of XML Documents Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

XML Schema Used for the XMLPLAN Option

```xml
<xs:fractionDigits value="0"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:element name="StepAttribute">
  <xs:annotation>
    <xs:documentation>
      The StepAttribute element provides additional information about a given QueryStep including index access methods and join algorithms.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="StepAttributeType" type="xs:string"/>
    <xs:attribute name="StepAttributeValue" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:element name="Predicate">
  <xs:annotation>
    <xs:documentation>
      A Predicate element represents a query condition that references fields and potentially uses an Index(s) to evaluate the condition.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="FieldRef" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="IndexRef" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="PredicateKind" type="xs:string"/>
    <xs:attribute name="PredicateText" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:element name="RelationRef">
  <xs:annotation>
    <xs:documentation>
      The 'Ref' attribute of the RelationRef element serves as a link to the 'Id' attribute of the Relation element. The default kind of relation is S(ource).
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="Ref" type="xs:IDREF" use="required"/>
    <xs:attribute name="AccessKind" type="xs:string" default="S"/>
  </xs:complexType>
</xs:element>
<xs:element name="IndexRef">
  <xs:annotation>
    <xs:documentation>
      The 'Ref' attribute of the IndexRef element serves as a link to the 'Id' attribute of the Index element.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="Ref" type="xs:IDREF" use="required"/>
  </xs:complexType>
</xs:element>
```

SQL Request and Transaction Processing
Appendix B: Content and Structure of XML Documents Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

XML Schema Used for the XMLPLAN Option

```xml
<xs:complexType>
  <xs:element name="FieldRef">
    <xs:annotation>
      <xs:documentation>
        The 'Ref' attribute of the FieldRef element serves as a link to the 'Id' attribute of the Field element.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="Ref" type="xs:IDREF" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PlanObjects">
    <xs:annotation>
      <xs:documentation>
        The PlanObjects element contains the detailed definition of each Database, Relation, Index, and Field referenced in the plan. Each object should be defined only once within PlanObjects but can be referenced multiple times within the Plan element.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="User_Database" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="Relation" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="Index" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="User_Database">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Relation" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Id" type="xs:ID" use="required"/>
      <xs:attribute name="UDB_Name" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Relation">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="Index" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Id" type="xs:ID" use="required"/>
      <xs:attribute name="User_DatabaseId" type="xs:IDREF" use="required"/>
      <xs:attribute name="Name" type="xs:string"/>
      <xs:attribute name="RelationKind" type="xs:string" default="P"/>
    </xs:complexType>
  </xs:element>
</xs:complexType>
```
Appendix B: Content and Structure of XML Documents Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

XML Schema Used for the XMLPLAN Option

```xml
<xs:schema>
  <xs:element name="SqlRequest">
    <xs:complexType>
      <xs:attribute name="SortInfo" type="xs:string" default="F"/>
      <xs:attribute name="SortKind" type="xs:string"/>
      <xs:attribute name="SortKey" type="xs:string"/>
      <xs:attribute name="GeomInfo" type="xs:string" default="I"/>
      <xs:attribute name="Cached" type="xs:string" default="F"/>
      <xs:attribute name="SyncScan" type="xs:string" default="T"/>
      <xs:attribute name="Cardinality" type="xs:float" default="0"/>
      <xs:attribute name="Confidence" type="xs:string"/>
      <xs:attribute name="MaxCardinality" type="xs:float" default="0"/>
      <xs:attribute name="ViewName" type="xs:string"/>
      <xs:attribute name="TableName" type="xs:string"/>
      <xs:attribute name="PartitionInfo" type="xs:string" default="F"/>
      <xs:attribute name="Version" type="xs:short"/>
      <xs:attribute name="SpoolCompressedAllowed" type="xs:string" default="N"/>
      <xs:attribute name="SpoolSize" type="xs:float"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Field">
    <xs:complexType>
      <xs:attribute name="Id" type="xs:ID" use="required"/>
      <xs:attribute name="RelationId" type="xs:IDREF" use="required"/>
      <xs:attribute name="FieldID" type="xs:int"/>
      <xs:attribute name="Name" type="xs:string"/>
      <xs:attribute name="FldAlias" type="xs:string"/>
      <xs:attribute name="ValueAccessFrequency" type="xs:int" default="0"/>
      <xs:attribute name="JoinAccessFrequency" type="xs:int" default="0"/>
      <xs:attribute name="RangeAccessFrequency" type="xs:int" default="0"/>
      <xs:attribute name="ChangeRate" type="xs:int" default="0"/>
      <xs:attribute name="DataLength" type="xs:int"/>
      <xs:attribute name="StatsKind" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Index">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="FieldRef" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Id" type="xs:ID" use="required"/>
      <xs:attribute name="RelationId" type="xs:IDREF" use="required"/>
      <xs:attribute name="IndexNum" type="xs:int"/>
      <xs:attribute name="OrderBy" type="xs:string" default="F"/>
      <xs:attribute name="AccessInfo" type="xs:string" default="N"/>
      <xs:attribute name="Field1Only" default="F"/>
      <xs:attribute name="RangeConstraint" type="xs:string" default="F"/>
      <xs:attribute name="IndexFlag" type="xs:string" default="F"/>
      <xs:attribute name="IndexName" type="xs:string"/>
      <xs:attribute name="IndexType" type="xs:string"/>
      <xs:attribute name="UniqueFlag" type="xs:string" default="F"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
Example XML Optimizer Query Plan Document Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

This section provides a sample XML plan document produced by the XMLPLAN option for BEGIN QUERY LOGGING. The corresponding query text and EXPLAIN output are also shown.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<QryPlanXML xsi:noNamespaceSchemaLocation="C:/v2r13/xmlspy/hybrid_schema.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Query  QueryID="1" User_DatabaseId="localhost_DBC" MachName="localhost" NumAMPs="2" NumPEs="1" NumNodes="1" ReleaseInfo="12v.00.00.00 " VersionInfo=" 12v.00.00.00" PENum="16383" QueryText="select employee.name, department.deptname, salary from department, employee where department.deptno = employee.deptno and employee.yrsexp &gt;=5 order by 3 desc;" DateTimeStamp="2007-10-01" Frequency="1" StatementType=" RET" DefaultDBName="PERSONNEL">
    <Plan>
      <QuerySteps StepNum="1" NoWaitFlag="1"  StepText=" 1) First, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to prevent global deadlock for PERSONNEL.department. " StepKind="LK">
        <QuerySteps StepNum="2" StepText=" 2) Next, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to prevent global deadlock for PERSONNEL.employee. " StepKind="LK">
          <QuerySteps StepNum="3" StepText=" 3) We lock PERSONNEL.department for read, and we lock PERSONNEL.employee for read. " StepKind="LK" LockType="R" NoWaitFlag="F">
            <RelationRef Ref="REL1" AccessKind="S"/>
          </QuerySteps>
          <QuerySteps StepNum="4" StepText=" 4) We do an all-AMPs RETRIEVE step from PERSONNEL.employee by way of an all-rows scan with a condition of ("employee.YrsExp &gt;= 5) AND((PERSONNEL.employee.DeptNo &lt;= 900) AND ((PERSONNEL.employee.DeptNo &gt;= 100) AND (NOT (PERSONNEL.employee.DeptNo IS NULL )))) into Spool 2(all_amps), which is redistributed by the hash code of (PERSONNEL.employee.DeptNo) to all AMPs. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated with noconfidence to be 3 rows (75 bytes). The estimated time for this step is 0.01 seconds. " StepKind="SR" ParallelKind="S" AmpUsage="A" EstIOCost="10.18" EstNetworkCost="0.04" Cost="10.23">
              <RelationRef Ref="REL2" AccessKind="S"/>
              <RelationRef Ref="REL3" AccessKind="T"/>
              <Predicate PredicateKind="S"/>
            </QuerySteps>
          </QuerySteps>
        </QuerySteps>
      </QuerySteps>
    </Plan>
  </Query>
</QryPlanXML>
```
PredicateText="(PERSONNEL.employee.YrsExp &gt;= 5) AND ((PERSONNEL.employee.DeptNo &lt;=900) AND ((PERSONNEL.employee.DeptNo &gt;=100) AND (NOT(PERSONNEL.employee.DeptNo IS NULL )))">
  <FieldRef Ref="REL2_1027"/>
  <FieldRef Ref="REL2_1030"/>
</Predicate>
</QuerySteps>
</QuerySteps>
<QuerySteps StepNum="5" StepText="5) We do an all-AMPS JOIN step from Spool 2 (Last Use) by way of a RowHash match scan, which is joined to PERSONNEL.department by way of a RowHash match scan with no residual conditions. Spool 2 and PERSONNEL.department are joined using a merge join, with a join condition of ("PERSONNEL.department.DeptNo = DeptNo"). The result goes into Spool 1 (group_amps), which is built locally on the AMPS. Then we do a SORT to order Spool 1 by the sort key in spool field1(PERSONNEL.employee.Salary). The size of Spool 1 is estimated with no confidence to be 3 rows (171 bytes). The estimated time for this step is 0.05 seconds." StepKind="MJ" AmpUsage="A"
EstiIOCost="50.14" Cost="50.15">
  <StepAttribute StepAttributeType="JoinType" StepAttributeValue="I"/>
  <RelationRef Ref="REL3" AccessKind="S"/>
  <RelationRef Ref="REL1" AccessKind="S"/>
  <RelationRef Ref="REL4" AccessKind="T"/>
  <Predicate PredicateKind="J" PredicateText="PERSONNEL.department.DeptNo = DeptNo ">
    <FieldRef Ref="REL1_1025"/>
    <FieldRef Ref="REL3_1027"/>
  </Predicate>
</QuerySteps>
</QuerySteps>
<QuerySteps StepNum="6" StepText="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 0.06 seconds." StepKind="MS">
</QuerySteps>
</Plan>
</PlanObjects>
<User_Database Id="localhost_PERSONNEL" MachineName="localhost"
UDB_Name="PERSONNEL">
  <Relation Id="REL1" User_DatabaseId="localhost_PERSONNEL"
Name="department" RelationKind="P" Cardinality="4" Confidence="L"
TableName="DEPARTMENT" Version="1">
    <Field Id="REL1_1026" RelationId="REL1" FieldID="1026"
Name="DeptName" DataLength="14"/>
    <Field Id="REL1_1025" RelationId="REL1" FieldID="1025"
Name="DeptNo" JoinAccessFrequency="1" DataLength="2"/>
  </Relation>
  <Relation Id="REL2" User_DatabaseId="localhost_PERSONNEL"
Name="employee" RelationKind="P" Cardinality="18" Confidence="L"
TableName="EMPLOYEE" Version="1">
    <Field Id="REL2_1025" RelationId="REL2" FieldID="1025"
Name="EmpNo" DataLength="2"/>
    <Field Id="REL2_1026" RelationId="REL2" FieldID="1026" Name="Name"
DataLength="12"/>
  </Relation>
</User_Database>
Name="DeptNo" JoinAccessFrequency="1" RangeAccessFrequency="2"
DataLength="2"/>
    </Field>
  </Index>
  <Index Id="REL2_4" RelationId="REL2" IndexNum="4"
IndexFlag="F" IndexType="S">
    <FieldRef Ref="REL2_1026"/>
  </Index>
</Relation>
</User_Database>
</PlanObjects>
</Query>
</QryPlanXML>

select employee.name, department.deptname, salary
from department, employee
where department.deptno = employee.deptno and employee.yrsexp >= 5 order by 3 desc;

Explanations

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 a condition of "((PERSONNEL.employee.YrsExp >= 5) AND ((PERSONNEL.employee.DeptNo <= 900) AND ((PERSONNEL.employee.DeptNo >= 100) AND (NOT (PERSONNEL.employee.DeptNo IS NULL)))))" into Spool 2 (all_amps), which is redistributed by the hash code of (PERSONNEL.employee.DeptNo) to all AMPs. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated with no confidence to be 3 rows (75 bytes). The estimated time for this step is 0.01 seconds.
5) We do an all-AMPS JOIN step from Spool 2 (Last Use) by way of a RowHash match scan, which is joined to PERSONNEL.department by way of a RowHash match scan with no residual conditions. Spool 2 and PERSONNEL.department are joined using a merge join, with a join condition of ("PERSONNEL.department.DeptNo = DeptNo") . The result goes into Spool 1 (group_amps), which is built locally on the AMPS. Then we do a SORT to order Spool 1 by the sort key in spool field1 (PERSONNEL.employee.Salary). The size of Spool 1 is estimated with no confidence to be 3 rows (171 bytes). The estimated time for this step is 0.05 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 0.06 seconds.
Sample Java Program For Processing XML Documents Like Those Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

This topic demonstrates how a client can process an XML document that uses the schema defined in “XML Schema Used for the XMLPLAN Option” on page 827. A sample Java program is provided that uses the standard Document Object Model (DOM) to traverse the document like a tree structure. The program traverses the tree nodes that represent the query plan and generates EXPLAIN request modifier-like output that describes the plan in English. The program uses the Apache Software Foundation Xerces parser.

Note that using a standard Document Object Model parser such as Xerces within Java is just one of many possible methods that clients can use to access and parse XML documents generated by Teradata Database. For example, you can access and manipulate the XML by using SQL functions offered by Teradata XML Services (Watzke, 2003). You can choose the method or technology for processing XML documents that is most appropriate for your environment.

```java
/**
 * DOMClient.java
 *
 * Example program demonstrating DOM API usage on XML documents representing
 * a logged or captured Teradata query plan.
 */

// use the Apache Software Foundation's Xerces parser
import org.apache.xerces.parsers.*;

// import the DOM interfaces
import org.w3c.dom.*;
import org.xml.sax.*;

// get the necessary java support classes
import java.io.*;
import java.util.*;

public class DOMClient {

    /* Main function that allows this class to be invoked from the command line
    * with the names of one or more XML documents as arguments.
    */
    public static void main(String[] args) {
        DOMClient dc = new DOMClient();
        try {
            for (int i = 0; i < args.length; i++) {
                dc.parseDocTree(args[i]);
            }
        } catch (Exception e) {
            System.err.println(e);
        }
    }

    // Empty constructor that allows this object to be instantiated. */
    public DOMClient() {
    }

    /* Function that parses the document and displays
    * query plan information. */
    public void parseDocTree(String strXMLSysID) throws IOException, SAXException {
        // create a new parser
    }
} /* DOMClient.java */
```
DOMParser dp = new DOMParser();

// turn on schema validation
dp.setFeature("http://apache.org/xml/features/validation/schema",true);

// parse the document and get the DOM Document interface
dp.parse(strXMLSysID);
Document doc = dp.getDocument();

// get the root element
Element root = doc.getDocumentElement();

// get the first query and print its text
NodeList nl = root.getElementsByTagName("Query");
Element query = (Element) nl.item(0);
System.out.println(query.getAttribute("QueryText");

// get this query's plan element
nl = query.getElementsByTagName("Plan");
Element plan = (Element) nl.item(0);

// Iterate thru the plan steps
NodeList steps = plan.getChildNodes();
for (int ii = 0; ii < steps.getLength(); ii++)
{
    if (steps.item(ii).getNodeType() != Node.ELEMENT_NODE)
        continue;

    Element step = (Element) steps.item(ii);

    // Display basic step information
    System.out.println("Step Number "+ step.getAttribute("StepNum") + " is of type " + step.getAttribute("StepKind") + " and has a cost of " + step.getAttribute("Cost") + " milliseconds");
    text = step.getAttribute("StepText");
    System.out.println("Step Text: "+ (text.length() > 80 ? text.substring(0,80) : text));

    // Display info about Source relation(s) accessed in this step
    NodeList RelRefs = step.getElementsByTagName("RelationRef");
    for (int jj = 0; jj < RelRefs.getLength(); jj++)
    {
        Element RelRef = (Element) RelRefs.item(jj);
        if (!RelRef.getAttribute("AccessKind").equals("S"))
            continue;

        // fetch the associated relation object element
        Element rel = doc.getElementById(RelRef.getAttribute("Ref"));
        if (rel == null)
            System.err.println("Error: call to getElementById with Relation Ref " + RelRef.getAttribute("Ref") + " failed");

        // get the parent database for this relation
        Element db = (Element)rel.getParentNode();
        if (db == null)
            System.err.println("Error: relation " + rel.getAttribute("Name") + " has no parent db");

        System.out.println("Step accesses source relation " + db.getAttribute("UDB_Name") + "." + rel.getAttribute("Name"));
    }

    // Display info for each Predicate applied on this step
    NodeList preds = step.getElementsByTagName("Predicate");
    for (int jj = 0; jj < preds.getLength(); jj++)
    {
        Element pred = (Element) preds.item(jj);
        System.out.println("Predicate of kind " + pred.getAttribute("PredicateKind") + " applied to this step");
        text = pred.getAttribute("PredicateText");
        System.out.println("Predicate Text: " + (text.length() > 80 ? text.substring(0,80) : text));

        // Display each field referenced in the current predicate
        NodeList FldRefs = pred.getElementsByTagName("FieldRef");
        for (int kk = 0; kk < FldRefs.getLength(); kk++)
        {

        }
select employee.name, department.deptname, salary from department, employee where department.deptno = employee.deptno and employee.yrsexp >=5 order by 3 desc;
Appendix B: Content and Structure of XML Documents Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

Sample Java Program For Processing XML Documents Like Those Produced by the XMLPLAN Option for BEGIN QUERY LOGGING

SQL Request and Transaction Processing 839

Step Number 1 is of type LK and has a cost of 0.0E1 milliseconds
Step Text: 1) First, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to

Step Number 2 is of type LK and has a cost of 0.0E1 milliseconds
Step Text: 2) Next, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash top

Step Number 3 is of type LK and has a cost of 0.0E1 milliseconds
Step Text: 3) We lock PERSONNEL.department for read, and we lock PERSONNEL.employee for re
Step accesses source relation PERSONNEL.department

Step Number 4 is of type SR and has a cost of 10.23 milliseconds
Step Text: 4) We do an all-AMPs RETRIEVE step from PERSONNEL.employee by way of an all-row
Step accesses source relation PERSONNEL.employee
Predicate of kind S applied to this step
Predicate Text: (PERSONNEL.employee.YrsExp >= 5) AND ((PERSONNEL.employee.DeptNo <=900) AND ((PE
Predicate applied to Fields: employee.DeptNo,employee.YrsExp,
Predicate uses no indexes
Step populates target relation DBC.SPOOL with 3 rows
The resulting spool geography is of type H with a sort kind of RHS

Step Number 5 is of type MJ and has a cost of 50.15 milliseconds
Step Text: 5) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of a RowHash match
Step accesses source relation DBC.SPOOL
Step accesses source relation PERSONNEL.department
Predicate of kind J applied to this step
Predicate Text: PERSONNEL.department.DeptNo = DeptNo
Predicate applied to Fields: department.DeptNo,SPOOL.DeptNo,
Predicate uses no indexes
Step populates target relation DBC.SPOOL with 3 rows
The resulting spool geography is of type L with a sort kind of F1S

Step Number 6 is of type MS and has a cost of 0.0E1 milliseconds
Step Text: 6) Finally, we send out an END TRANSACTION step to all AMPs involved in processi
Related Topics

The following topics and documents are related to the material covered by this appendix:

- “XMLQCD” on page 631
- “BEGIN QUERY LOGGING” in SQL Data Definition Language
- “INSERT EXPLAIN” in SQL Data Manipulation Language
- “EXPLAIN” in SQL Data Manipulation Language
This appendix is a compilation of the individual reference sections for the topics of query processing and transaction processing in Chapter 2: “Query Rewrite and Optimization” and Chapter 9: “Locking and Transaction Processing,” respectively.

The two topics are presented in separate reference lists as follows:

- “Query Processing References” on page 842
- “Transaction Processing References” on page 847

There is a very slight overlap between the contents of the two lists.
Appendix C: References
Query Processing References


Available as a free PDF file from the following URL: http://www.ecp6.jussieu.fr/pageperso/bondy/books/gtwa/pdf/GTWA.pdf.


Note that the PDF version of the book cannot be printed.


Ahmad Ghazal and Thu Pham, *Teradata Block Optimization*, Teradata Database Orange Book 541-0003787.


Waqar Hasan, Optimization of SQL Queries for Parallel Machines, Berlin: Springer-Verlag, 1996.


Rama Krishna Korlapati, Optimizer Cardinality Estimation Improvements, Teradata Database Orange Book 541-0006588.


Bill McKenna and Ahmad Ghazal, Query Rewrite, Teradata Database Orange Book 541-0006382.


Tony Rollins and Ranjan Priyadarshi, OCES Type 2 Costing, Teradata Database Orange Book 541-0006384.


Appendix C: References

Transaction Processing References


**2PC**  Two-Phase Commit

A method of ensuring that updates in a distributed database management system either commit to all target nodes in the transaction or all rollback.

See *Introduction to Teradata* for details.

**2PL**  Two-Phase Locking

A method of locking database objects that ensures serializability, thus preserving the consistency of the database. The two phases are the growing phase, during which all locks on database objects are acquired, and the shrinking phase, during which those locks are dropped.

See “Two-Phase Locking” on page 706 for details.

**ACCESS Lock**  Permits a user to have READ access to an object that might already be locked for READ or WRITE. An ACCESS lock does not restrict access by another user except when an EXCLUSIVE lock is required; therefore it is sometimes referred to as a *dirty READ lock*.

A user requesting an ACCESS lock 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.

The Archive/Recovery utility can also place a HUT ACCESS Lock, or ROW RANGE Lock on a database resource.

ACCESS locks are the least restrictive locks available for use on a Teradata system.

See “Teradata Database Locking Levels and Severities” on page 723 for details.

**ACID**  Atomicity, Consistency, Isolation, Durability. The four fundamental properties all transactions in a relational database management system must possess.

See “The ACID Properties of Transactions” on page 698 for details.

**ACM**  Association for Computing Machinery (http://www.acm.org)

The leading US-based professional society for computer professionals.

**Alternate Key**  Any Candidate Key for a table that is not selected to be the Primary Key.

**AMP**  Access Module Processor vproc

The set of software services that controls the file system and data management components of a Teradata Database.

**AMP Steps**  A set of machine language directives to the AMP software derived primarily from the Optimizer White Tree.
ANSI  American National Standards Institute (http://wwwansi.org)
A US-based umbrella standards organization based in Washington, D.C., that defines,
certifies, and administers the SQL standard.
The ANSI SQL standards are available for purchase at the following web site: http://
webstoreansi.org/default.aspx.
The ANSI SQL standard is also recognized by the ISO.

API  Application Programming Interface
A set of software services with a well-defined program interface.

Arity  The number of columns in a relation.
Arity is a synonym for Degree.

ASCII  American Standard Code for Information Interchange
A standard seven-bit code designed to establish compatibility between various types of data
processing equipment. Originally proposed in 1963, ASCII is documented by the following
The standard ASCII character set defines 128 decimal numbers ranging from 0 through 127,
inclusive. The individual characters are assigned to alphanumerics, punctuation marks, and a
set of commonly used special characters.
There is also an extended ASCII character set consisting of an additional 128 decimal numbers
ranging from 128 through 255, inclusive. These characters are assigned to additional special,
mathematical, graphic, and “foreign” characters.
Because ASCII uses only 7 bits, it is possible to use the 8th bit for parity checking.
Compare with EBCDIC.

AWP  AMP Worker Task

Bidirectional Inheritance  The property of base tables and their underlying indexes being
able to inherit and use existing statistics from one another when either database object in a
pair has no existing interval histogram statistics.
If both database objects have existing interval histogram statistics, the Optimizer uses the set
with the more recent collection timestamp.
See “Statistical Inheritance” on page 216 for details.

Black Tree  Synonym for SynTree. See “Syntaxer” on page 30 for details.

BLOB  Binary Large Object
A data object, usually larger than 64K, that contains only binary data such as pictures, movies,
or music.
Compare with CLOB.
**Blocked Request**  A request that needs to access a database resource that is currently locked by another request that is imposing a lock that prevents that access. As a result, the system places the request in an I/O wait state until the lock on the resource it needs is released.

Note that blocked requests do not time out.

Compare with “Deadlock” on page 856. See “Blocked Requests” on page 746 for details.

**BNF**  Backus-Naur Form or Backus Normal Form

A metalanguage invented by John Backus of IBM and simplified by Peter Naur of Regnecentralen, the Danish computing institute, that is used to express context-free grammars. In other words, BNF is a formal way to describe formal languages. In this specific case, the formal language being described is the computer programming language SQL.

**BTEQ**  Basic Teradata Query facility.

A Teradata request and script processing facility based on the CLIv2 API.

See Basic Teradata Query Reference for details.

**BTET**  BTET represents BEGIN TRANSACTION … END TRANSACTION, the SQL statements that must delimit all explicit transactions in Teradata session mode. You can optionally code transaction initiation and termination explicitly in Teradata session mode, but not in ANSI session mode. In ANSI session mode, all transactions begin implicitly, but must be terminated explicitly.

BTET is also an option for the Preprocessor2 TRANSACT and -tr commands signifying that the transaction protocol the application is to use is Teradata session mode (see Teradata Preprocessor2 for Embedded SQL Programmer Guide for details).

**BYNET**  Banyan Network - The high speed interconnect channel between the nodes of a Teradata MPP system.

**Candidate Key**  Any key for a relational table that is eligible to be selected as its Primary Key. Also see Alternate Key.

**Cardinality**  The number, or estimated number, of rows in a relation. The relation can be a base table, a materialized global temporary table, a volatile table, a spool file, or an index subtable.

**CHECKSUM Lock**  Placed in response to a user-specified LOCKING FOR CHECKSUM request modifier (see SQL Data Manipulation Language) 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.

**CJK**  Chinese, Japanese, and Korean

A common abbreviation used to represent the multibyte character sets used to write the Chinese, Japanese, and Korean languages.
**CLIv2** Call Level Interface Version 2.

The Teradata API for presenting SQL requests to Teradata Database and receiving their results. See *Teradata Call-Level Interface Version 2 Reference for Channel-Attached Systems* or *Teradata Call-Level Interface Version 2 Reference for Network-Attached Systems* for details.

**CLOB** Character Large Object

A data object, usually larger than 64K, that contains only character data such as XML or other text files. Compare with BLOB.

**Column Correlation** The degree to which the values in one column depend on the values from another column in the same table.

Column correlation is something different than the correlation used in statistical analysis, where it is a bivariate measure of the association between two variables. Statistical correlation ranges between -1.00 and 1.00, with -1.00 defining a perfect negative linear relationship, 1.00 defining a perfect positive linear relationship, and 0.00 defining no relationship between the two variables.

In contrast to the statistical definition, column correlation captures the mapping relationship of values from two columns.

**Compressed Interval Histogram** A family of histograms that combines high-biased intervals with equal-height intervals. The name of this family of histograms is somewhat misleading because unlike the other histogram types, there is no similarly named interval (because its intervals are a mix of high-biased and 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 \( n + 1 \) in a compressed histogram.

See “Types of Interval Histograms Used By Teradata Database” on page 166 for details.

**Compression** The term compression is used to mean two entirely different things for the Teradata system. Both forms are lossless, meaning that the original data can be reconstructed exactly from their compressed forms.

When describing compression of hash and join indexes, compression refers to a logical row compression in which multiple sets of nonrepeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set.

When describing compression of column values, compression refers to the storage of those values one time only in the table header, not in the row itself, and pointing to them by means of an array of presence bits in the row header. The method is called Dictionary Indexing, and it can be viewed as a variation of Run-Length Encoding.

See *Database Design* for details about both types of compression.
**Concrete Steps**  The steps created when Optapply binds USING request modifier data parcels into the Plastic Steps produced by the Generator.

See “Definition: Concrete Steps” on page 69 for details.

**Condition**  See Predicate.

**Connecting Term**  A connecting term is a predicate that connects a query and its immediate subquery. Connecting terms have the following additional properties:

- They can specify any comparison operator.
- The specified comparison operator can be qualified by an ANY or ALL qualifier.
- If the comparison operator is qualified by ANY or ALL, then the subquery can return multiple rows.
- If the comparison operator is *not* qualified by ANY or ALL, then the subquery can return only a single row.

In the following example, \( y_1 = \text{ANY} \left( \text{SELECT} \ y_2 \ \text{FROM} \ t_2 \right) \) is the connecting term:

\[
\text{SELECT} \ x_1 \ \text{FROM} \ t_1 \ \text{WHERE} \ y_1 = \text{ANY} \left( \text{SELECT} \ y_2 \ \text{FROM} \ t_2 \right);
\]

**Correlating Term**  A correlating term is a predicate that references an outer table from a subquery.

In the following example, \( x_1 = x_2 \) is the correlating term. Column \( x_1 \) correlates with table \( t_1 \) in the outer (main) query block, and column \( x_2 \) correlates with table \( t_2 \) in the inner (subquery) query block:

\[
\text{SELECT} \ * \ \text{FROM} \ t_1 \ \text{WHERE} \ y_1 \ \text{IN} \left( \text{SELECT} \ y_2 \ \text{FROM} \ t_2 \ \text{WHERE} \ x_1 = x_2 \right);
\]

**Cost Profile**  A cost profile is a relocatable container of cost prediction method parameters. Cost profile specifications are stored in the dictionary, and consist of two kinds of specifications:

- A cost profile *type specification* is a list of parameter specifications.
- A cost profile *instance specification* of that type is a list of parameter values assigned to a subset of those parameters.

Two system tables, DBC.CostProfileTypes and DBC.ConstantDefs, are used for type definitions and another two system tables, DBC.CostProfiles and DBC.ConstantValues, are used for instance definitions.

See “Cost Optimization” on page 267 for details.

**Cover**  A condition in which all the column data requested by a query can be obtained by index-only access, typically using a hash, join, or nonunique secondary index to cover the query.
**Deadlock**  Consider two requests, 1 and 2, that are running concurrently. Suppose both requests need to lock database resource A and database resource B. If 1 locks B before 2, and 2 locks A before 1, neither request 1 nor request 2 can access the other resource it needs, and both must wait for the lock on the resource they need to access to be released before they can proceed. The more colorful term for this situation is *deadly embrace*, but it is more technically referred to as a deadlock.

Deadlock is resolved in Teradata Database by aborting the most recently submitted, or younger, request, and rolling back its updates. Teradata Database does not detect HUT locks; instead, the system returns error code 2631 to the application, so your code must be written to detect error code 2631 and if it is found, to resubmit the rolled back transaction.

Compare with **Blocked Request**.

See “**Deadlock**” on page 753 for details.

**Degree**  The number of attributes, or columns, in a relation. If a relation has a degree of 5, then it has 5 columns.

Degree is a synonym for **Arity**.

**Delayed Partition Elimination**  Partition elimination can occur with conditions comparing a partitioning column to a USING variable or built-in function. This cannot be done when building a plan that is cached, because a cached plan needs to be general enough to handle changes in these values in subsequent executions.

In certain cases, partition elimination can be delayed until the finalized plan is built from a cached plan using the values for this specific execution of the plan, and this deferred partition elimination is called delayed partition elimination.

Delayed Partition Elimination is similar to **Static Partition Elimination** in that it is performed by the Optimizer, but it is done at a later stage of the optimization process.

Delayed Partition Elimination is unlike **Dynamic Partition Elimination** because it is done as part of the optimization process undertaken by the Optimizer, not done post optimization by the AMP-based database software.

Also see **Partition Elimination**.

See “**Delayed Partition Elimination**” on page 307 for details.

**Derived Statistics**  Statistics that are transformed from various constraint sources, including query predicates, and then adjusted dynamically at each stage of the query optimization process. Derived Statistics are propagated from optimization stage to optimization stage by means of a data structure that contains both the relevant static interval histogram statistics and the dynamically adjusted derived statistics.

See “**Derived Statistics**” on page 206 for details.
**Direct Join**  A binary join operation for which the relation of interest is not spooled in preparation for the join.

For example, a direct Merge Join is a join in which the relation of interest is not spooled in preparation for a Merge Join. Similarly, a direct Product Join is a Product Join in which the relation of interest is not spooled in preparation for a Product Join.

**Dispatcher**  The PE query processing component that takes the concrete steps produced by OptApply and creates AMP steps, which it then passes across the BYNET for processing by an AMP set.

The Dispatcher also manages any abort processing that might be required, as well as receiving the results of the AMP Steps from the BYNET and returning them to the requesting application.

See “Dispatcher” on page 70 for details.

**DOM**  Document Object Model.

The platform- and language-neutral de facto W3C standard object model used to represent text that is structured using XML and similar markup language formats.

**Dynamic Partition Elimination**  A form of dynamic reoptimization performed by the AMP software when query conditions reference values in other tables that permit partition elimination, but which cannot be determined at the time a query is initially optimized.

See Partition Elimination, Delayed Partition Elimination, and Static Partition Elimination. Also see “Partition Elimination” on page 298 for details.

**E2I**  External-to-Internal

**EBCDIC**  Extended Binary-Coded Decimal Interchange Code

An 8-bit code for alphanumerics, punctuation marks, and special characters devised by IBM Corporation as an alternative to ASCII. EBCDIC and ASCII use different coding schemes to define their respective character sets, and EBCDIC defines some special characters that are not defined in ASCII. EBCDIC is only used by IBM computing equipment.

Because EBCDIC is an 8-bit coding scheme, it is not possible to perform parity checks using the 8th bit.

Compare with ASCII.

**Eliminated Partition**  A PPI partition that is skipped for a particular query because the Optimizer has determined that it contains no qualifying rows.

See Partition Elimination.

**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 rows as a function of interval number is flat.

This is achieved by varying the width of each interval so it contains approximately the same number of rows (but usually different attribute value ranges) as its neighbors.
Also known as an equal-depth interval.
See “Types of Interval Histograms Used By Teradata Database” on page 166 for details.

**Equal-Height Interval Histogram** A family of histograms characterized by approximately equal cardinalities and a variable attribute value range per bucket.
See “Types of Interval Histograms Used By Teradata Database” on page 166 for details.

**EquiSet** In join planning, an EquiSet is a set of columns that was equated in predicates from a previous join. Propagating EquiSets to subsequent join operations in a query for reuse is part of the “Derived Statistics” framework.
See “Using Join Predicate Redundancy After Each Binary Join” on page 228 for details.

**EUC** Extended UNIX Code
The EUC code set uses control characters to identify characters in some of the character sets. The encoding rules are based on the ISO-2022 definition for the encoding of 7-bit and 8-bit data. The EUC code set uses control characters to separate some of the character sets.
The various UTF-n formats are defined partly by the EUC standard and partly by the various parts of ISO standard ISO-8859.

**EVL** EVaLuator Machine. The EVL (pronounced evil) machine is a code set that behaves like a pseudocomputer that processes a defined set of pseudoinstructions. It takes the White Tree produced by the Optimizer and converts it to an EVL tree, which can then either be interpreted directly or used to generate machine code that can be assembled to produce assembled EVL code.

**EXCLUSIVE Lock** 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 the Archive/Recovery utility, in which case a HUT EXCLUSIVE Lock is placed on the resource (see Teradata Archive/Recovery Utility Reference for details).
An EXCLUSIVE lock restricts access to the object by any other user and is the most restrictive lock available for use on a Teradata system.
See “Teradata Database Locking Levels and Severities” on page 723 for details.

**FIFO** First-In-First-Out
A type of queue in which the first entries placed in the sequence are also the first read from it.

**Foreign Key** A means of establishing referential integrity between tables in a relational database. A foreign key in a child table is typically the logical primary key of its parent table. If it is not the primary key for the parent table, then it is one of its alternate keys.
Compare with Primary Key.
See Database Design for details.
FTS  See Full-Table Scan.

Full-Table Scan  A situation in which indexes are not used to access table values. Instead, every row of the specified table is touched during the table scan.

Full table scans are used whenever a request condition set does not specify an indexed column and whenever statistics are collected.

Future Date  When a row containing a date value is inserted into a table after date statistics have been collected on that table, the date value is said to be a future date. Note that this cannot be a true future date; it is only futuristic with respect to when the date statistics were last collected on the particular date column. This means that “future” dates range from the date statistics were last collected on the date column through the current date, inclusive.

If such a date value is not part of a composite column or index statistics set, the Optimizer can use extrapolation to estimate its current statistics; however, it cannot apply date extrapolation for multicolunn or multicolunn index statistics.

See “Using Extrapolation to Replace Stale Statistics” on page 254 for details.

Generator  The PE query processing component that takes the White Tree produced by the Optimizer and creates plastic steps, which it then passes to Steps Packaging.

Plastic steps are, except for statement literals, a data-free skeletal tree of AMP directives derived from the white tree. The completed plastic steps tree is then passed to Steps Packaging for further processing.

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 (see “USING Request Modifier” in SQL Data Manipulation Language), then that data is also added to the steps tree. The final product of this process is referred to as plastic steps. The plastic steps are then passed to OptApply for further processing.

See “Generator” on page 66 for details.

Generic Plan  An optimizer plan that is based on not having peeked at parameterized USING values during query parsing.

Compare with Specific Plan.

See “Peeking at the Request Cache” on page 36 for further information.

Global Index  A join index (see Join Index) defined with the ROWID keyword to reference the corresponding base table rows.

See Database Design for details.

GROUP READ Lock  A special type of HUT READ lock, implemented internally as a ROW RANGE Lock that permits other users to update a table during an all-AMP or cluster-level Archive operation on tables that have an after-image journal.
To do this, the Archive/Recovery utility does a moving read on the subject table with the following process stages:

1. Place a HUT ACCESS lock at the table level.
2. Read a small group of rows (roughly 64 Kbytes) using a HUT READ lock.
3. Archive the HUT READ-locked rows.
4. Release the HUT READ lock on those rows.
5. Place a HUT READ lock on the next group of rows.
6. Repeat stages 2 through 5 until the Archive operation is completed.

Note that the utility releases the intermediate HUT READ locks, but you must still release the GROUP READ lock using a RELEASE LOCK Archive command in order to release the table-level HUT ACCESS lock set in stage 1.

Also see ROW RANGE Lock and HUT GROUP READ Lock.

Hash Index  A vertical partition of a base table having properties similar to a single-table Join Index.

Unlike the primary index, which is stored in-line with the row it indexes, hash indexes are stored in separate subtables that must be maintained by the system. Hash index subtables also consume disk space, so you should monitor your queries periodically using EXPLAIN modifiers to determine whether the Optimizer is using any of the hash indexes you designed for them. If not, you should either drop those indexes or rewrite your queries in such a way that the Optimizer does use them.

See Database Design for details.

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 85) is summarized by a high-biased interval.

Each high-biased interval contains statistics for at most two attribute row values.

See “Types of Interval Histograms Used By Teradata Database” on page 166 for details.

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.

See “Types of Interval Histograms Used By Teradata Database” on page 166 for details.

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 number of rows 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 relational query optimization theory, the term is used to describe the rows in a dictionary table or system catalog that store the buckets defined by the particular intervals used to characterize the frequency distribution of the attribute values for a column set.

All histograms described in this manual are frequency histograms. Each bucket in a frequency histogram contains some number of tokens representing the number of rows that have the attribute values belonging to its range.

See “Interval Histograms” on page 161 for details.

**HUT ACCESS Lock**  See ACCESS Lock and HUT Lock.

**HUT EXCLUSIVE Lock**  See EXCLUSIVE Lock and HUT Lock.

**HUT GROUP READ Lock**  See GROUP READ Lock, ROW RANGE Lock, and HUT Lock.

**HUT Lock**  Host UTility (HUT) locks are the class of locks placed on Teradata Database resources by the Archive/Recovery utility and related utility commands.

HUT ACCESS, HUT READ, and HUT GROUP READ locks are the only severities of HUT locks available for an Archive operation, while HUT EXCLUSIVE and HUT WRITE locks are the only severities available for a Recovery, Build, Copy, Rollback, or Rollforward operation (see Teradata Archive/Recovery Utility Reference for details).

**HUT READ Lock**  See READ Lock, ROW RANGE Lock, and HUT Lock.

**HUT WRITE Lock**  See WRITE Lock and HUT Lock.

**I2E**  Internal-to-External

**IEEE**  Institute of Electrical and Electronics Engineers (http://www.ieee.org)

The leading US-based professional society for electrical and electronics engineers.

The largest of its member societies is the IEEE Computer Society (http://www.computer.org), whose mission is similar to that of the ACM.

**Interval**  With respect to an interval histogram, an interval is a bounded, nonoverlapping set of attribute value counts.

**IMS**  Information Management System. The hierarchical database management product developed by IBM Corporation in the late 1960s.

**Interval Histogram**  See Histogram. Also see “Interval Histograms” on page 161.

**ISO**  International Organization for Standardization http://www.iso.org

An international umbrella standards organization based in Geneva, Switzerland, that also certifies the ANSI SQL standard.
The following passage from the ISO web site explains why the name of the organization does not match its (apparent) initialism: “Because “International Organization for Standardization” would have different abbreviations in different languages (“IOS” in English, “OIN” in French for Organisation internationale de normalisation), it was decided at the outset to use a word derived from the Greek isos, meaning "equal". Therefore, whatever the country, whatever the language, the short form of the organization’s name is always ISO.”

The ISO packaging of the SQL standard can be obtained from the following web site: http://www.iso.org/iso/en/

StandardsQueryFormHandler.StandardsQueryFormHandler?scope=CATALOGUE&sortOrder=ISO&committee=ALL&isoDocType=ALL&title=true&keyword=sql

Join Index  A vertical partition of a base table that can, depending on how it is defined, create various types of prejoins of tables, including sparse and aggregate forms. Join indexes cannot be queried directly by an SQL request; instead, they are used by the Optimizer to enhance the performance of any queries they Cover.

A join index that only vertically partitions a base table is referred to as a single-table join index.

A join index that prejoins two or more base tables is referred to as a multitable join index.

Both types of join index can be created in sparse or aggregate forms and can have a subset of their columns compressed (see Compression).

Unlike the primary index, which is stored in-line with the row it indexes, join indexes are stored in separate subtables that must be maintained by the system. Join index subtables also consume disk space, so you should monitor your queries periodically using EXPLAIN modifiers to determine whether the Optimizer is using any of the join indexes you designed for them. If not, you should either drop those indexes or rewrite your queries in such a way that the Optimizer does use them.

LOB  Large Object

Any data object that is larger than the maximum row size for Teradata Database. There are two types of LOB: BLOB and CLOB.

Lock  A mechanism for regulating the accessibility of a system resource. In the context of Teradata Database, a lock is a mechanism for regulating access to a database resource.

Teradata Database locks have two orthogonal components:

• Level
• Severity

See “Database Locks, Two-Phase Locking, and Serializability” on page 706 for details.
**Locking Level**  The level of a lock refers to the database object on which it enforces some level of severity. Note that the system sets and enforces HUT locks only at the table and database levels.

The range of levels for the locks available for use in Teradata Database are as follows:

<table>
<thead>
<tr>
<th>Most Encompassing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
</tr>
<tr>
<td>Table</td>
</tr>
<tr>
<td>Row hash</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Least Encompassing</th>
</tr>
</thead>
</table>

See “Teradata Database Locking Levels and Severities” on page 723 for details.

**Locking Severity**  The severity of a lock refers to how exclusive it is in accepting or rejecting other locking severities. Note that the system sets and enforces HUT locks only at the EXCLUSIVE, READ, and GROUP READ severities.

The range of severities for the locks available for use in Teradata Database is as follows:

<table>
<thead>
<tr>
<th>Most Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCLUSIVE</td>
</tr>
<tr>
<td>WRITE</td>
</tr>
<tr>
<td>READ</td>
</tr>
<tr>
<td>CHECKSUM</td>
</tr>
<tr>
<td>ACCESS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Least Exclusive</th>
</tr>
</thead>
</table>

See “Teradata Database Locking Levels and Severities” on page 723 for details.

Also see EXCLUSIVE Lock, GROUP READ Lock, READ Lock, and ROW RANGE Lock.

**Loner**  A loner is an attribute value in an interval histogram 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.

By this definition, the maximum number of loners in a histogram is 400.

In practice, a column might produce more than 400 loners, so the implemented upper bound on the number of loners per interval histogram is 398 (or 199 intervals).
This leaves at least one equal-height interval to account for the remaining values in the distribution.

See “Interval Histograms” on page 161 for details.

**LT/ST** Large Table/Small Table (join)

An optimized join type used to join fact (large) tables with their satellite (small) dimension tables.

See “Star Join Optimization” on page 444 for details.

**MLPPI** Multilevel Partitioned Primary Index

A Partitioned Primary Index defined with multiple partitioning expressions and an equal number of subpartitioning levels, hence multiple levels.

Compare with SLPPI.

See Database Design for details.

**NoPI Table** A table that has no primary index. All access to NoPI table rows is by means of a full-table scan unless you define Secondary Indexes on the table that are used in conditions for queries against it or Join Indexes that cover queries made against it.

See Database Design for details.

**NPPI** Nonpartitioned Primary Index

A Primary Index that is not partitioned into range buckets by means of a partitioning expression.

See Database Design for details.

**Null Filtering Condition** Any predicate that evaluates to FALSE for nulls is referred to as a null filtering condition, or NFC.

**NUPI** Non-Unique Primary Index

A Primary Index that is not uniquely constrained. NUSIs are often used to position rows from multiple tables on the same AMP to facilitate join operations on those tables.

See Database Design for details.

**NUSI** Non-Unique Secondary Index

An AMP-local Secondary Index designed to be used for set (multirow) selection rather than single row selection.

See Database Design for details.

**OCC** Optimistic Concurrency Control. An experimental method of transaction concurrency control that does not use locking. Commercial RDBMSs do not use OCC.

OCES  Optimizer Cost Estimation Subsystem.
See “Cost Optimization” on page 267 for details.

ODBC  Open DataBase Connectivity.
A de facto standard API for communicating between client applications and relational databases using SQL.
The ANSI SQL standard API, referred to as CLI (sometimes as SQL/CLI), or Call-Level Interface, is based on the ODBC specification. Beginning with ODBC version 3.0, the ODBC standard and the ANSI SQL Call Level Interface standards became identical.
See http://www.sqlsummit.com/ODBCPORT.HTM.

OLTP  OnLine Transaction Processing

Operation Tree  The parse tree created by the Optimizer by transforming the ResTree for an SQL request. A synonym for White Tree.

OptApply  The PE query processing component that takes the plastic steps produced by the Generator and produces Concrete Steps after applying any data parcels carried by the request from a USING request modifier (see SQL Data Manipulation Language).
Steps packaging passes the concrete steps to the Dispatcher for assignment to the AMPs.
See “OptApply” on page 68 for details.

Optimizer  The PE query processing component that takes the ResTree produced by Query Rewrite and creates optimized access and, when necessary, join plan for the query.
The Parse Tree output of the Optimizer is known as the White Tree, which then becomes the input to the Generator or, if the optimized request is prefaced by the EXPLAIN request modifier (see “Chapter 5 Interpreting the Output of the EXPLAIN Request Modifier” and “EXPLAIN Request Modifier” in SQL Data Manipulation Language), is converted to a textual translation of the plan.
In this case, the system adds additional costing information about various operations the Optimizer does not cost in order to both make the EXPLAIN text output more comprehensive and to provide information for the benefit of Teradata Workload Manager, which can process that data with its Query Estimator tool to estimate the resources required to process a given query or query workload.
See Chapter 2: “Query Rewrite and Optimization” for details.

Parameterized PK Request  A request that contains only the following:
- Parameterized PK statements
- Null statements
  See SQL Data Manipulation Language for the definition of null statement.
See “Peeking at the Request Cache” on page 36 for details.
Parameterized PK Statement  A single SQL statement with a set of USING variables that are used in equality predicate conditions on an NPPI, Partitioned Primary Index, or USI of a single table with no ORed conditions.

See “Peeking at the Request Cache” on page 36 for details.

Parser  The PE-based software system responsible for parsing, optimizing, and dispatching SQL requests.

See Dispatcher, Generator, OptApply, Optimizer, Query Rewrite, Resolver, and Syntaxer for details.

Also see “The SQL Parser” on page 18.

Parse Tree  A tree data structure that maps 1:1 with an SQL request submitted to the Parser. Also known as a SynTree. The initial parse tree for a request is further transformed in stages to a ResTree and then to an Operation Tree before being further transformed into plastic steps.

See Chapter 1: “Request Parsing” and “Parse Tree Representations of an SQL Request” on page 134 for details.

Partial Cover  A Hash Index, Join Index, or nonunique secondary index (NUSI) that covers some, but not all, of the columns referenced in a query. In this case, the system must join to the relevant base table to pick up the remaining columns required to complete the Cover.

See Database Design for details.

Partition  An AMP-based cluster of PPI table rows that share the same value, or partition number, for their evaluation by the partitioning expression for that table.

Each row is inserted into a partition based on the evaluation of the user-defined partitioning expression for the table. Rows are first hashed to an AMP and then stored within each partition on that AMP in hash value/uniqueness order.

See Database Design for details.

PARTITION  A system-derived column in the row header for each row in a PPI table. A PARTITION column contains the partition number for the row it represents.

See Database Design for details.

Partition Elimination  An automatic optimization in which the Optimizer determines, based on query conditions and a partitioning expression, that some partitions for that partitioning expression cannot contain qualifying rows, and causes those partitions to be skipped. Partitions that are skipped for a particular query are called eliminated partitions (see Eliminated Partition). When multiple partitioning expressions are defined on a table or join index, the system can combine partition elimination at each of the levels to further reduce the subsets of data that need to be scanned. Generally, the greatest benefit of a PPI is obtained from partition elimination.

Also see the following glossary definitions: Delayed Partition Elimination, Dynamic Partition Elimination, and Static Partition Elimination.

See “Partition Elimination” on page 298 for details.
**Partitioned Primary Index**  A **Primary Index** that first distributes rows to the AMPs as they would be by an **NPPI**, then distributes them to a set of range partitions that are determined by the DBA and specified using a `PARTITION BY` clause in a `CREATE TABLE` or `ALTER TABLE` request (see “CREATE TABLE” and “ALTER TABLE” in SQL Data Definition Language for details).

Partitioned primary indexes can be partitioned at a single level (see **SLPPI**) or at multiple levels (see **MLPPI**).

PPIs are very useful for various types of range queries.

See Database Design for details.

**PDE**  Parallel Database Extensions

A virtual machine layer between the Teradata Database software and the Teradata file system and the underlying operating system.

The PDE presents a common interface to the Teradata Database software that permits the RDBMS and file system to be more easily ported to different operating systems.

**PE**  Parsing Engine vproc

The set of software services that controls the query processing and session management components of a Teradata Database.

**PI**  See Primary Index.

**Pink Tree**  A partially transformed Red Tree that has not become a completed White Tree.

See “Query Rewrite” on page 54 and “Query Rewrite” on page 74 for details.

**PK**  See Primary Key.

**Plastic Steps**  Plastic steps are, except for statement literals, a data-free skeletal tree of compiled machine language AMP directives derived from the White Tree. The completed plastic steps tree is then passed to the Request Cache and to OptApply for further processing into Concrete Steps.

See “Definition: Plastic Steps” on page 67 for details.

**PPI**  See Partitioned Primary Index.

**Predicate**  A truth-valued function that describes either a property of objects or a relationship among objects as represented by a set of variables. In relational database management systems, predicates are also referred to as *conditions*, *search conditions*, or *terms*.

The purpose of predicates in SQL queries is to enhance selectivity, narrowing the results set for a query to only those rows that are relevant to the desired outcome.

In the SQL language, predicates are the specifications made in the **WHERE** clause of any query (including those specified with **AND** or **OR** operators), the **ON** clause of a join or **MERGE** request specification, the **HAVING** clause of an aggregation, or the **QUALIFY** clause of an ordered analytic function.
Internally, query optimizers often add predicates to queries in order to create optimization opportunities that would otherwise not be available. This application is derived from work done in the fields of logic and constraint databases.

**Predicate Redundancy**  A redundant *Predicate* is one that does not contribute additional selectivity to a query because other predicates with equivalent results are specified.

See “Join Predicate Redundancy Handling” on page 242 for details.

**Primary Index**  Primary Index

A set of columns in a table whose values are hashed to create a code used to distribute its rows to, and retrieve them from, the AMPs.

Except for NoPI Tables, each table in a Teradata database must have one, and only one, primary index, which might or might not be unique.

Compare with Primary Key.

See Database Design for details.

**Primary Key**  A set of columns in a table whose values make each row in that table unique.

Primary keys are a logical, not physical, concept that are often, but not necessarily, used as the primary index for a table when it is physically implemented.

A table can have multiple candidate keys, but only one primary key. Those candidate keys that are not defined as the primary key for a table are referred to as alternate keys.

Relationships between primary and foreign keys are often used to establish referential integrity between tables. These relationships are also frequently exploited by the Optimizer to enhance query performance.

See Database Design for details.

**QCD**  Query Capture Database. A system-installed database of tables into which various request information can be captured.

See Chapter 6: “Query Capture Facility” for details.

**QCF**  Query Capture Facility. A set of tools and tables that can be used to capture various information about SQL requests, which can then be subjected to further analysis.

See Chapter 6: “Query Capture Facility” for details.

**QITS**  Queue Insertion TimeStamp

A required, user-defined column that must be defined for all queue tables.

**QSN**  Queue Sequence Number

A useful, but not required, column that can be defined for queue tables.

**Query Block**  In the context of Query Rewrite, any view, a SELECT request or subquery, or an INSERT, DELETE, UPDATE, ABORT, ROLLBACK, CREATE TABLE AS, or MERGE request is considered to constitute a query block.
**Query Rewrite**  The PE query processing component that takes the ResTree produced by the Resolver and rewrites it in such a way that it is both semantically equivalent and runs faster. Query Rewrite then passes the revised ResTree, called ResTree’, to the Optimizer.

See “Query Rewrite” on page 54 and “Query Rewrite” on page 74 for details.

**RDBMS**  Relational Database Management System

A database management system based on relational set theory and the theorems, axioms, and operators provided by first order predicate logic. The set theoretic and logical foundation for an RDBMS provide a mathematically sound, predictable set of tools for managing data.

**READ Lock**  Placed in response to a SELECT request.

Archive requests can also place a HUT READ Lock or a HUT GROUP READ Lock on database resources.

A READ lock restricts access by users who require EXCLUSIVE or WRITE locks.

See “Teradata Database Locking Levels and Severities” on page 723 for details.

Also see ROW RANGE Lock.

**Red Tree**  The version of the parse tree produced by the Resolver. Synonym for ResTree.

The version of the red tree produced by the Resolver is further processed by Query Rewrite, which produces a rewritten, but semantically identical, red tree that it passes to the Optimizer.

See “Query Rewrite” on page 54 and “Query Rewrite” on page 74 for details.

**Releasing a Join**  Conversion from INNER JOIN syntax to comma syntax is referred to as releasing a join.

**Request**  One or more SQL statements submitted to Teradata Database as a single unit of work.

**Resolver**  The PE query processing component that takes the SynTree produced by the Syntaxer 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 the ResTree, to Query Rewrite.

See “Resolver” on page 33 for details.

**ResTree**  The version of the parse tree produced by the Resolver. Synonym for Red Tree.

See “Resolver” on page 33 for details.

**ResTree’**  The transformed version of the Red Tree produced by Query Rewrite.

ResTree’ is semantically identical to ResTree, but runs faster because it has been rewritten to eliminate unnecessary operations and to convert other operations to simpler or more explicit forms.

See “Query Rewrite” on page 54 and “Query Rewrite” on page 74 for details.
RI  Referential Integrity

A method of ensuring that no parent-child relationship data is ever orphaned in a relational database. Referential integrity uses the parent-child relationships between a Primary Key and a Foreign Key to prevent child table rows from ever being orphaned from deleted parent table rows.

The Teradata relational database management software supports three different kinds of relational integrity constraints:

- Referential Integrity constraint
  This is the standard RI constraint defined by the ANSI SQL standard.

- Batch Referential Integrity constraint
  This is a special Teradata Database form of RI that is less expensive to enforce in terms of system resources than standard referential integrity because it is enforced as an all-or-nothing operation (the entire transaction must complete successfully) rather than on a row-by-row basis, as standard referential integrity is checked.

- Referential Constraint
  This is a special Teradata Database form of RI, sometimes informally referred to as soft RI, that specifies constraints the Optimizer can use to optimize queries, but which are not enforced by the system.

The Optimizer often uses referential integrity constraints to enhance query performance.

See Database Design for details.

Rolling Column  A column characterized by having a constant number of rows per value and a varying number of unique values. Examples of rolling columns are those having a DATE or TIMESTAMP data type. For these columns, the demographics of existing data never changes, and only new data can add new distinct values to the column. Compare with Static Column.

ROW RANGE Lock  This is the internal implementation of the HUT GROUP READ Lock. The severity of a ROW RANGE lock is the same as that of a READ lock.

Search Condition  See Predicate.

Secondary Index  A vertically partitioned subset of base table columns used to facilitate data manipulation operations.

Unlike the Primary Index, which is stored in-line with the row it indexes, secondary indexes are stored in separate subtables that must be maintained by the system. Secondary index subtables consume disk space, so you should monitor your queries periodically using EXPLAIN modifiers to determine whether the Optimizer is using any of the secondary indexes you designed for them. If not, you should either drop those indexes or rewrite your queries in such a way that the Optimizer does use them.

There are two types of secondary index: USI and NUSI.

See Database Design for details.
Selectivity  A measure of the ability of a predicate or index to return a highly discriminating subset of rows from a table. The higher the selectivity, the fewer rows retrieved.

A predicate or index that retrieves many rows is said to have low selectivity. By definition, a predicate or index with low selectivity typically accesses more than one row per data block from the table on which it is defined. Older definitions of low selectivity stated that a predicate or index with low selectivity accesses more than ten percent of the rows in the table on which it is defined, but the greatly increased size of newer data block configurations makes this definition obsolete, because access rates of 1.0 or even 0.1 percent frequently occur for predicates and indexes with low selectivity.

A predicate or index that retrieves few rows is said to be highly selective. A highly selective predicate or index is one that does not access all of the data blocks for the table on which it is defined. Older definitions stated that a highly selective predicate or index typically accesses fewer than ten percent of the rows in the table on which it is specified or defined, but this definition is no longer accurate for the same reasons that the old definition of low selectivity is obsolete.

The more selective a predicate or index is, the more useful it is for enhancing performance.

Semijoin  A join operation in which all data that is unnecessary for answering a query is removed by the relational operations Select and Project before the join is considered. Teradata Database refers to semijoins more specifically as either Exclusion Joins or Inclusion Joins, and supports them for both Product and Merge join types.

SHARE Lock  SHARE lock is a deprecated synonym for READ Lock.

SI  See Secondary Index.

The leading US-based professional society for applied mathematics professionals.

SIGACT  ACM Special Interest Group for Algorithms and Computation Theory, the ACM SIG devoted to theoretical computer science (including theoretical issues of importance to database management, complexity theory, and formal logic).

SIGART  ACM Special Interest Group for ARTificial Intelligence

SIGFIDET  ACM Special Interest Group for FIle DEscription and Translation, the predecessor organization to ACM SIGMOD.

SIGKDD  ACM Special Interest Group for Knowledge Discovery and Data Mining

SIGMOD  ACM Special Interest Group for MODification of Data, the ACM SIG devoted to database management issues.

Skew  A measure of the asymmetry of the distribution of a set of attribute values or their cardinalities.

Formally, skewness is the third moment of the probability density function for a population of attribute value counts. The first two moments are the mean and the standard deviation, respectively, while the fourth moment is kurtosis.
With respect to skew in parallel databases, there are several possible types of skew:

- **Attribute value skew** refers to skew that is inherent in the data. An example might be a column that can have only two values such as TRUE or FALSE.
- **Partition skew** refers to skew that results from an uneven distribution of data across the AMPs.

The difference in usage 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.

See “Interval Histograms” on page 161 for further information.

**SLPPI**  
Single-Level Partitioned Primary Index  
A **Partitioned Primary Index** that has only one partitioning expression, hence a single level.  
Compare with **MLPPI**.  
See [*Database Design*](#) for details.

**Specific Plan**  
An optimizer plan that is based on having peeked at parameterized USING values during query parsing.  
Compare with **Generic Plan**.  
See “Peeking at the Request Cache” on page 36 for further information.

**SQL**  
The programming language used to create relational database objects (Data Definition Language, or DDL), to manipulate their contents (Data Manipulation Language, or DML), and to define their security attributes (Data Control Language, or DCL).

Now preferably pronounced ess-kew-ell, the language was originally named SEQUEL (Structured English QUEry Language) and was pronounced the same way as it was spelled. According to the ANSI SQL standard, SQL does not stand for Structured Query Language, as is commonly thought. The name is not an abstraction and does not represent anything other than the characters S, Q, and L.

**SQL Assistant**  
A Teradata request processing facility based on the ODBC API.  
See [*Teradata SQL Assistant for Microsoft Windows User Guide*](#) for details.

**Stale Statistics**  
Interval histogram statistics that no longer represent an accurate description of the column sets on which they were originally collected.

See “When Should Statistics Be Collected Or Recollected?” on page 190 and “Stale Statistics” on page 251 for further information.

**Static Column**  
A column characterized by having a varying number of rows per value and a constant number of unique values. Compare with **Rolling Column**.

**Static Partition Elimination**  
Partition elimination undertaken by the Optimizer during the primary optimization process, as opposed to being delayed until the time the finalized plan is built from a cached plan using built-in function values or USING request modifier variable values for the specific execution of the plan as is done in **Delayed Partition Elimination**.
See “Static Partition Elimination” on page 300 for details.
Also see Dynamic Partition Elimination and Partition Elimination.

**Synoptic Data Structure**  A data structure that contains summary, or synopsis, metadata.
In the case of an interval histogram, the object is actually a file structure rather than a data structure, meaning that it is stored on disk, not just manipulated in memory.
See “Interval Histograms” on page 161 for details.

**Syntaxer**  The PE query processing component that checks the validity of the syntax for the SQL text in a request parcel and then, if the syntax is valid, converts the text to a parse tree.
See “Syntaxer” on page 30 for details.

**SynTree**  The skeletal parse tree for an SQL request created by the “Syntaxer.”
See Chapter 2: “Query Rewrite and Optimization” for details.
Also see Black Tree.

**Term**  See Predicate

**TLE**  Target Level Emulation. TLE is a component of Teradata System Emulation Tool that is used to specify RAS, cost parameters, and cost profiles at various levels for a connected test system. The tool permits you to create multiple emulation scenarios covering a wide range of testing possibilities on a single test system.

**TPA**  Trusted Parallel Application
A TPA is an application that Teradata has certified to run Teradata Database safely. The Teradata Database software itself is a TPA.

**TSET**  Teradata System Emulation Tool. A client tool used to copy demographic information such as environmental cost factors and statistics from a production system to a test system in order to facilitate testing by emulating the production system environment.

**UDT**  User-Defined Type
A data type defined by someone other than Teradata. UDTs come in two variations: Distinct and Structured.
See SQL Data Definition Language and SQL External Routine Programming for further information about UDTs.

**UCS-2**  Universal Coded Character Set containing 2 bytes
UPI  Unique Primary Index

A Primary Index that is uniquely constrained. The rows from a table defined with a UPI tend to be distributed more evenly across the AMPs than rows from a table defined with a NUPI.

See Database Design for details.

USI  Unique Secondary Index

A Secondary Index designed to facilitate single-row access.

Primary keys and UNIQUE constraints are implemented internally as USIs.

See Database Design for details.

vproc  Virtual PROCess

A software emulation of a hardware Parsing Engine or Access Module Processor.

The Version 1 Teradata architecture used several different specialized node types to process data, including the following:

- IFP (InterFace Processor)
- COP (Communications Processor)
- APP (Application Processor)
- AMP

The Version 2 Teradata architecture is based on a common node configuration. Each TPA node can run one or more PE and AMP vprocs that emulate the functions of the Version 1 hardware nodes. The functions of the Version 1 IFP and COP nodes are consolidated in the PE vproc, while the analogous functionality of an APP node is running Teradata Tools and Utilities software on a non-TPA node in a Teradata system.


W3C is the principal standards organization for the World Wide Web.

WAL  Write Ahead Log or Write Ahead Logging.

A transaction logging scheme maintained by the File System in which a write cache for disk writes of permanent data is maintained using log records instead of writing the actual data blocks at the time a transaction is processed. Multiple log records representing transaction updates can then be batched together and written to disk with a single I/O, thus achieving a large savings in I/O operations and enhancing system performance as a result.

White Tree  The optimized parse tree for an SQL request. Synonym for Operation Tree. A textual version of the white tree is returned to a requestor who submits an EXPLAIN modifier for a request.

See Chapter 2: “Query Rewrite and Optimization” for details.
**WRITE Lock**   Placed in response to an INSERT, UPDATE, or DELETE request.

A WRITE lock restricts access by other users (except for applications that are not concerned with data consistency and choose to override the automatically applied WRITE lock by specifying a less restrictive ACCESS lock). See “Teradata Database Locking Levels and Severities” on page 723 for details.

Various Archive utility commands can also place a HUT WRITE Lock on database resources. See Teradata Archive/Recovery Utility Reference for details.

**Xerces**   A software library used to parse and manipulate XML.

Xerces implements a number of standard APIs for XML parsing, including DOM.
Numerics

2PC protocol 776, 778, 784, 803
2PC session mode 776, 778, 784, 803
2PC session protocol 776

A

ACCESS locks
compatibility with aggregate processing 798
described 725
enforced in a view 799
row hash-level vs. table-level for tactical queries 794
tactical queries 766, 794
tactical queries with group AMP steps 797
tactical queries with join indexes 796
tactical queries with joins 798
AccessLockForUncomRead flag 712
ACID properties of transactions 698
Activity Transaction Modeling process 645
AddWorkload macro 637
AggrEstCapPct cost profile flag 278
All AMPs statistical sampling 188
ALTER TABLE
locks placed 744
AnalysisLog QCD table 560
AnalysisStmts QCD table 562
ANSI join syntax
converting to comma join syntax 91
ANSI session mode 774, 777
rules 777
transaction processing case studies 779
Apply
component processes of 69
described 68
ApplySpoolLimit cost profile flag 282
AssumedPIUniquePct cost profile flag 278
ATM process 645
Attribute value skew 165

B

BestVals statistic 247, 248, 249
BIGINT data type
possible effects of large values on usefulness of statistics
collected on them 174
Bit mapping
EXPLAIN request modifier 484

Black tree 30
Block optimization
partial GROUP BY 353
Blocked locking requests 746
and deadlock 747
differences from deadlock 747
multistatement transactions 748
Query Session utility 746
single-statement transactions 747
transaction throughput 746
Bucket
definition for interval histograms 162
Bushy search tree for join order optimization 327

C

CacheableBlocks cost profile flag 283
Caching
non-immediate 24
Cardinality estimation
comparative accuracy of sampled statistics and random
AMP sampled statistics 176
comparison of relative accuracies of various methods of
collecting statistics 188
dealing with stale statistics for derived statistics 251
derived statistics 206
derived statistics extrapolation case 1 261
deriving column demographics from derived statistics 217
deriving column statistics from join predicate redundancies for derived statistics 228
deriving column statistics from join predicates for derived
statistics 227
extrapolated boundaries for extrapolated statistics in
derived statistics 257
extrapolating a nonrolling column case 7 265
extrapolating an open range date predicate case 3.1 263
extrapolating an open range date predicate case 4.2 264
extrapolating an open range predicate case 5.3 264
extrapolating an open range predicate with a wide final
interval case 6 265
extrapolating derived statistics 255
extrapolation case 2 262
handling multiple sources of information for derived
statistics 228
join plan with star join optimization for LT subquery join
465
optimized join cardinality estimate with single-row access
Index

via a unique index 253
propagating derived statistics across query blocks 230
propagating derived statistics for aggregate and OLAP
    function estimates across query blocks 230
propagating derived statistics to materialized instances
    within a query block 229
propagating derived statistics to temporary and committed
    joins 229
random AMP sampling 178
reasonable indexed join plan with star join optimization
    460
reasonable indexed join plan with star join optimization
    with LT NUSI 464
reasonable indexed join plan with star join optimization
    with LT USI 462
reasonable indexed join plan without star join
    optimization 458
reasonable indexed joins 457
reasonable unindexed join plan with optimization 469
reasonable unindexed join plan without join optimization
    468
relative accuracy of residual statistics versus random AMP
    samples for static columns in derived statistics 254
rolling columns 258
static columns 258
terminology for extrapolating statistics for derived
    statistics 257
using aggregate join indexes for derived statistics 218
using both single-column and multicolumn statistics for
    derived statistics 225
using CHECK constraints for derived statistics 219
using extrapolation to replace stale statistics for derived
    statistics 254
using join index discovery for derived statistics 231
using sampled statistics 175
using single-table predicates and multicolumn
    PARTITION statistics for join cardinality estimates
    from derived statistics 224
using single-table predicates for derived statistics 222
using single-table predicates for derived statistics when no
    statistics have been collected 224
using single-table sparse join indexes for derived statistics
    217
using subset and superset statistics for derived statistics 226
CCFIndependentValueRatio cost profile flag 278
CCFMult2SnglValueRatio cost profile flag 278
CHECKSUM lock
described 725
Classic hash join 398
CNF, see conjunctive normal form
COLLECT DEMOGRAPHICS 551, 552
COLLECT STATISTICS (Optimizer Form) 649
COLLECT STATISTICS (QCD form) 551, 552, 576, 645, 654
Collecting statistics 190
Columns
    rolling 258
    static 258
Common steps 64
definition 64
Complex updates 800
description 167
Concrete steps 21, 69, 70
definition 69
Concurrency control
    lock manager and 719
    locking 744
ConditionCosting cost profile flag 283
Confidence levels for cardinality estimates 488
Confidence levels for join cardinality estimates 490
Confidence levels for optimizer cardinality estimates
    EXPLAIN request modifier 487
Conjunctive normal form 268
Constant movearound 113
Constant predicate evaluation 109
ConstrScanAxPath cost profile flag 284
Correlated exclusion fast path merge join 437
Correlated exclusion slow path merge join 437
Correlated inclusion fast path merge join 437
Correlated inclusion slow path merge join 437
Correlated join 436
correlated exclusion merge join 437
correlated exclusion product join 437
correlated inclusion merge join 437
correlated inclusion product join 437
exclusion fast path merge 437
exclusion slow path merge 437
EXISTS join 437
inclusion fast path merge 437
inclusion slow path merge 437
NOT EXISTS join 437
types 437
Cost estimation
effect of stale statistics on 293
Cost functions
    optimizer 271
Cost optimization
definition 267
environmental cost factors 296
external cost parameters 297
path selection 268
types of environmental cost factors 296
Cost prediction 269
Cost profile flags
    AggrEstCapPct 278
    ApplySpoolLimit 282
    AssumedPIUniquePct 278
CacheableBlocks 283
cardinality estimation 277, 278
CCFIndependentValueRatio 278
CCFMulti2SnglValueRatio 278
ConditionCosting 283
ConstrScanAxPath 284
EnableSessLevelDrvStats 278
EstWithSingleRowOpt 279
ExtrapolateStatistics 279
flags not used for cardinality estimation or join costing 284
GroupAMPPISampling 279
IndexFillFactor 284
join costing 277, 282
JoinCardSkewAdj 280
LegacyJoinCardDefault 280
LIKEEstPartialIntRatio 280
MaxJoinTables 284, 322
MaxStatsInterval 285
ocsDebug 285
PreJCost 283
RASHistAbsDeviation 281
RASHistDeviationPct 281
RevertEstimationLogic 282
SoftUniquePct 282
SpoolFillFactor 285
SpoolLimitPercent 285
TableFillFactor 285
UseActualBlockSize 286
Cost profile types
type 1 276
type 2 276
Cost profiles
activating 289
administering 286
components 273
DBC.CopyCostProfile 288
DBC.CreateNewCostProfile 288
DBC.DeleteCostProfile 288
DBC.InsertConstantValue 288
DBSCostProfile 290
dictionary tables 276
instance specifications 277
modifying 287
SQL DIAGNOSTIC statements 289
terminology 272
type specifications 273
Cost-based query optimization 130
Costing a product join with dynamic partition elimination 369
CREATE INDEX 576, 649
locks placed 744
Creating QCD tables
procedure using BTEQ 552
VEComp procedure 551
Cursors
positioned cursors and their locking modes 772
updatable cursors and their locking modes 772

D
Data Dictionary
cache 22
how used by the Parser 22
Database lock
described 719
Database Query Analysis
Index analysis 646
index application 655
workload definition 643
Database Query Log 641, 642, 645
DataDemographics QCD table 564
DateSpecific generic plan 37
DateSpecific specific plan 37
DBC.ConstantDefs 276
DBC.ConstantValues 276
DBC.CostProfiles 276
DBC.CostProfileTypes 276
DBC.TableSizeVX system view 565
DBC.TVFields, FieldStatistics column 168
DBQL 645
DBQLStepTbl table 270
DBS Control flags
AccessLockForUncomRead flag 712
DBSControl flags
DisablePeekUsing flag 40
DCL statements
locks 766
DDL statements
locks 766
Deadlock and blocked locking requests 747
Deadlocks 753
access mode 755
detecting for HUT locks 753
detection 755
detected and resolution 755
differences from blocked requests 747
difference of prevention 745
local 754
LOCKING FOR ACCESS and deadlock prevention 759
LOCKING modifier in view definitions and deadlock prevention 759
LOCKING modifier NOWAIT option and deadlock prevention 758
prevention 756
prevention with LOCKING modifier 756
resolution 755
DECIMAL data type
possible effects of large DECIMAL values on usefulness of
statistics collected on them 174
Delayed partition elimination 298, 299, 307
Derived statistics 206
aggregate join index discovery for deriving column
demographics 234
and stale statistics 251
cost profile flag for enabling or disabling extrapolation of
statistics 256
criteria for derivation 207
definition 206
deriving column correlations from join index statistics 233
deriving column demographics 217
deriving column demographics from aggregate join
indexes 218
deriving column demographics from both single-column
and multicolumn statistics 225
deriving column demographics from CHECK constraints
219
deriving column demographics from join predicate
redundancies 228
deriving column demographics from join predicates 227
deriving column demographics from referential integrity
constraints 219
deriving column demographics from single-table
predicates 222
deriving column demographics from single-table
predicates and multicolumn PARTITION statistics
224
deriving column demographics from single-table
predicates when no statistics exist 224
deriving column demographics from single-table sparse
join indexes 217
deriving column demographics from statistics subsets and
supersets 226
deriving selectivities by combining them 238
deriving selectivities by combining them using value
mapping 240
deriving single-table cardinality estimates 235
deriving single-table cardinality estimates using
histograms 236
deriving single-table cardinality estimates using single-
table join indexes 235
determining the number of unique values 246
extrapolated boundaries 257
extrapolating a nonrolling column case 7 265
extrapolating an open range date predicate case 3.1 263
extrapolating an open range date predicate case 4.2 264
extrapolating an open range predicate case 5.3 264
extrapolating an open range predicate with a wide final
interval case 6 265
extrapolation 255
extrapolation case 1 261
extrapolation case 2 262
flow of 213
handling join predicate redundancy 242
handling multiple sources of information 228
how they work 207
inherited statistics 216
join index discovery 231
join index discovery for single-table cardinality estimates
231
optimized join cardinality estimates for single-row access
using a unique index 253
propagating across query blocks 230
propagating for aggregate and OLAP function estimates
230
propagating to materialized instances within a query block
229
propagating to temporary and committed joins 229
relative accuracy of residual statistics versus random AMP
samples 254
rolling columns 258
static columns 258
terminology for extrapolating statistics 257
using extrapolation to replace stale statistics 254
Direct merge join 381
DisablePeekUsing flag 40
Disjunctive normal form 268
Dispatcher 19, 21, 69, 70
congeete steps 70
deadlock management 70
described 70
execution control 70
queue table FIFO cache management 71
request abort management 70
response control 70
transaction abort handling 70
transaction management 70
DML statements
locks placed 767
DNF, see disjunctive normal form
DROP INDEX 576
DUMP EXPLAIN
effects of request cache peeking on 657
Duplicate predicate removal 113
Duplicate rows
MULTISET table kind option 778
SET table kind option 784
Dynamic AMP sampling 178
Dynamic hash join 404
Dynamic partition elimination 298, 299
E
EnableSessLevelDrvStats cost profile flag 278
Enabling TLE 684
Environmental cost factors 296
external cost parameters 296, 297
performance constants 296
   types 296
Equal-height interval 172
   definition 162
   fields 172
Equal-height interval histograms 166
   definition 163
EstWithSingleRowOpt cost profile flag 279
Exclusion join 418
   nullable columns 418
   types 418
Exclusion merge join 419
Exclusion product join 364, 422
Exclusion product join with dynamic partition elimination 432
EXCLUSIVE locks
   described 724
EXISTS correlated join 437
EXPLAIN report confidence levels
   join operations 490
   nonjoin operations 488
EXPLAIN request modifier 484
   hash joins 514
   indexed access 517
   join processing 512
   limited to processing SQL requests 485
   MERGE conditional steps 529
   parallel steps 520
   partitioned primary index 522
   report confidence levels 487
   report overview 484
   report terminology 493
   request cache peeking 485
   Teradata Visual Explain utility 486
   Teradata Workload Manager 485
   time estimates 484
   UPDATE (Upsert Form) 534
ExtrapolateStatistics cost profile flag 279
ExtrapolateStatistics flag 256
Extrapolating statistics 255
   case 1 261
   case 2 262
   extrapolating an open range date predicate case 3.1 263
Extrapolation
   cost profile flag for enabling or disabling extrapolation of statistics 256
   cost profile threshold flags to define deviation thresholds 260
   equation to determine the extrapolated boundary for date statistics 257
   extrapolated boundary for statistics 257
Index

high-biased interval histogram 164
how stored 168
HTMemAlloc 401, 402
HTMemAllocBase 401, 402
HUT ACCESS locks
  described 725
HUT GROUP READ locks
  described 724
HUT locks
  compatibilities with database locks 726, 728
  described 732, 733, 735
  GROUP READ lock 736
  interactions with database locks 735
  levels 733
  severities 733
HUT READ locks
  described 724
HUT WRITE locks
  described 724
Hybrid Hash Join 398

I
Inclusion join 424
  inclusion merge join 424
  inclusion product join 424
Inclusion merge join 424
Inclusion product join 364, 424
Inclusion product join with dynamic partition elimination
  429
Index analysis 646
  DBSControl flags for tuning workload cache limits 640
Index application 655
Index Wizard utility 574, 629, 630, 635, 642, 643, 646, 649,
  654, 656
  AnalysisLog table 560
  DataDemographics table 564
  how used 635
  IndexRecommendations QCD table 573
Index_Field QCD table 569
IndexColumns QCD table 570
Indexed access 294
  cost optimization 294
  EXPLAIN request modifier 517
Indexes
  ensuring use 294
  optimizer use for indexed access 294
IndexFillFactor cost profile flag 284
IndexMaintenance QCD table 571
IndexRecommendations QCD table 573
IndexTable QCD table 578
Inherited statistics
  bidirectional 216
  unidirectional 216

INITIATE INDEX ANALYSIS 560, 574, 576, 585, 629, 640,
  646, 647, 649, 650, 651, 652
INITIATE PARTITION ANALYSIS 639, 640, 646, 647, 650,
  651, 652, 664, 665
Inner product join 364
INSERT EXPLAIN 551, 552, 645
  effects of request cache peeking on 657
  WITH STATISTICS 643
Interval
  definition for interval histograms 164
Interval histograms
  as synopsis data structures 161, 165
  cardinality estimates 197
  compressed 161
  described 161
  equal-height 166
  high-biased interval 167
  storage 168
  terminology 162
  types 166
  types of intervals 172
  values collected and computed 193
Isolation level
  definition 710

J
Japanese character code notation, how to read 819
Join algorithms
  types 321
Join algorithms, see Join methods 361
Join costs
  evaluation 323
Join distribution strategies 333
Join geography 333
  effects of stale statistics on join planning 333
Join methods 361
  correlated join 436
  exclusion join 418
  exclusion merge join 419
  exclusion product join 422
  fast path local nested join 411
  hash join 397
  inclusion join 424
  inclusion merge join 424
  inclusion product join 424
  local nested join 407
  merge join 372
  minus all join 438
  nested join 405
  overview 361
  product join 363, 364
  remote nested join 412
  rowID join 434
slow path local nested join 408
  types 364
Join modes, see Join methods
Join order
  as a function of the number of tables to be joined 330
evaluation 344
  search tree 326
Join order optimization
  bushy search trees 327
left-deep search tree branches 330
right-deep search tree branches 330
Join order search tree types
  bushy 327
left-deep 326
Join planning 318
  bushy search trees 327
  classic hash join 398
components 318
controlling size of hash table 401
correlated exclusion fast path merge 437
correlated exclusion product join 437
correlated exclusion slow path merge 437
correlated inclusion fast path merge 437
correlated inclusion merge join 437
correlated inclusion product join 437
correlated inclusion slow path merge 437
correlated join 436
cost effectiveness of nested joins 406
costing hash joins 402
costing nested joins 406
direct merge join 381
dynamic hash join 404
evaluating join costs 323
evaluating join order 344
evaluating join orders 344
exclusion join 418
exclusion merge join 419
exclusion product join 422
EXISTS correlated join 437
fast path local nested join 411
five-join lookahead 352
hash join 397
hash join control variables 401
hash join terminology 397
inclusion join 424
inclusion merge join 424
inclusion product join 424
indexed star join 447
join geography 333
join index
  general DELETE maintenance example 476
  general UPDATE maintenance example 480
  INSERT maintenance example 478
  optimized DELETE maintenance examples 477
  optimized UPDATE maintenance example 482
join index DELETE, INSERT, and UPDATE examples 474
join index example for a multiway join 473
join index example for search on indexed rows 470
join index example for simple join query 470
join index example of aggregation on indexed rows 471
join index example of index with a secondary index 472
join index example of joining with a base table 471
join index example of resolving a single-table query 472
join index maintenance 474
join indexes 470
join order 330
join order search trees 326
join plan with star join optimization for LT subquery join 465
join processing methods 321
joining relations on the same domain 322
left-deep search trees 326
limit on number of joined tables 322
local nested join 405, 407
merge join 372
minus all join 438
miscellaneous star join optimization considerations 450
nested join 405
NOT EXISTS correlated join 437
one-join lookahead 349
optimizer 318
overview of join methods 361
parser memory requirements 323
partial GROUP BY block optimization 353
planning for n-way joins 325
process overview 344
product join 363
reasonable indexed join plan with star join optimization 460
reasonable indexed join plan with star join optimization with LT NUSI 464
reasonable indexed join plan with star join optimization with LT USI 462
reasonable indexed join plan without star join optimization 458
reasonable LT/ST indexed join 448
reasonable LT/ST join 448
reasonable LT/ST unindexed join 449
reasonable unindexed join plan with join optimization 469
reasonable unindexed join plan without join optimization 468
reasonable unindexed LT/ST join 449
remote nested join 405, 412
rowID join 434
rowkey-based merge join 383
selecting indexes for a star join 452
single window merge join 386
sliding window merge join 389

SQL Request and Transaction Processing 883
slow path local nested join 408
snowflake join optimization 442
star join examples 455
star join optimization 442, 444
table lookahead 349
types of correlated joins 437
types of star joins 447
unindexed LT/ST join 449
unindexed star join 447
unreasonable indexed LT/ST join 448
unreasonable unindexed join 449
unreasonable unindexed LT/ST join 449
use of secondary indexes 325

Join process
ing EXPLAIN request modifier 512
Join syntax
converting from ANSI form to comma form 91
JoinCardSkewAdj cost profile flag 280
JoinIndexColumns QCD table 582
Joins
 distribution strategies 333
   methods 361
   Optimizer and join plans 325
   row elimination 331
Joins. See individual join methods

L
Large Table/Small Table join. See LT/ST join
Left outer product join 364
Left-deep search trees
 branches for join order optimization 330
 for join order optimization 326
LegacyJoinCardDefault cost profile flag 280
Lexer 30
LIKEEstPartialIntRatio cost profile flag 280
Local nested join 407
Lock escalation
 automatic 795
Lock Manager
described 719
 locking levels 719
 releasing locks 698, 707, 721, 744, 777, 778, 784, 789, 791, 792
 LOCK ROW modifier 759
 LOCKING request modifier 756
Locks
 ACCESS 725
 ACCESS and group AMP steps 797
 ACCESS with join operations 798
 blocked requests 746
 changing default assignments with LOCKING request
 modifier 742
 CHECKSUM 725
 compatibility among modes 728
 cursor locking modes 772
 deadlock detection 755
 deadlocks 753
 default assignments for various SQL requests 739
 EXCLUSIVE 724
 explicit transaction 722
 group AMP considerations 793
 HUT 726, 728, 732, 733, 735
 HUT ACCESS LOCK 725
 HUT GROUP READ 724
 HUT READ 724
 HUT WRITE 724
 lock manager and 720
 maximizing concurrency 744
 modes 723
 object levels 719
 pseudo-table 750
 READ 724
 releasing 698, 707, 721, 744, 777, 778, 784, 789, 791, 792
 row-level and aggregation 798
 rules for upgrading 743
 set by client utilities 730, 732
 SHARE 724
 when released 744
 WRITE 724
Locks. See also Deadlocks
 Loner 164, 171, 173
 definition 165
 Lookahead
 five-join 352
 one-join 349
 LT/ST join
described 442
 examples 455
 indexed join
   reasonable 448
   unreasonable 448
 reasonable indexed join 448
 reasonable unindexed join 449
 unindexed join 449
 unreasonable indexed join 448
 unreasonable unindexed join 449

M
MaxJoinTables cost profile flag 284, 322
MaxStatsInterval cost profile flag 285
MaxVals statistic 247, 248, 249
MERGE
 EXPLAIN request modifier for conditional steps 529
 Merge join 372
 costing 374
 definition 372
direct 381
distribution strategies 376
fast path inner 375
generic strategy 373
rowkey-based 383
single window 386
sliding window 389
slow path inner 374
Minus all join 438
MinVals statistic 246, 247, 248, 249
MULTISET table kind option and duplicate rows 778

N

Nested join 405
cost effectiveness 406
costing 406
local 405
remote 412
remote nested join 405
Nested Join and NoPI tables 405
NoPI 364
NoPI tables 60, 333
assumptions for random AMP sampling and NoPI tables 178
column access and NoPI tables 294
indexed access 294
Nested Join and NoPI tables 405
random AMP sampling and NoPI tables 144
random AMP sampling for NoPI tables 144
Rowkey-Based Merge Join and NoPI tables 383
skew and NoPI tables 165
Sliding Window Merge Join and NoPI tables 393
Sliding Window Merge Join with dynamic partition elimination and NoPI tables 394
strategies for joining a NoPI table with a PPI table 362
NOT EXISTS correlated join 437
NOWAIT locking modifier option 730, 758
NULLEQ marking 113
n-way joins
planning 325

O

ocsDebug cost profile flag 285
Operation tree 21
OptApply 67, 68
block diagram 68
component processes 69
Optimizer 18, 20, 58, 59, 80, 126
block diagram 62
bushy search trees 327
cardinality estimation 197
component processes 63
component processes of 63
compressed interval histograms 167
conjunctive normal form 268
correlated join 436
cost estimation and statistics 293
cost functions 271
cost optimization 267
database object substitution 125
disjunctive normal form 268
environmental cost factors 296
equal-height interval histograms 166
evaluating join orders 344
exclusion join 418
exclusion product join 422
five-join lookahead planning 352
hash join 397
high-biased interval histograms 167
how it decides between histogram and random AMP sample statistics for NUSI subtables 147
inclusion join 424
inclusion merge join 424
inclusion product join 424
indexed access 294
interval histograms 161
join methods 361
join order as a function of the number of tables to be joined 330
join order search trees 326
join planning 318, 325
join plans 294
join processing methods 321
join table lookahead 349
left-deep search trees 326
merge join 372
minus all join 438
nested join 405
one-join lookahead planning 349
parallel steps and 64
parse tree representation of a query 134
path selection 268
process overview 132
processes 132
product join 364
redundant join elimination 118
remote nested join 412
statistical profiles 193
Teradata Index Wizard utility 59
translation to internal representation 134
view materialization 125
Optimizer confidence levels
effect of random AMP sampling 492
high confidence 487
index join confidence 487
join operations 490
low confidence 487
Index

no confidence 487
not related to Teradata Index Wizard confidence levels stored in Level attribute of StatsRec QCD table 487
single-table retrievals 488
Optimizer join plans 318
bushy search trees 327
classic hash join 398
components 318
controlling size of hash table 401
correlated exclusion fast path merge 437
correlated exclusion product join 437
correlated exclusion slow path merge 437
correlated inclusion fast path merge 437
correlated inclusion merge join 437
correlated inclusion product join 437
correlated inclusion slow path merge 437
correlated join 436
cost effectiveness of nested joins 406
costing hash joins 402
costing nested joins 406
direct merge join 381
evaluating join costs 323
evaluating join order 344
evaluating join orders 344
exclusion join 418
exclusion merge join 419
exclusion product join 422
EXISTS correlated join 437
fast path local nested join 411
five-join lookahead planning 352
general DELETE maintenance examples 476
hash join 397
hash join control variables 401
hash join terminology 397
inclusion join 424
inclusion merge join 424
inclusion product join 424
indexed star join 447
join geography 333
join index DELETE, INSERT, and UPDATE examples 474
join index example for a multiway join 473
join index example for search on indexed rows 470
join index example for simple join query 470
join index example of aggregation on indexed rows 471
join index example of index with a secondary index 472
join index example of joining with a base table 471
join index example of resolving a single-table query 472
join index general UPDATE maintenance example 480
join index INSERT maintenance example 478
join index maintenance 474
join index optimized DELETE maintenance examples 477
join index optimized UPDATE maintenance example 482
join indexes 470
join order 330
join order search trees 326
join plan with star join optimization for LT subquery join 465
joining relations on the same domain 322
left-deep search trees 326
limit on number of joined tables 322
local nested join 405, 407
merge join 372
minus all join 438
miscellaneous star join optimization considerations 450
nested join 405
NOT EXISTS correlated join 437
n-way joins 325
one-join lookahead planning 349
overview of join methods 361
parser memory requirements 323
partial GROUP BY block optimization 353
product join 363
reasonable indexed join plan with star join optimization 460
reasonable indexed join plan with star join optimization with LT NUSI 464
reasonable indexed join plan with star join optimization with LT USI 462
reasonable indexed join plan without star join optimization 458
reasonable LT/ST indexed join 448
reasonable LT/ST join 448
reasonable LT/ST unindexed join 449
reasonable unindexed join plan with join optimization 469
reasonable unindexed join plan without join optimization 468
reasonable unindexed LT/ST join 449
remote nested join 405, 412
rowID join 434
rowkey-based merge join 383
selecting indexes for a star join 452
single window merge join 386
sliding window merge join 389
slow path local nested join 408
snowflake join optimization 442
star join examples 455
star join optimization 442, 444
table lookahead 349
types of correlated joins 437
types of star joins 447
unindexed LT/ST join 449
unindexed star join 447
unreasonable indexed LT/ST join 448
unreasonable unindexed join 449
unreasonable unindexed LT/ST join 449
use of secondary indexes 325
Optimizer statistics
values collected and computed 193

P
Parallel steps 64
definition 64
EXPLAIN request modifier 520
Parameterized requests
USING-independent 38
Parameterized PK requests 37
Parameterized PK statements 37
Parameterized requests
USING-dependent 38
USING-independent 38
Parse tree representation of a query 134
Parser 18
column processes of 20
major components of 18
Parsing Engine
block diagram 19
column processes 20
Partial GROUP BY block optimization 353
Partition analysis
DBSCControl flags for tuning workload cache limits 640
Partition elimination 38, 298
delayed 298, 299, 307
dynamic 298, 299
dynamic partition elimination 366
static 298, 299, 300
Partition skew 165
PARTITION statistics
restriction on sampling 177
Partitioned primary index
EXPLAIN request modifier 522
PartitionRecommendations QCD table 585
Path selection 268
Piggybacked product join 364
Plastic step 67
Plastic steps 21, 133
definition 67
polation 256
PPI join methods
direct merge join 381
product join with dynamic partition elimination 366
rowkey-based merge join 383
single window merge join 386
sliding window merge join 389
strategies for joining a PPI table with a NoPI table 362
Predicate martialing 123
Predicate push down and pullup 115
Predicate QCD table 589
Predicate simplification 109
Predicate_Field QCD table 591
PreJCost cost profile flag 283
Product join 363, 364
cost 366
costing with dynamic partition elimination 369
definition 363
exclusion 364
exclusion product join with dynamic partition elimination 432
exclusion with dynamic partition elimination 425
full outer 364
inclusion 364
inclusion product join with dynamic partition elimination 429
inclusion with dynamic partition elimination 425
inner 364
left outer 364
piggybacked 364
processing 364
right outer 364
Pseudo-table locks
defined 750
properties 750

Q
QCD database 637, 641, 643, 645, 649, 654, 656
AnalysisLog table 560
AnalysisStmts table 562
applications 547
capacity planning 548
DataDemographics table 564
DBSCControl flags for tuning workload cache limits for
index and partition analysis 548
defined 549
deleting query plans 549
dropping QCD tables 553
Field table 566
Index_Field table 569
IndexColumns table 570
IndexMaintenance table 571
IndexRecommendations table 573
IndexTable table 578
information source 547
JoinIndexColumns table 582
macros 556
noncompatibility with pre-Teradata Database V2R5.0
releases 546
PartitionRecommendations table 585
physical model 550
Predicate table 589
Predicate_Field table 591
privileges for different user categories 557
QryRelX table 592
Query table 594
QuerySteps table 598
Index

RangePartExpr table 607
Relation table 609
SeqNumber table 615
setting up 551
StatsRec table 617
tables 559
TableStatistics table 619
User_Database table 622
UserRemarks table 623
ViewTable table 625
ways to query 555
Workload table 627
WorkloadQueries table 628
WorkloadStatus table 629
XMLQCD table 631
QCF
applications 547
QryRelX QCD table 592
Query Capture Database. See QCD
Query Capture Facility. See QCF
Query cost optimization 267
Query optimization
artificial intelligence, compared with 129
compiler optimization, compared with 129
derived statistics 206
objectives of 130
Query optimizer
described 128
need for 126
parallels with artificial intelligence systems 129
parallels with compiler optimization 129
types of 130
Query QCD table 594
Query Rewrite 18, 20, 34, 74
constant movearound and predicate simplification 113
constant predicate evaluation and predicate simplification 109
converting ANSI join syntax to comma join syntax 91
converting outer joins to inner joins 94
database object substitution 125
duplicate predicate removal and predicate simplification 113
NULLEQ marking and predicate simplification 113
predicate martialing 123
predicate push down and pullup 115
predicate simplification 109
processes 75
range-based simplification and predicate simplification 110
redundant join elimination 118
satisfiability and transitive closure 57, 102
techniques 90
view folding 56
view materialization 125
Query Rewrite subsystem 54
block diagram 55
component processes 56
Query Session utility
blocked lock requests 746
QuerySteps QCD table 270, 598
Queue table
locking issues with consume mode operations 769
locks not granted if request delayed 739
WRITE locks and consume mode operations 722
R
Random AMP sampling 141, 142, 178, 188, 294
accuracy 254
estimated statistics 293
estimating cardinality from a multi-AMP sample 182
estimating cardinality from a single-AMP sample 181
NoPI tables and random AMP sampling 144
random AMP sampling and NoPI tables 144
Range-based simplification 110
RangePartExpr QCD table 607
RASHistAbsDeviation 260
RASHistAbsDeviation cost profile flag 281
RASHistDeviationPct 260
RASHistDeviationPct cost profile flag 281
READ locks
described 724
Reasonable indexed join
cardinality estimation 457
uniqueness statistics 457
Reasonable LT/ST indexed join 448
Reasonable LT/ST unindexed join 449
Reasonable unindexed LT/ST join 449
Recollecting sampled statistics 177
Recollecting statistics 190
Red tree 33
Relation QCD table 609
Remote nested join 412
Request cache 23, 67, 69
management structure 25
matching process 26
peeking at USING and date-based built-in function values 36
purging 28
statement parsing and 23
Request cache peek 18, 20, 23, 25
Request cache peek subsystem 36, 69
DateSpecific generic plan 37
DateSpecific specific plan 37
generic plan 36, 37
specific plan 36, 37
Request cache peeking
block diagram 41
component process 45
class component process terminology 43
class component processes 45
disabling 40
effects on EXPLAIN 485
effects on INSERT EXPLAIN and DUMP EXPLAIN 657
enabling 40
Residual statistics
  relative accuracy 254
Resolver 18, 20, 24, 32, 33, 56, 58, 75, 80
  block diagram 33
  component processes 34
  component processes of 34
described 33
ResTree 20, 33, 56, 58, 75
ResTree’ 20, 58, 80, 132, 136
RevertEstimationLogic cost profile flag 282
Right outer product join 364
Right-deep search tree branches for join order optimization 330
Rollback processing
  ANSI session mode 792
  Teradata session mode 791
  Transient Journal 783
transient journal 699
Rolling columns
  definition 258
Row hash locks
  cannot lock multiple row hashes 759
described 719
RowID join 434
Rowkey-based merge join 383
  supported join methods 383
Rowkey-Based Merge Join and NoPI tables 383
Rule-based query optimization 130

S
Sampled statistics 175
  all AMPs 141, 142
  random AMP 141, 142
Satisfiability and transitive closure 38, 57, 102
Search trees
  bushy 327
  left-deep 326
Security checking
  described 35
Selecting indexes for a star join 452
SeqNumber QCD table 615
Session management
  blocked requests 746
Session modes
  2PC 776
  ANSI 774, 777, 779
rollback processing in ANSI session mode 792
rollback processing in Teradata session mode 791
Teradata 774, 783, 785
Session modes and transaction semantics 774
SET table kind option and duplicate rows 784
SHARE locks
  described 724
Single window merge join 386
Skew
  attribute value skew 165
  definition 165
  effects on hash join processing 401
  NoPI tables and skew 165
  partition skew 165
  Random AMP sampling
    NoPI tables and random AMP sampling 178
SkewAllowance 402
Sliding window merge join 389
Sliding Window Merge Join and NoPI tables 393
Sliding Window Merge Join with dynamic partition elimination and NoPI tables 394
Slow path local nested join 408
Snowflake join
  definition 443
  optimization 442
Snowflake join. See LT/ST join
SoftUniquePct cost profile flag 282
Specific plan 36, 37
SpoolFillFactor cost profile flag 285
SpoolLimitPercent cost profile flag 285
SQL flagger 32, 34
SQL parser 18
SQL statements
  COLLECT DEMOGRAPHICS 551, 552
  COLLECT STATISTICS (Optimizer Form) 649
  COLLECT STATISTICS (QCD Form) 645, 654
  COLLECT STATISTICS (QCD form) 551, 552, 576
  CREATE INDEX 576, 649
  DROP INDEX 576
  EXPLAIN request modifier 484
  INITIATE INDEX ANALYSIS 560, 574, 576, 585, 629, 640, 646, 647, 649, 650, 651, 652
  INITIATE PARTITION ANALYSIS 639, 640, 646, 647, 650, 651, 652, 664, 665
  INSERT EXPLAIN 551, 552, 645
  INSERT EXPLAIN ... WITH STATISTICS 643
  LOCKING request modifier 756
Stale statistics
  and derived statistics 251
  effects on join planning 333
Star join
  definition 442
  examples 455
  indexed join 447
optimization 442, 444
optimizer 444
selecting indexes 452
selecting indexes for 452
types 447
unindexed join 447
Star join optimization
miscellaneous considerations 450
Star join. See LT/ST join
Static columns
definition 258
Static partition elimination 298, 299, 300
Statistical profiles and the optimizer 193
Statistics
accuracy 174, 188
all AMPs sampling 141, 142, 188
all null 152
average AMP rows per value 157
collection time as a function of cardinality 149
collection time as a function of number of distinct values 149
collection time as a function of system configuration 149
estimating cardinality from a multi-AMP random sample 182
estimating cardinality from a single-AMP random sample 181
example of how stale statistics can produce poor query plans 150
extrapolating an open range date predicate case 3.1 263
extrapolation 255
extrapolation case 1 261
extrapolation case 2 262
how statistics are collected by the AMPs 152
interval histogram terminology 162
interval histograms 161
minimum column set for collecting statistics 192
minimum column set for collecting statistics by table size 191
null 152
performance overhead 142, 148
policies for collecting 191
possible effect of large DECIMAL and BIGINT values on their usefulness 174
purposes 141
random AMP sampling 141, 142, 176, 178, 294
ranking methods of collection for accuracy 188, 189
recollecting sampled statistics 177
regular row statistics 152
relative accuracy of various methods of collecting 188
residual 254
sampled 175
synopsis data structures 165
Teradata Statistics Wizard utility 151
terminology for extrapolating statistics 257
time overhead 142, 148
values collected and computed 193
when to collect 190, 191
when to recollect 190
Statistics Wizard utility 151
StatsRec QCD table 617
Steps
defined 63
Steps generator 20, 66
Support tools
Teradata System Emulation Tool (TSET) 681
VEComp facility 681
Query Capture Facility. See also QCF
Support tools, VEComp utility 545, 548
SynAct 30
Synopsis data structures 161, 165
Syntaxer 18, 20, 30, 34
time overhead 142, 148
values collected and computed 193
when to collect 190, 191
when to recollect 190
Statistics Wizard utility 151
StatsRec QCD table 617
Steps
defined 63
Steps generator 20, 66
Support tools
Teradata System Emulation Tool (TSET) 681
VEComp facility 681
Query Capture Facility. See also QCF
Support tools, VEComp utility 545, 548
SynAct 30
Synopsis data structures 161, 165
Syntaxer 18, 20, 30, 34
block diagram 31
compartment processes 32
compartment processes of 32
components 30
described 30
SynTree 20, 30, 34, 136
T
Table locks
described 719
Table lookahead in join planning 349
TableFillFactor cost profile flag 285
TableStatistics QCD table 619
Tactical queries
ACCESS locks 766, 794
ACCESS locks with joins 798
automatic lock escalation 795
complex update operations 800
group AMP steps 797
locking for simple update 799
locking for update 796, 799
locking issues 793
row hash-level ACCESS locks and join indexes 796
row hash-level versus table-level ACCESS locks 794
Target Level Emulation Facility 679
benefits of emulation 680, 682
capturing cost data 680
capturing Optimizer environmental cost information 687
capturing random AMP sample statistical data 680
cost profiles 690
dynamic emulation 681
enabling 684
forms 680
mapping files to a test system 693
procedures for capturing target system optimizer cost values and random AMP statistical samples 685
related tools and utilities 681
static emulation 681
SystemFE.Opt_Cost_Table 681
target system 680
Target system (production system). See Target Level Emulation
Teradata Index Wizard utility 59, 574, 629, 630, 635, 642, 643, 646, 649, 654, 656
AnalysisLog table 560
DataDemographics table 564
how used 635
IndexRecommendations table 573
Teradata session mode 774, 783
rules 783
transaction processing case studies 785
Teradata Statistics Wizard utility 151
determining when statistics should be recollected 190
Teradata System Emulation Tool utility 641, 644, 654
Transaction modes
2PC 776
ANSI mode
rules 777
ANSI mode case studies 779
Teradata mode case studies 785
Teradata mode rules 783
two-phase commit 776
Transaction properties
atomicity 698
consistency 698
consistency vs. correctness 805
durability 698
Transaction semantics
ANSI session mode 774
Teradata session mode 774
Transactions
ACID properties 698
atomicity property 698
consistency property 698
consistency vs. correctness 805
durability property 698
implicit locking 722
isolation property 698
multistatement blocked locking requests 748
rolling back 699, 783, 791
single-statement blocked locking requests 747
transient journal 699
Transient Journal and 783
Transient Journal 699, 783
Trees
bushy search 327
join order search 326
left-deep search 326
left-deep search branches 330
right-deep search branches 330
TSET utility 641, 644, 654, 681
Two-phase commit protocol 776, 778, 784, 803
Two-phase commit session mode 776, 778, 784, 803
Type 1 cost profile 276
Type 1 view folding 96
Type 2 cost profile 276
Type 2 view folding 96
U
Unindexed LT/ST join 449
Uniqueness statistics
reasonable indexed joins 457
Unreasonable indexed LT/ST join 448
Unreasonable unindexed LT/ST join 449
UPDATE (Upset Form)
EXPLAIN request modifier 534
Update processing
complex updates 800
UseActualBlockSize cost profile flag 286
User_Database QCD table 622
UserRemarks QCD table 623
USING-dependent parameterized requests 38, 39
USING-independent parameterized requests 38, 39
V
View folding 56, 96
type 1 76, 96
type 2 76, 78, 96
View locks
described 719
ViewTable QCD table 625
Visual EXPLAIN utility 641, 642, 654
W
White tree 21, 133, 135
Workload definition 643
Workload definition macros
AddWorkload 637
Workload QCD table 627
WorkloadQueries QCD table 628
WorkloadStatus QCD table 629
WRITE locks
described 724
X
XMLQCD QCD table 631