Teradata Database

# SQL Reference

## Statement and Transaction Processing

Release V2R6.2
B035-1142-096A
September 2006

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

# Preface

## Purpose

*SQL Reference: Statement 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 Database SQL transaction processing environment.

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

## Audience

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

## Supported Software Release

This book supports Teradata® Database release V2R6.2.

## Prerequisites

If you are not familiar with the Teradata Database, you will find it useful to read *Introduction to Teradata Warehouse* 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/Data Dictionary Quick Reference*.

# Changes to This Book

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

| Date | Description |
|---|---|
| September 2006 | • Clarified the use of the terms *statement* and *request* throughout the book.<br>• Added optimizer costing information to Chapter 2.<br>Some of this information was in a special release of V2R6.1, but it is new to any customer who did not receive the special V2R6.1 release of this manual.<br>• Added some join costing information to Chapter 2.<br>• Copied information on interpreting EXPLAIN reports from *SQL Reference: Data Manipulation Statements* to a new Chapter 4 in this book.<br>• Updated Chapter 5 for new PredicateKind code in Predicate table. Also revised description of Predicate table.<br>• Updated Chapter 8 to account for transient journaling now being maintained by WAL. |
| November 2005 | • Added new dynamic hash join type to Chapter 2.<br>• Removed all references to cost-related DIAGNOSTIC statements from Chapter 2.<br>• Updated Chapter 4 to account for new QCD features and corrected some past errors.<br>• Removed all references to cost-related DIAGNOSTIC statements and all procedures that use them from Chapter 6.<br>• Revised and expanded material on locking and transaction processing in Chapter 7.<br>• Added a new Appendix B for references to research literature on query optimization and transaction processing.<br>• Added and updated numerous glossary entries.<br>• Performed DR fixes and enhancement requests. |
| November 2004 | • Revised parser diagram and associated text to deal with array support for DML operations in Chapter 1.<br>• Added material about the queue table FIFO cache to the section on the Dispatcher in Chapter 1.<br>• Updated some existing optimizer-related material regarding cost profiles and added new material about them in Chapter 2.<br>• Added material on star join IN list optimizations in Chapters 2 and 3.<br>• Updated attributes for several QCD tables in Chapter 4.<br>• Added material on queue table transaction management in Chapter 7. |

# Additional Information

Additional information that supports this product and the Teradata Database is available at the following Web sites.

| Type of Information | Description | Source |
|---|---|---|
| Overview of the release<br>Information too late for the manuals | The Release Definition provides the following information:<br>• Overview of all the products in the release<br>• Information received too late to be included in the manuals<br>• Operating systems and Teradata Database versions that are certified to work with each product<br>• Version numbers of each product and the documentation for each product<br>• Information about available training and support center | http://www.info.ncr.com/<br>Click **General Search.** In the Publication Product ID field, enter *1725* and click **Search** to bring up the following Release Definition:<br>• *Base System Release Definition* B035-1725-096K. |
| Additional information related to this product | Use the NCR Information Products Publishing Library site to view or download the most recent versions of all manuals.<br>Specific manuals that supply related or additional information to this manual are listed. | http://www.info.ncr.com/<br>Click **General Search**, and do the following:<br>• In the Product Line field, select **Software - Teradata Database** for a list of all of the publications for this release.<br>• In the **Publication Product ID** field, enter a book number. |
| CD-ROM images | This site contains a link to a downloadable CD-ROM image of all customer documentation for this release. Customers are authorized to create CD-ROMs for their use from this image. | http://www.info.ncr.com/<br>Click **General Search.** In the Title or Keyword field, enter *CD-ROM*, and click **Search**. |
| Ordering information for manuals | Use the NCR Information Products Publishing Library site to order printed versions of manuals. | http://www.info.ncr.com/<br>Click **How to Order** under **Print & CD Publications**. |
| General information about Teradata | The Teradata home page provides links to numerous sources of information about Teradata. Links include:<br>• Executive reports, case studies of customer experiences with Teradata, and thought leadership<br>• Technical information, solutions, and expert advice<br>• Press releases, mentions and media resources | Teradata.com |

# References to Microsoft Windows

This book refers to "Microsoft Windows." For Teradata Database V2R6.2, such references mean Microsoft Windows Server 2003 32-bit and Microsoft Windows Server 2003 64-bit.

# Table of Contents

# CHAPTER 1 Statement Parsing

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

Topics include:

# 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 data

## Major Components of the Parsing Engine

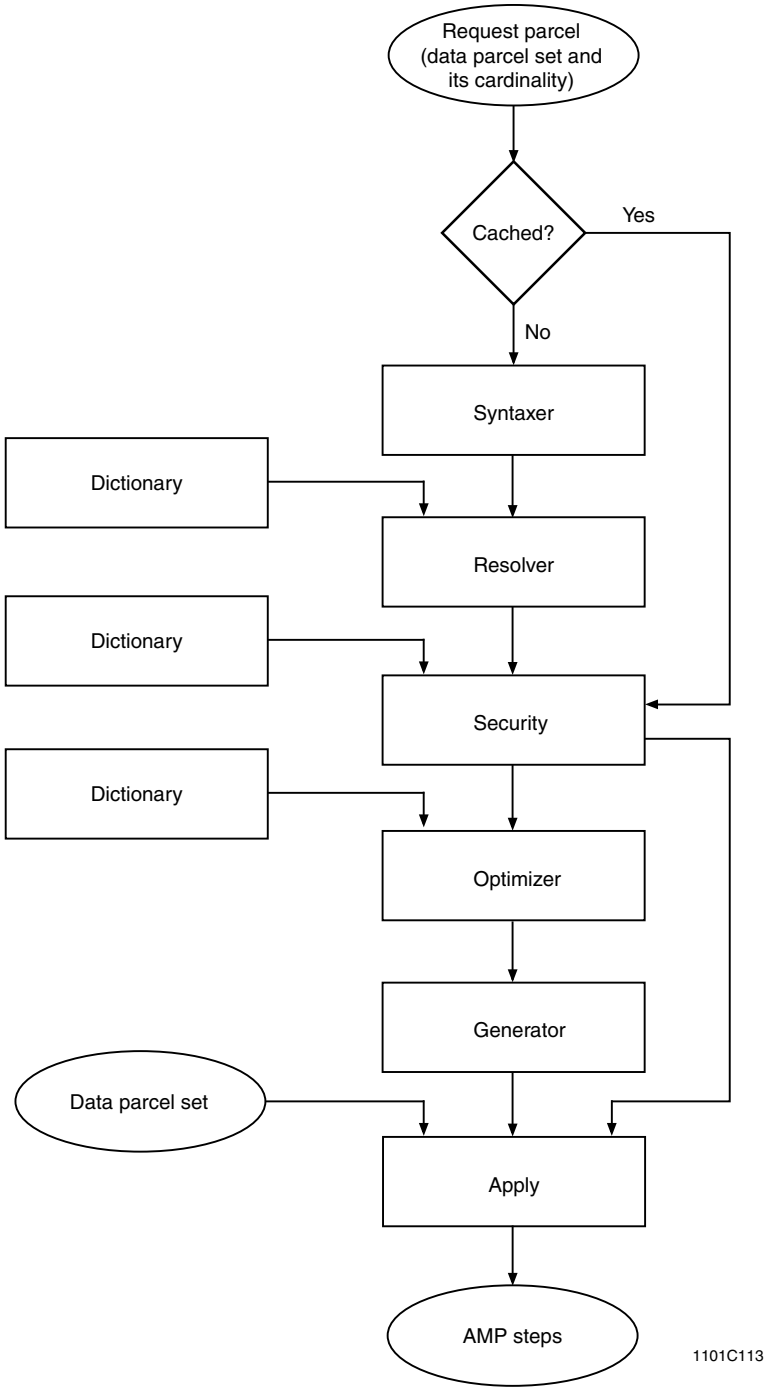
The parsing engine has four major and several lesser components related to query processing:

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

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

## Block Diagram of Parsing Engine Activity

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




The diagram does not include the Teradata Index Wizard because it is not used to process queries in a production environment. For more information about the Teradata Index Wizard, see Chapter 6: “Database Foundations for the Teradata Index Wizard.”

## Parsing Engine Component Processes

Parsing engine components perform the following functions for each SQL request sent to the server 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 41).

   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 the Optimizer.

   The ResTree is also referred to as the *Red Tree* for the query.

3. The Optimizer analyzes the ResTree to see if it can factor and combine the SQL statement components into a simpler, faster-performing form. It does this using various database statistics and configuration data to determine the optimum plans to access and join the tables specified by the request.

   The Optimizer then examines any locks placed by the SQL statement and attempts to optimize their placement to enhance performance and avoid deadlocks.

   The Optimized Parse Tree 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. See Chapter 2: "Query Optimization" and the documentation for the EXPLAIN modifier in *SQL Reference: Data Manipulation Statements* and *Teradata Visual Explain User Guide* for further information.

4. 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 Steps Packaging for further processing.

5. 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 USING modifier, then that data is also added to the steps tree. The final product of this process is referred to as Concrete Steps.

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 non-iterated 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.

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

7 The Dispatcher sequentially, incrementally, and atomically transmits the Concrete Steps, called AMP Steps at this point in the process, across the BYNET to the appropriate AMPs for processing.

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

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

10 End of process.

# 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 64 and "Environmental Cost Factors" on page 106), are used to convert database object names to their numeric IDs.

## Why Cache?

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

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

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

## Keeping the Cache Contents Fresh

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

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

## Use DBSControl to Fine Tune Cache

For each PE, 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.

# 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. The Syntaxer does not transform the query text in any way.

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

## Syntaxer Processing Rule

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

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

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

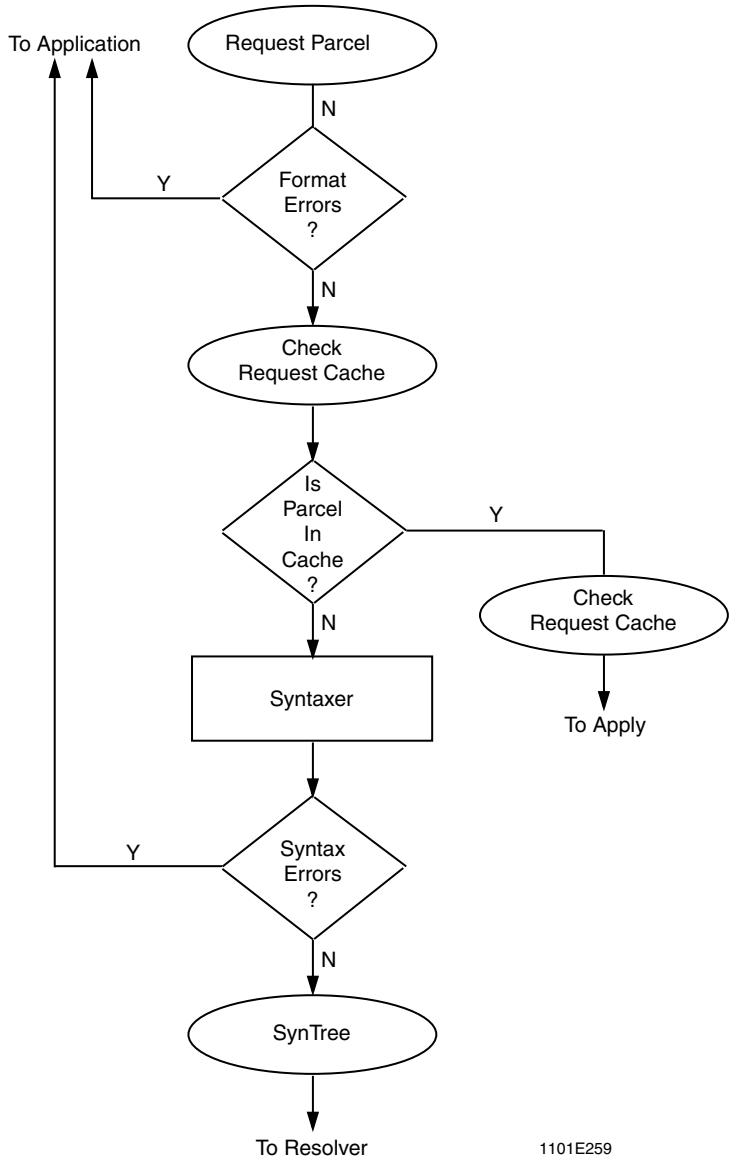
## Syntaxer Components

The Syntaxer is composed of the Lexer and Synact.

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

## Block Diagram of Syntaxer Activity

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



## Syntaxer Component Processes

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

| IF an identical parcel is … | THEN the Syntaxer … |
|---|---|
| found | 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.<br>Plastic steps are directives to the AMPs. The steps do not yet contain data values from the data parcel set (from the USING modifier).<br>Plastic steps are flexible, thereby allowing different sets of values (from the USING clause) to be inserted during subsequent operations. |
| not found | produces an initial parse tree, then calls the Resolver. |

2 If no matching parcel is found in cache and if no syntax errors are detected, then the Syntaxer generates a skeletal parse tree called a SynTree and passes it to the Resolver.

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

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

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.

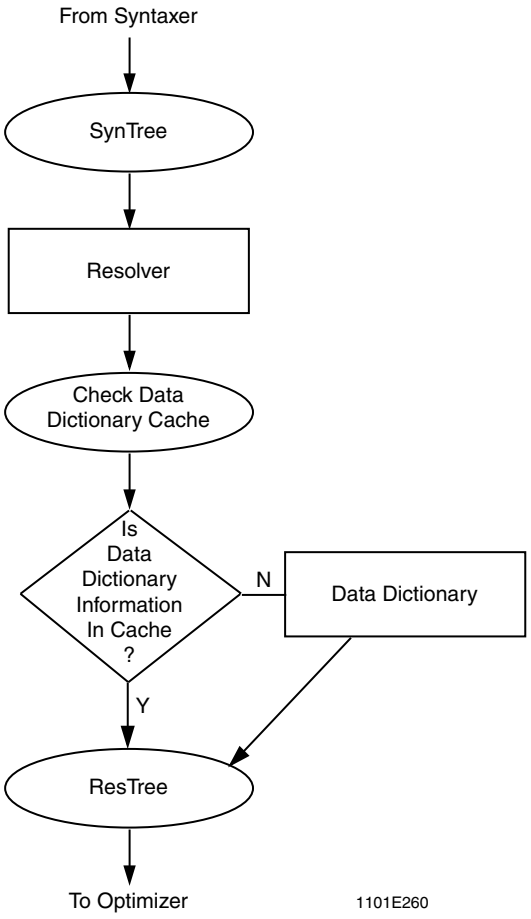
# Resolver

## 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 the Optimizer. The Resolver does not transform the query text in any way.

## Block Diagram of Resolver Activity

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



## Resolver Component Processes

1. Each database or user, table, view, trigger, stored procedure, and macro is assigned a globally unique numeric ID.

   Each column and each index is assigned a numeric ID that is unique within its table.

   These IDs are maintained in the data dictionary.
2. The Resolver refers to the data dictionary to verify all names and access rights and to convert those names to their equivalent numeric IDs.
3. The Resolver takes available information from the data dictionary cache, which is used on a least-recently-used or most-recently-used basis.

   If the needed information is not cached, it is retrieved from the appropriate system tables.
4. If a Request parcel contains views or macros, then the Resolver retrieves the view or macro text from the data dictionary, resolves it, and then merges the elements of the resulting tree into the request tree.

   This step is the point at which the majority of query rewrite processing occurs (see "Query Rewrite" on page 47).
5. 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. Database object access rights for cached requests are not revalidated when requests are processed from cache.

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

- *Security Administration*
- *Database Administration*
- *Data Dictionary*

# 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 Optimization."

- "Query Optimizers" on page 34
- "Translation to Internal Representation" on page 41
- "Query Rewrite" on page 47
- "Interval Histograms" on page 64
- "All-AMPs Sampled Statistics" on page 73
- "Random AMP Sampling" on page 76
- "When Should Statistics Be Collected Or Recollected?" on page 85
- "How the Optimizer Uses Statistical Profiles" on page 87
- "Examples of How Cardinality Estimates Are Made for Simple Queries Using Interval Histograms" on page 91
- "Cost Optimization" on page 101
- "Environmental Cost Factors" on page 106
- "Optimizer Join Plans" on page 108
- "Evaluating Join Orders" on page 135

Also see Chapter 3: "Join Optimizations" for information about how the Teradata Database optimizes join requests.

For information about the Teradata Index Wizard, which is also a component of the Optimizer, see "Chapter 6 Database Foundations for the Teradata Index Wizard" on page 379.

## What Sorts of Questions Does a Query Optimizer Ask?

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

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

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

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

- What is the degree of the table?

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

## How Does a Query Optimizer Answer the Questions It Asks?

Ideally, these questions are answered largely based on statistical data that you have generated using the SQL COLLECT STATISTICS statement. When database statistics are collected regularly, then you can 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 61 and "An Example of How Old Statistics Can Produce a Poor Query Plan" on page 62).

If *no* statistics have been collected, then the Optimizer makes a random snapshot sampling of data from a single AMP and uses that estimate to make a best guess about the optimum data retrieval path (see "Random AMP Sampling" on page 76).

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

## What Query Optimizers Do Not Do

Query optimizers do *not* do either of the following things:

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

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

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

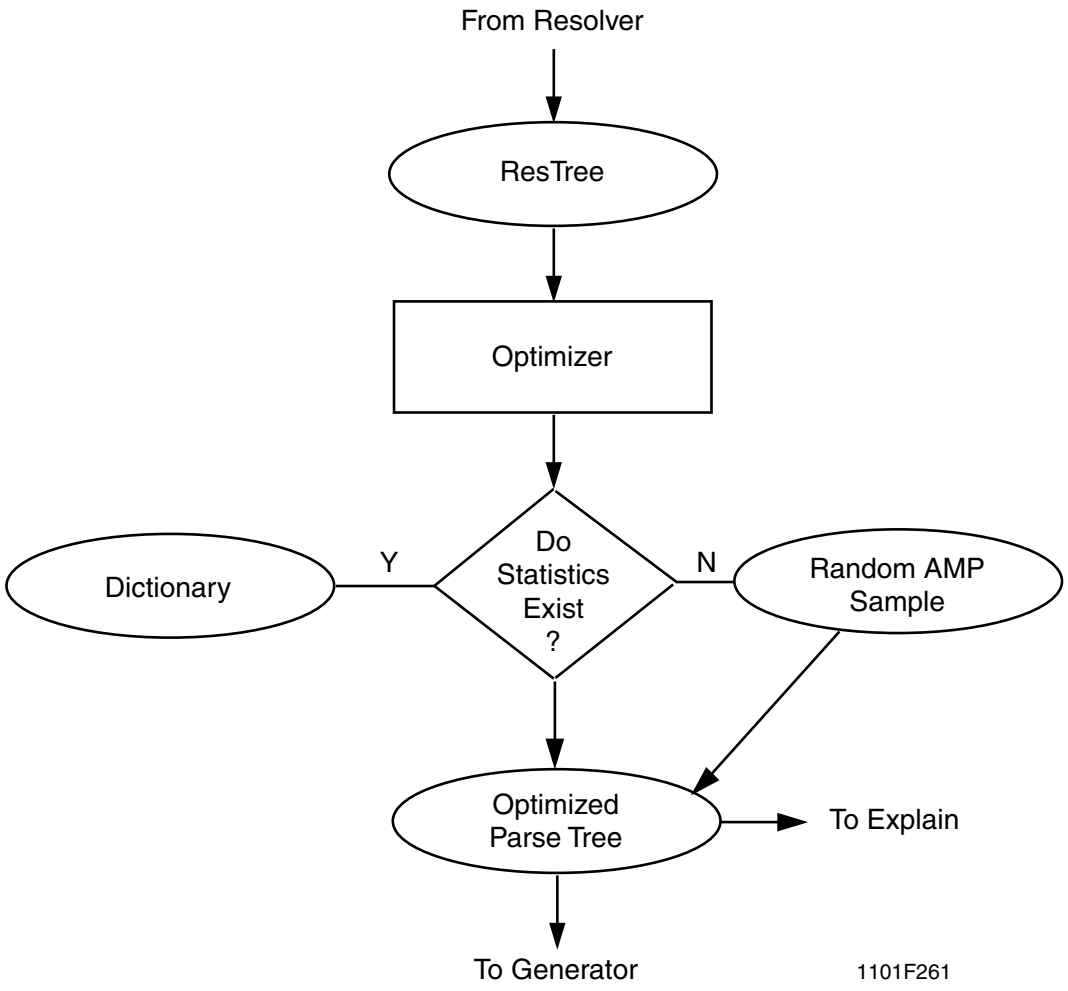
  You *should* assume that the query plan selected for use is more optimal than the otherwise unoptimized Resolver ResTree[3] formulation would have been.
- Rationalize poorly formed queries in such a way as to make their performance as effective as a semantically equivalent well-formed query that returns 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 47) to increase the efficiency of a given user-written query.

3. Also known as the Red Tree.

## Block Diagram of Query 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 17:



## Optimizer Component Processes

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

| IF the Request parcel contains this type of SQL statement … | THEN the Optimizer … |
|---|---|
| DDL or DCL | deletes the original statement from the parse tree after it has been replaced with specific data dictionary operations.<br>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. |
| DML | produces access paths and execution plans.<br>The Optimizer then uses whatever statistical information it has, whether complete or sampled, to determine which access paths or plans are to be used.<br>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. |

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 table statistics it has to work with.

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

5 Finally, the Optimizer passes its fully optimized parse tree on to the Generator for further processing.

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.

More information about plastic steps is in “Generator” on page 20.

More information about concrete steps is in “Apply” on page 28.

## Definition: Parallel Steps

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

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

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

## Definition: Common Steps

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

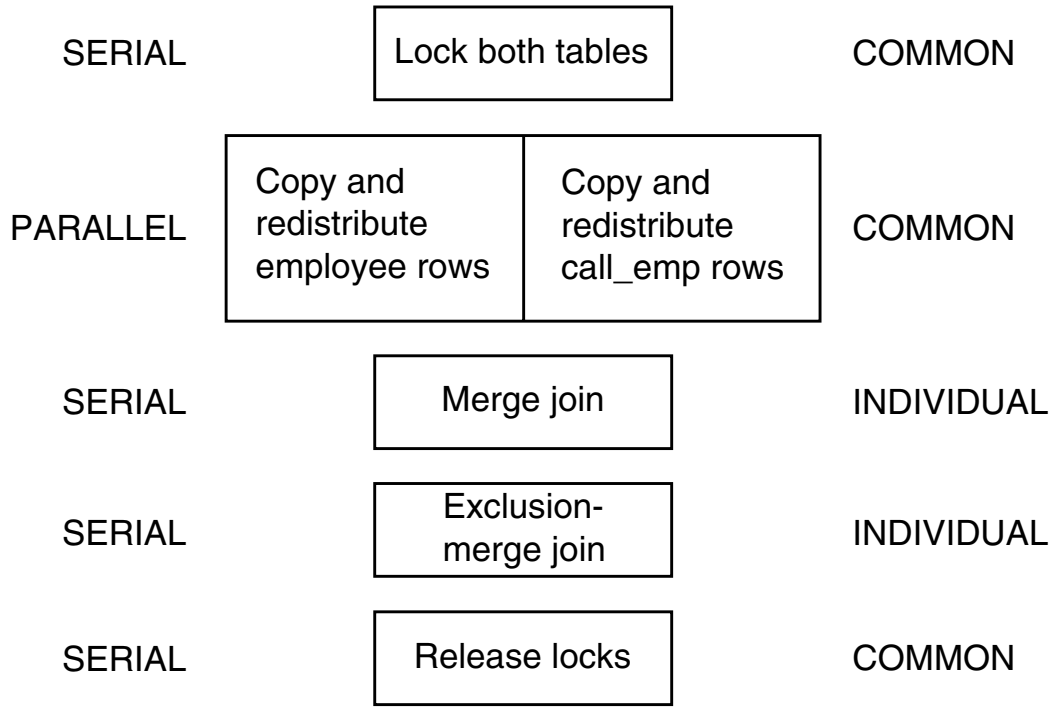
For example, consider the following multistatement request parcel.

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

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

| Stage | Process | Processing Mode | Step Type |
|---|---|---|---|
| 1 | Both tables (employee_number and call_emp) are locked. | Serial | Common |
| 2 | The rows from the employee_number table are copied and redistributed.<br>The rows from the call_emp table are copied and redistributed. | Parallel | Common |
| 3 | Results are merge joined. | Serial | Individual |
| 4 | Results are exclusion merge joined. | Serial | Individual |
| 5 | The table locks are released. | Serial | Individual |
| 6 | End of process. | None | None |

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




FF03A008

# Generator

## Introduction

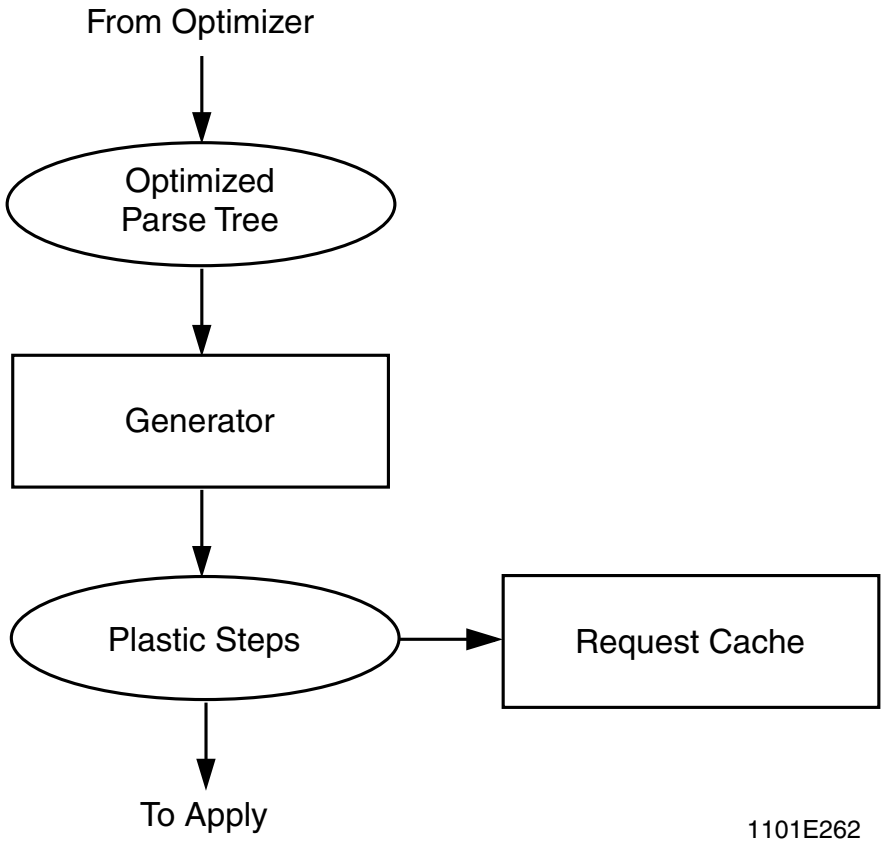
The Generator formulates the AMP processing steps, called plastic steps, based on the optimized parse tree plan it receives as the output of the Optimizer.

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

A single request parcel can generate many plastic steps.

## Block Diagram of Generator Activity

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

| IF the plastic steps were generated as the result of ... | THEN they are sent to ... |
|---|---|
| a real data-containing request parcel | Apply and dispatched across the BYNET to the AMPs. |
| an EXPLAIN modifier request | the user in the form of a report on steps taken and the projected speed of their execution. |

4. End of process.

## Definition: Plastic Steps

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

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

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

# Request Cache

## Definition

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

## The Value of the Request Cache

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

## The Role of the Request Cache in Statement Parsing

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

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

## Immediate Caching

Not all requests are cached, and not all cached requests are cached immediately. If a request has a data parcel,[4] it is cached immediately. Data parcels contain user-supplied information for a USING modifier.

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

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

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

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).

4. In this chapter, the term data parcel always refers to a data parcel set. A non-iterated 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.

## Non-Immediate Caching

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

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

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

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

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

## The Cache Management Structure

Each statement cache consists of two major entities:

- A data structure to manage the cached requests

  This structure 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: it is only as large as necessary.
- The individual requests

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

  Individual cached requests are stored in PE memory.

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

## Criteria For Caching a DML Request

The Teradata Database uses the following criteria to determine whether to cache a DML request or not:

| If a DML statement has … | THEN its steps are … |
|---|---|
| a USING modifier | cached the first time the request is parsed. |
| no USING modifier | not cached the first time the request is parsed.<br>If it is parsed a second time, then the steps are cached. |

## Criteria for Matching New Requests to Cached Requests

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

The new request must:

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

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

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

## Request Cache Matching Process

This process 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
   - Request length
   - Other flags
2. The Parser tests the preliminary variable list for the new request against the same information for the requests in the statement cache.

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

3 End of process.

# Purging the Request Cache

## Introduction

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

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

## Nonexempt Requests

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

## Exempt Requests

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

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

## Purging a Full Cache

The maximum number of possible entries in the request cache depends on the setting for the MaxRequestsSaved flag in the DBSControl 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 request cache is purged of all entries whose access or join plans are dependent on statistics. Entries that use only unique indexes for access are not affected by this periodic purge.

## Purging Individual Request Cache Entries

Request cache entries are purged individually under the following conditions:

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

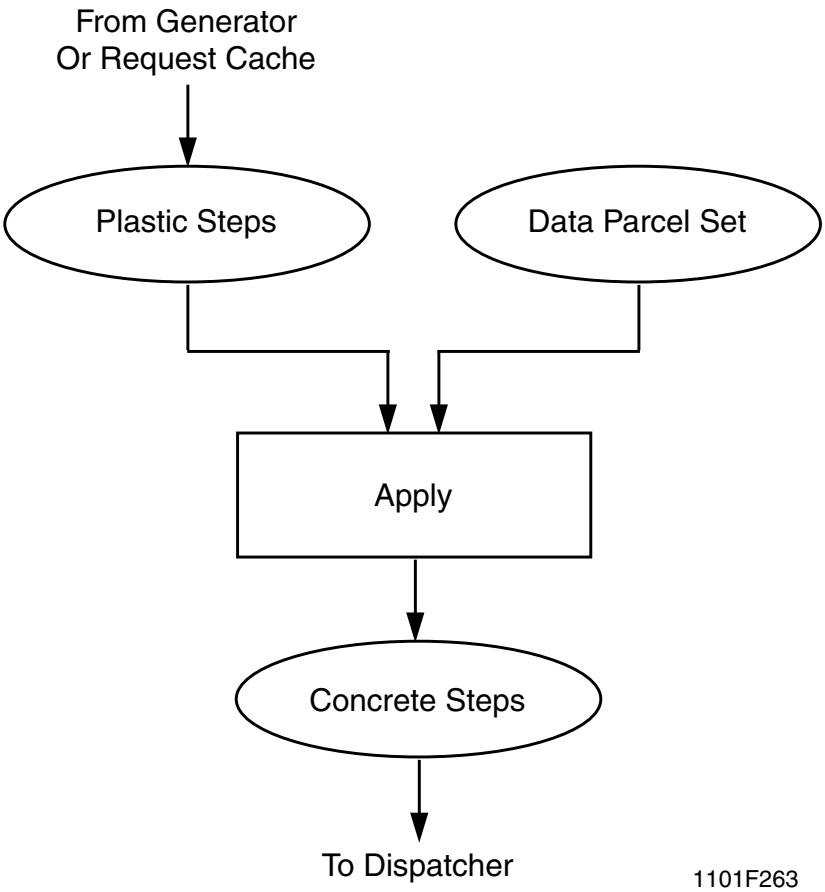
# Apply

## Introduction

Apply produces concrete steps by placing data from a USING data parcel set, if present, into the plastic steps. Concrete steps are passed to the Dispatcher, which then distributes them to the AMPs via the BYNET.

## Block Diagram of Apply Activity

The following diagram illustrates Apply activity:



## Apply Component Processes

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

1. Apply receives the plastic steps from the Request Cache or from the Generator.
2. The plastic steps are compiled into executable machine code.
3. Apply retrieves any data parcels associated with the Request parcel and applies them to the plastic steps, creating concrete steps.

   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, containing user- and session-specific information in addition to Data parcels.

Each concrete step is composed of four parts:

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

# Dispatcher

## Introduction

The Dispatcher has three primary functions:

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

## Execution Control

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

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

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

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

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

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

## Response Control

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

## Transaction and Request Abort Management

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

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

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

Typical normal aborts are caused by the following actions.

- TDP logoff command
- Client application terminates normally

Typical abnormal aborts are caused by the following actions:

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

## 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.

Cache row entries are shared by all queue tables that hash to that PE. Each queue table row entry is a pair of QITS and rowID values for each row to be consumed from the queue, sorted in QITS value order. The entry for a given queue table exists only in 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 (see *SQL Reference: Statement and Transaction Processing*) partition by hashing its table ID using the following algorithm:

```
queue_table_cache_PE_number = table_ID[1]MOD(number_system_PEs)
```

where:

| Syntax element …| Specifies the …|
|---|---|
| queue_table_cache_PE_number | position number within the configuration map array of the PE containing the queue table cache to which the given queue table entry is assigned. |
| tableID[1] | numeric value of the second word of the double word tableID value (where the first word is notated as tableID[0]). |
| MOD | modulo function. |
| number_system_PEs | number of PEs in the system configuration. |

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

```
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 the description of the CREATE TABLE (Queue Table Form) statement in *SQL Reference: Data Definition Statements* for more information about the queue table FIFO cache and its operation.

# CHAPTER 2 Query Optimization

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

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

Among the topics covered are the following:



This chapter does not describe the Teradata Index Wizard, which is also a component of the Optimizer. For information on the Teradata Index Wizard, see Chapter 6: "Database Foundations for the Teradata Index Wizard."

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 Optimizers

## Introduction

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

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

## Why Relational Systems Require Query Optimizers

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

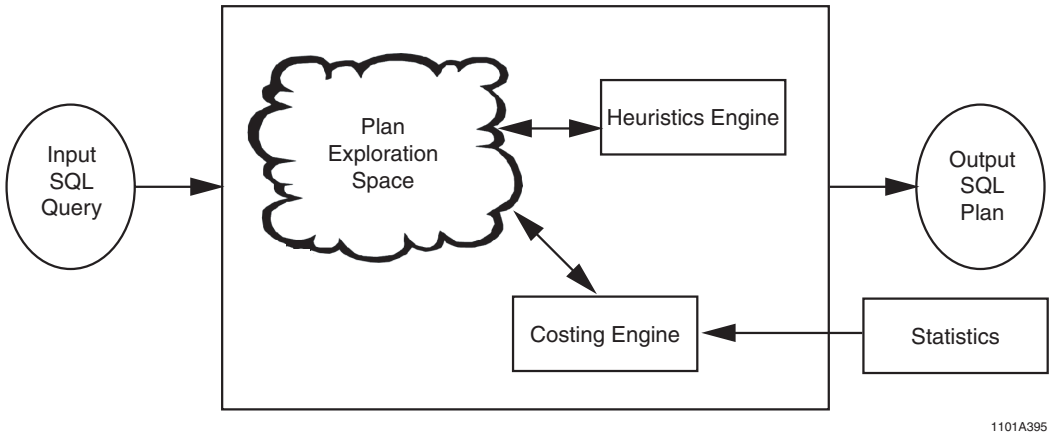
Queries in prerelational database systems do not require optimization because their access paths are determined in advance by the application programmer. The IBM hierarchical database management system IMS, for example, uses data structures called database descriptor blocks (DBD) and program control blocks (PCB) to define 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 high-level representation of an SQL query optimizer:




The input to the Optimizer, labelled as *Input SQL Query* in the graphic, is actually the ResTree[2] output from the Resolver, 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 116 and "Possible Join Orders as a Function of the Number of Tables To Be Joined" on page 119).

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 101).

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 59).

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.

Given this framework from which to work, 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*[3]).

2. Sometimes called the Red Tree.

3. 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 47 for more information about query rewrite.

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[4] are joined can have a powerful impact on join cost.

4 Generates several candidate query plans[5] 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.

## 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.

4. 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.
5. 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 "Statistics And Cost Estimation" on page 103 for a description of query plan costing).

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

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

## Parallels With Artificial Intelligence

SQL query optimization draws from two areas of artificial intelligence:

- Expert systems

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

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

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

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

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

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

## Objectives of Query Optimization

The principal objectives of SQL query optimization are the following:

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

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

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

## Types of Query Optimizers

There are two basic types of query optimizers:

- Rule-based

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

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

Although the behavior of both types converges in theory, in practice, rule-based optimizers have proven to be less performant in decision support environments than cost-based optimizers. The Teradata Database query optimizer is cost-based.[6]

What are the specific costs that a cost-based optimizer seeks to minimize? The Teradata Database 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.

6. 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.

# Teradata Database Optimizer Processes

## Introduction

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

## Query Optimization Processes

The following table shows the logical sequence of the processes undertaken by the Optimizer as it optimizes a DML request. The input to this process is the Resolver ResTree (see "Resolver" on page 10).

1. Marshal the predicates (see "Translation to Internal Representation" on page 41).
2. Perform preliminary processing of outer joins.
   a. When possible, convert outer joins to inner joins.
   b. Push down predicates.
   c. Build outer join terms.
   d. Generate information about dependencies between joined tables and their connecting conditions.
   e. End of sub-process.
3. Process correlated subqueries by converting them to non-nested SELECTs or simple joins.
4. Search for a relevant join or hash index.
5. Materialize subqueries to spool files.
6. Analyze the materialized subqueries for optimization possibilities.
   a. Separate conditions from one another (see "Predicate Marshaling" on page 51).
   b. Push down predicates (see "Predicate Push Down and Pullup" on page 53).
   c. Generate connection information.
   d. Locate any complex joins.
   e. End of sub-process.
7. Generate size and content estimates of spool files required for further processing (see "How the Optimizer Uses Statistical Profiles" on page 87).
8. Generate an optimal single table access path.
9. Simplify and optimize any complex joins identified in 6d.
10. Map join columns from a join (spool) relation to the list of field IDs from the input base tables to prepare the relation for join planning.

11 Generate information about local connections. A connecting condition is one that connects an outer query and a subquery. A direct connection exists between two tables if either of the following conditions is found.
   - ANDed bind; miscellaneous terms such as inequalities, ANDs, and ORs; cross, outer, or minus join term that satisfies the dependent information between the two tables
   - A spool file of an uncorrelated subquery EXIST predicate that connects with any outer table
12 Generate information about indexes that might be used in join planning, including the primary indexes for the relevant tables and pointers to the table descriptors of any other useful indexes.
13 Use a recursive greedy 1-table lookahead algorithm to generate the best join plan (see "One-Join Lookahead Processing of n-Way Joins" on page 140).
14 If the join plan identified in step 13 does not meet the heuristics-based criteria for an adequate join plan, generate another best join plan using an *n*-table lookahead algorithm (see "Five-Join Lookahead Processing of n-Way Joins" on page 143).
15 Select the better plan of the two plans generated in step 13 and 14.
16 Generate a star join plan (see "Star and Snowflake Join Optimization" on page 201).
17 Select the better plan of the selection in step 15 and the star join plan generated in stage 16.
18 Generate plastic steps for the plan chosen in step 17 (see "Definition: Plastic Steps" on page 21).
19 End of process.

# Translation to Internal Representation

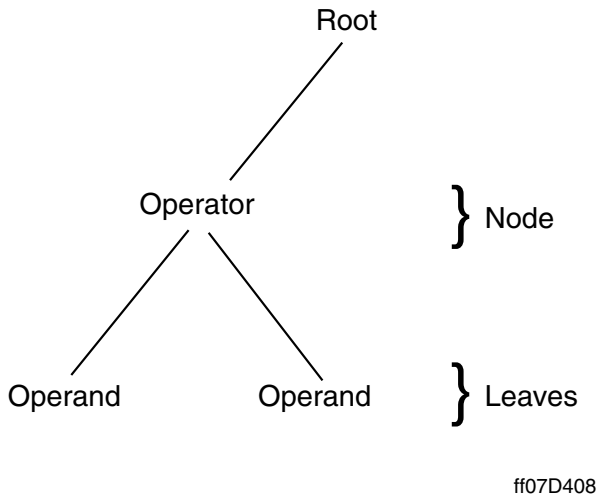
## Parse Tree Representations of an SQL Request

Like most cost-based optimizers, the Teradata Database 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.[7] 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 leafs of the parse tree.

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

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




7. 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:

**Parts**

| PartNum | Description | MfgName | MafPartNum |
|---|---|---|---|
| PK | | FK | |
| PI | | | |
| ND, NN | | | |

**Manufacturer**

| MfgNum | MfgName | MfgAddress | MfgCity |
|---|---|---|---|
| PK | | | |
| PI | | | |
| ND, NN | | | |

**Customer**

| CustNum | CustName | CustAddress | CustCity |
|---|---|---|---|
| PK | | | |
| PI | | | |
| ND, NN | | | |

**Order**

| OrderNum | CustName | MfgPartNum | OrderDate |
|---|---|---|---|
| PK | FK | | |
| PI | | | |
| ND, NN | | | |

## Example SQL Request

Consider the following example SQL request:

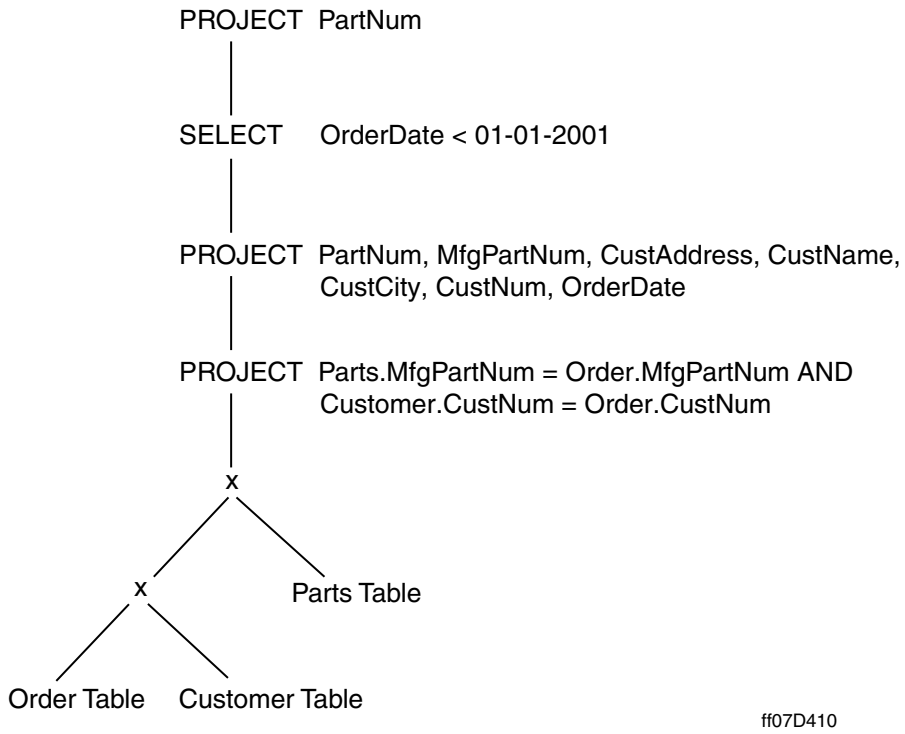
```
SELECT PartNum, Description, MfgName, MfgPartNum, CustName,
       CustAddress, CustNum, OrderDate
FROM Order JOIN Customer JOIN Parts
WHERE Customer.CustNum = Order.CustNum
AND   Parts.MfgPartNum = Order.MfgPartNum
AND   OrderDate < DATE '2001-01-01' ;
```

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. The Optimizer receives a processed parse tree as input from the Resolver. This request tree is just a parse tree representation of the original query text.

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

At this point, the original 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 this tree, also referred to as a white tree, that the Optimizer passes to you as EXPLAIN text when you explain a request.

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




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

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

The process involved in pushing down these predicates is indicated by the following 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.

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

   ```
   SELECT Customer.CustNum = Order.CustNum
   SELECT Parts.MfgPartNum = Order.MfgPartNum
   ```

2. By commutativity, the SELECTION Order.OrderDate < 2001-01-01 can be pushed the furthest down the tree, and it is pushed below the PROJECTION of the Order and Customer tables.

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

**a** Begin with the following predicate.

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

**b** Transform it to the following form.

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

**c** Transform it further to the following form.

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

This is as far as the predicate can be transformed, and it has moved as far down the parse tree as it can be pushed.

**d** End of sub-process.

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

```
SELECT Parts.MfgPartNum = Order.MfgPartNum
```

Because this SELECTION contains one column from the Parts table and another column from a different table (Order), it cannot be pushed down the tree any further than the position it already occupies.

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

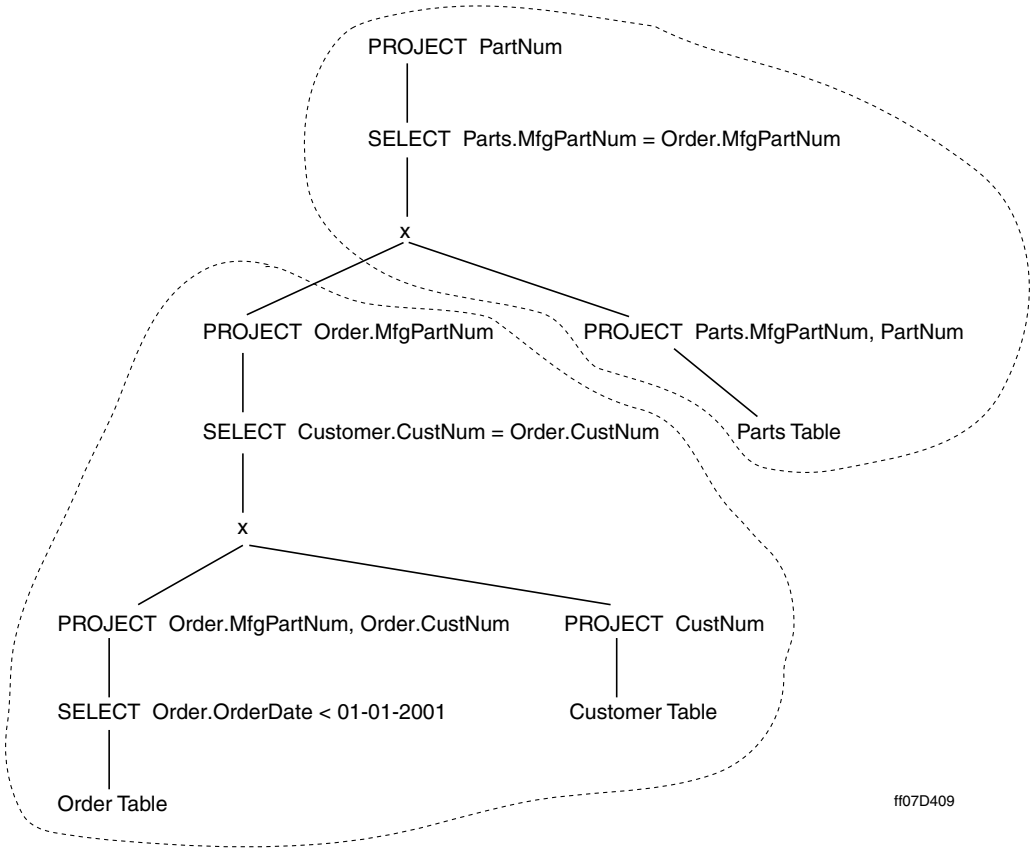
```
SELECT Customer.CustNum = Order.CustNum
```

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

Order.CustNum is an attribute in SELECT Date < 2001-01-01(Order) because the result of a selection accumulates its attributes from the expression on which it is applied.

**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.

```
PROJECT Parts.PartNum
SELECT Parts.MfgPartNum = Order.MfgPartNum
PROJECT Parts.PartNum, Parts.MfgPartNum, Order.MfgPartNum
```

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

```
PROJECT PartNum, Parts.MfgPartNum
```

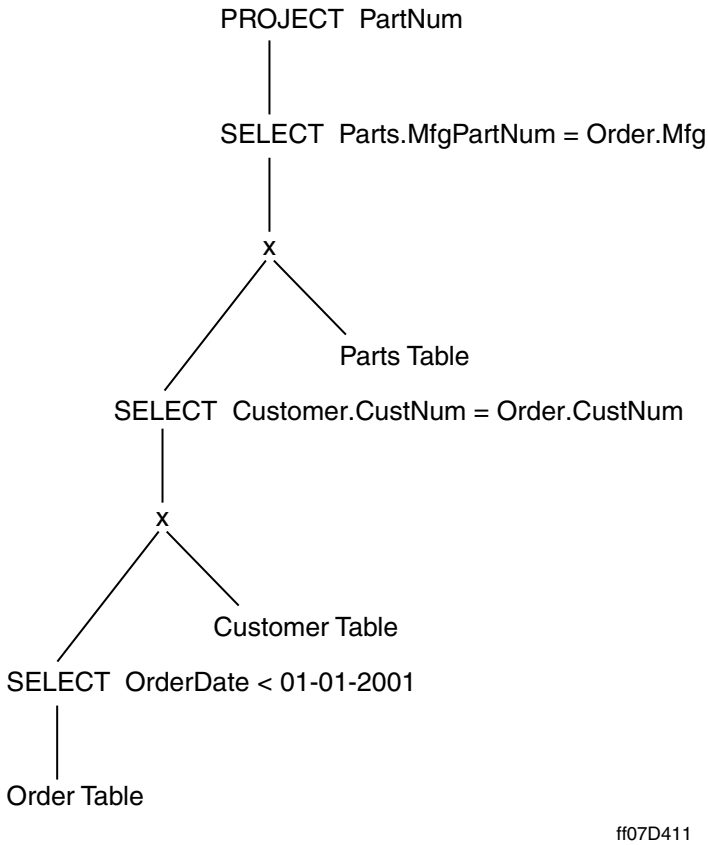
8 Similarly, apply PROJECT Order.MfgPartNum to the left operand of the higher of the two Cartesian products. This projection further interacts with the SELECT operation immediately below it (Customer.CustNum = Order.CustNum) to produce the following series of algebraic operations.

```
PROJECT Order.MfgPartNum
SELECT Customer.CustNum = Order.CustNum
PROJECT Order.MfgPartNum, Customer.CustNum, Order.CustNum
```

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.




**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.

# Query Rewrite

## Introduction

Query rewrite is arguably the most fundamental process undertaken by a query optimizer, and is the first piece of optimization performed in the query optimization process.[8] The fundamental motivation behind query rewrite is to flatten the code in order to make optimization of query components such as views, correlated subqueries, and aggregates more tractable for optimization.

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

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, query rewrite has become more crucial than ever.[9] The Optimizer rewrites queries aggressively in order to produce query plans that are not exceeded in the commercial relational database market.

## Typical Query Rewrite Techniques

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

8. Note that query rewrite is, in the strict sense, a process that precedes true query optimization. This is not just a peculiarity of the Teradata implementation, but is true universally. For the Teradata system, the vast majority of query rewrite steps are performed in the Resolver component of the Parser (see “Resolver” on page 10).
9. 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 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)! (Date, 1998c).

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

- View folding
- ANSI join syntax-to-comma join syntax
- Predicate marshaling (also called *predicate move around*)
- Predicate push down and pullup (also called *pushing conditions*)
- Outer join-to-inner join conversion
- Satisfiability and transitive closure
- Join elimination
- View materialization and other database object substitutions

## View Folding

View[10] folding is a process in which a query that references a view is rewritten without an explicit reference to that view.

For example, consider the following query:

```
SELECT *
FROM t1,(
  SELECT *
  FROM t2) AS dt
WHERE a1=a2;
```

Folding the derived table in this query produces the following equivalent rewritten query:

```
SELECT *
FROM t1,t2
WHERE a1=a2;
```

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

The Optimizer attempts to fold views whenever it can because folding a view provides it 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.

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

1. Removes the table reference of the folded view/derived table from the containing query block.
2. Removes the view reference of the folded view/derived table from the view list of the containing query block.
3. Merges the FROM clauses of the containing block and the view/derived table.

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.

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/derived table.
5 Merges the WHERE and HAVING clauses of the containing block and the view/derived table.
6 Sets the DISTINCT flag of the containing query block to TRUE if the view/derived table is a SELECT DISTINCT. This is done for simple view merging only.
7 Changes all references to the view/derived table SELECT list to references to the expressions these references map to in the assignment list of the view/derived table.
8 Moves any SAMPLE, QUALIFY, and GROUP BY clauses in the view/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:

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

The algorithm for folding views is driven by the conditions that 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
  - WHERE
  - HAVING
  - QUALIFY
  - ORDER BY
  - GROUP BY
  - and so on
- Main query can have only SELECT * or SELECT list of fields from the view

The main logic of the view folding optimization checks for a list of conditions to spool a view. If the searched conditions are not found, the system folds the view. The list of spooling conditions along with planned enhancements are summarized below. Note that the list below is checked if the query is not a "simple query" as defined above.

The conditions that invoke spooling if *any* of them exist 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 statement.

    The ABORT statement has 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 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 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/EXCEPT
  - 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.

## Converting ANSI Join Syntax To Comma Join Syntax

This rewrite converts ANSI-style inner join syntax to comma syntax if the entire query is based on inner joins. For example, consider the following query:

```
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:

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

## Predicate Marshaling

The first step in the process of query rewrite is to marshal the predicates, both ordinary and connecting, for the DML operations presented to the Optimizer by the vanilla Resolver ResTree. This means that first that the query predicates are, where possible, isolated from the parse tree, converted to conjunctive normal form, or CNF (see "Path Selection" on page 102 for a definition of CNF), and then they might be combined with other predicates or eliminated from the query altogether by a process analogous to factoring and cancellation in ordinary algebra.

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

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

where the symbol → 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.

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

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

Where possible, new conditions are derived using transitive closure[11]. For example,

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

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:

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

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

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

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

The following transformation is more complex.

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

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

11. Transitive closure is a relational algebraic method the Optimizer uses 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.

The next step is to push the marshaled predicates down the parse tree as far as possible. See "Translation to Internal Representation" on page 41 and "Predicate Push Down and Pullup" on page 53 for additional information on predicate push down and pullup.

## Predicate Push Down and Pullup

The concept of parse trees was introduced in "Translation to Internal Representation" on page 41, as was predicate push down.

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

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

A significant component of query rewrite 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.

For example, consider the derived table in the following query:

```
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 the view. This rewrite produces the following query:

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

Note that the original query would have performed a full file 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:

```
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 `SUM(a3)` term can be removed from the SELECT list of the view definition as far as this query is concerned. 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 `SUM(a3)`.

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

```
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 query if the references to t1 are removed from the select list of the derived table:

```
SELECT 1
FROM (SELECT *
      FROM t1,t2
      WHERE a1=a2) AS dt;
```

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

## Outer Join-To-Inner Join Conversion

Some left and right outer joins can be converted to inner joins if a query has at least one condition that filters out the nonmatching rows. For example, consider the outer join in the following query:

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

This outer join can be converted to an inner join because b2 < 10 is false if b2 is null, and therefore the condition removes all nonmatching rows of the outer join.

Similarly, a full outer join can be converted to a left or right outer joins, or even to inner joins given the appropriate conditions in the query.

## Satisfiability and Transitive Closure

Satisfiability is a term that describes checks the Optimizer performs to determine if a set of constraints is contradictory. 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 having the conditions `a=1` and `a=2` in the same query. If the Optimizer discovers such an unsatisfiable condition is, it simplifies and optimizes the condition in such a way that all joins and retrieves are done on single AMP basis.

One way to fix a contradictory condition is to add CHECK constraints to the query, allowing the Optimizer to find better execution plans. 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, the query looks like this:

```
SELECT *
FROM ordertbl
WHERE EXTRACT(month,o_orderdate) <= 3;
```

Without being able to add such constraints to a query during the rewrite phase, the system would access all of the tables orders1, orders2, … orders12 using the constraint `EXTRACT(month,o_orderdate) <= 3`, where it only needs to access orders1, orders2, and orders3 to satisfy the query. The only way the Optimizer 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, the Optimizer sees `EXTRACT(month,o_orderdate) <= 3 AND EXTRACT(month,o_orderdate) = 4`, which is a contradiction. For this particular case, the Optimizer can simply eliminate this step. In general, the Optimizer needs to know if a set of constraints is satisfiable.

Of course a user 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.

There are 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.

12. The term update operation here signifies an insert, update, or delete operation against the base table in question.

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,l_shipdate) <=6;

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

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

```
DELETE lineitem
WHERE EXTRACT(month,l_shipdate) = 12;
```

This implies that there is a need to update j2, but not j1. The system can make this decision because the satisfiability check returns TRUE for `EXTRACT(month,l_shipdate) = 12 AND EXTRACT(month,l_shipdate) >=7`, but FALSE for `EXTRACT(month,l_shipdate) = 12 AND EXTRACT(month,l_shipdate) <=6`.

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 once more complex conditions like constants in WHERE clause predicates are allowed in the join index definition.[13]

Transitive closure is a relational algebraic method the Optimizer uses to solve questions of the sort "can I get to point **b** from point **a**"?[14] The transitive closure of a set of constraints $\mathbf{S}_1$, denoted by TC($\mathbf{S}_1$), is the set of all possible derivable constraints from $\mathbf{S}_1$. For example if $\mathbf{S}_1$ is `(a=b AND a=1)` then TC($\mathbf{S}_1$) is `(b=1)`. The Optimizer finds transitive closure, but limited to simple cases like the previous example. The Optimizer only finds TC($\mathbf{S}_1$) if $\mathbf{S}_1$ is a conjunction of constraints, and it only derives new ones for a sequence of equality constraints.

In many decision support and CRM applications, transitive closure is needed for date ranges and IN clauses.

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 Reference: Data Definition Statements for more information about sparse join indexes.

14. This problem is often referred to as *Reachability* and the determination of its answer is called the Reachability problem.

The following example illustrates one of these cases:

```
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 this example, you can find the sequence of <= as follows:

```
S1, =    (l_shipdate <= l_commitdate-1
     AND l_commitdate <= l_receiptdate-1
     AND l_receiptdate <= '1994-06-05')
```

The new constraints that can be derived from $\mathbf{S}_1$ or TC($\mathbf{S}_1$) are the following set:

```
   (l_commitdate <=  '1994-06-04'
AND l_shipdate <= '1994-06-03')
```

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)AND(a=2)` is contradictory from the following:

```
TC{(a=1) AND (a=2)}={(1=2)}
```

Consider another example, which has the contradiction of `(2<=1)`:

```
TC{(a>=2 AND a<=b AND b<=1} = {b>=2 AND a<=1 AND 2<=1}
```

These examples suggest that satisfiability is a by-product of transitive closure.

## Redundant Join Elimination

Joins are among the most frequently specified operations in SQL queries. They are also among the most demanding resource consumers in the palette of an SQL coder. 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 both cases, a join and a table[15] 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, the 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 Reference: Data Manipulation Statements*. The topics "First Attempt" and "Third Attempt" are particularly good examples of how the Optimizer can streamline poorly written joins.

## View Materialization and Other Database Object Substitutions

Perhaps the most obvious query rewrites concern 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.

The Optimizer also replaces base tables and table joins with hash, join, or even secondary indexes whenever the substitution makes the query more performant. These substitutions also apply to view materialization[16] if base tables referenced by the view can be profitably replaced.

15. The table eliminated is the parent table in inner joins and the inner table in outer joins.

16. The term materialized view is sometimes used to describe database objects like snapshot summary tables, and hash or join indexes. For purposes of the current discussion, the term refers to materializing the base table components of a view definition in a spool relation.

# 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, average cardinality per value, average cardinality per index, average size of each index on each AMP, and the number of distinct index values plus a small set of system-defined heuristics for various predicate selectivities.

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

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

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 Old Statistics Can Produce a Poor Query Plan" on page 62 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.

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

In many cases, random all-AMPs (see "All-AMPs Sampled Statistics" on page 73) or random AMP sampled statistics (see "Random AMP Sampling" on page 76) are not accurate enough for the Optimizer to generate an optimal, or even a good, join plan. However, it is sometimes true that statistics collected by sampling small subpopulations of table rows can be as good as those collected with a full-table scan. The value of collecting full-table statistics is that they provide the Optimizer with the most accurate information that can be gathered for making the best possible query plan cost estimates.

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

A query plan generated using full-table statistics is guaranteed to be at least as good as a query plan generated using any form of sampled statistics[17].

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

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

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

17. 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.

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

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

## 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.[18]

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

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

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

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

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

18. 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.

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

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

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

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

Consider the following two tables:

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

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

In reality, table A currently has a cardinality 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 almost certain that the statistics for that column are stale.

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

```
HELP STATISTICS tablename;
```

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

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

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

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

## Teradata Statistics Wizard

The Teradata Statistics Wizard client utility provides an objective method of determining when fresh statistics should be collected on a table, column, or index based on a set of user-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* for additional information about the uses and capabilities of this utility.

# Interval Histograms

## Introduction

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

The 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. Each histogram is composed of a maximum of 100 intervals.[19]

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

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

See Ioannidis (2003) for a brief history of histograms and their use in query optimization for relational database management systems.

## Types of Interval Histograms Used By The Teradata Database

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

- Equal-height interval histogram

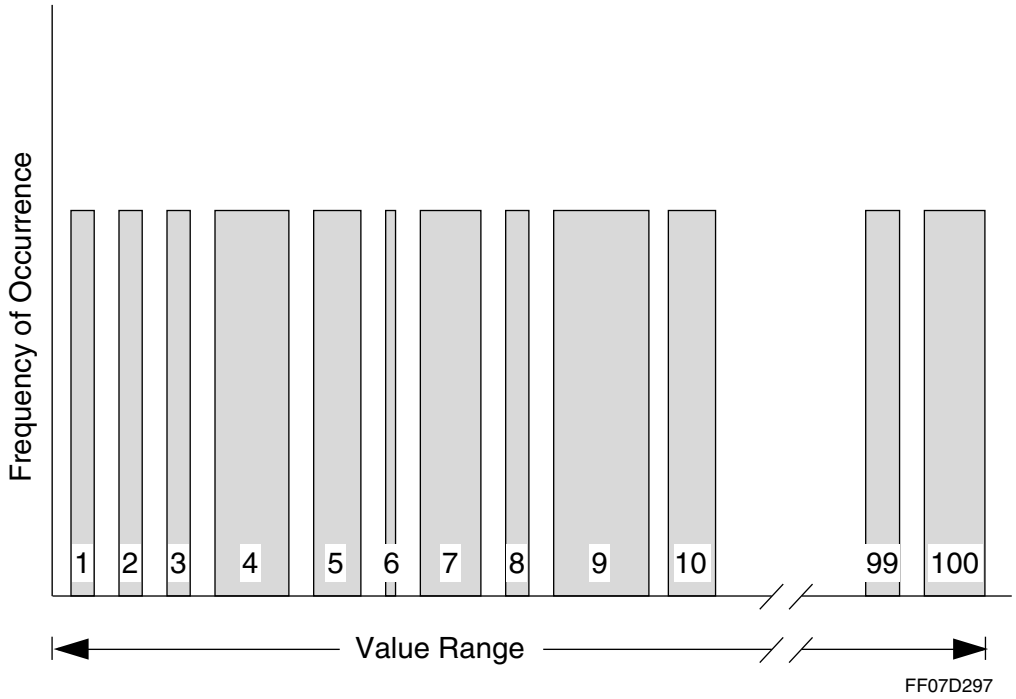
  In an equal-height interval histogram, each interval has the same number of values. To achieve this, the interval widths must vary.

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

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

19. The number of intervals used to store statistics is a function of the number of distinct values in the column or index represented. For example, if there are only 10 unique values in a column, the system does not store the statistics for that column across 100 interval histograms. 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 for which statistics are being captured equals or exceeds the maximum number of intervals.

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




- High-biased interval histogram

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

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

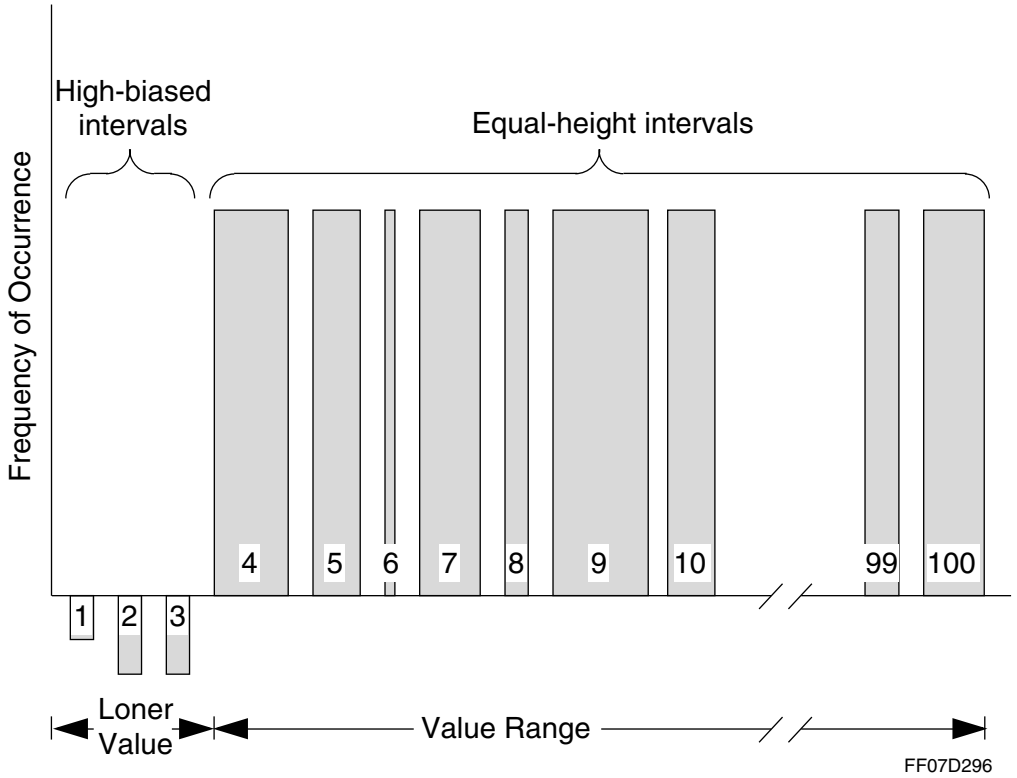
| WHEN the interval stores this many loners … | THEN values is set to … |
|---|---|
| 1 | -1 |
| 2 | -2 |

- Compressed histogram

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

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

The following graphic illustrates the concept of a compressed histogram:




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

## Terminology

| Term | Definition |
|---|---|
| Bucket | A synonym for an interval in an interval histogram. |
| Cardinality | The number of rows per AMP that satisfy a predicate condition.<br>Note that in this context, cardinality generally is not the number of rows in the entire table, but the number of rows that qualify for a predicate filter. See "Examples of How Cardinality Estimates Are Made for Simple Queries Using Interval Histograms" on page 91 for a fuller explanation of what cardinality means in this context. |
| Compressed interval histogram | A family of histograms that combines high-biased intervals with equal-height intervals.<br>The high-biased intervals are always stored in buckets 1 through *n*, where *n* represents the highest numbered bucket containing high-biased interval information.<br>Equal-height intervals begin at bucket *n* + 1. |

| Term | Definition |
|---|---|
| Equal-height interval | An interval containing column statistics normalized across the distribution in such a way that the graph of the distribution of the number of values as a function of interval number is flat.<br>This is achieved by varying the width of each interval so it contains the same number of values (but usually different attribute value ranges) as its neighbors. |
| Equal-height interval histogram | A family of histograms characterized by equal numbers of occurrences and non-constant attribute value ranges per bucket. |
| High-biased interval | An interval used to characterize a skewed value set for a column.<br>Any attribute value that is significantly skewed (see the statistic defined under "Loner" on page 68) is summarized by a high-biased interval.<br>Each high-biased interval contains statistics for at most two attribute values. |
| High-biased interval histogram | A family of histograms characterized by many loner buckets as seen, for example, with a multimodal or otherwise-skewed attribute value distribution. |
| Histogram | A graphic means for representing distributions as a function of the number of elements per an arbitrarily determined interval width.<br>Histograms are often called bar charts. Each bar in a histogram represents the frequency of occurrence of the value or range of values for the defined interval. Histogram intervals are sometimes referred to as buckets because they contain a number of values that summarize the demographics for the values that fall into the range defined for the interval.<br>In Optimizer theory, the term is used to describe the rows in a dictionary table that store the buckets defined by the particular intervals used to characterize the frequency distribution of the attribute values for a column.<br>All histograms described in this topic are frequency histograms. Each bucket in a frequency histogram contains some number of tokens representing the frequency of occurrence of the attribute values belonging to its range. |
| Interval | A bounded, non-overlapping set of attribute values. |

<table>
<tr><th>Term</th><th>Definition</th></tr>
<tr><td>Loner</td><td>A loner is an attribute value whose frequency in the sampled population deviates significantly from a defined criterion; an unusually frequent value indicating significant frequency skew. By definition, no more than two loners are stored per high-biased interval.<br>The formal definition of a loner is as follows:<br>$f \geq \frac{T}{200}$<br>where:<table><tr><th>This variable ...</th><th>Defines the ...</th></tr><tr><td>f</td><td>frequency of the loner.</td></tr><tr><td>T</td><td>cardinality of the base table.</td></tr></table>By this definition, the maximum number of loners in a histogram is 200. In practice, a column might produce more than 200 loners, so the implemented upper bound on loners is 198 (or 99 intervals). This leaves one equal-height interval to account for the remaining values in the distribution.</td></tr>
<tr><td>Skew</td><td>A measure of the asymmetry of the distribution of a set of attribute values.<br>Skewness is the third moment of the probability density function for a population of attribute values.<br>The first two moments are the mean and the standard deviation, respectively.<br>With respect to skew in parallel databases, there are several possible types.<ul><li>Attribute value skew refers to skew that is inherent in the data. An example might be a column that can take on only two values.</li><li>Partition skew refers to skew that results from an uneven distribution of data across the AMPs.</li></ul>The difference is apparent from the context. As used in this manual, the term usually refers to the partition skew that occurs when the primary index for a table is defined on a column set that is highly nonunique.</td></tr>
<tr><td>Synoptic data structure</td><td>A data structure that contains summary, or synopsis, metadata.</td></tr>
</table>

## Content and Storage of Statistical Histograms

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

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

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

<table>
<tr><th>Position</th><th>Field Length (Bytes)</th><th>Information</th></tr>
<tr><td>9</td><td>40</td><td>Interval 0 contains global statistics for the column, multicolumn, or index. The following information is stored in each field of Interval 0:<br><br><table><tr><th>Field</th><th>Statistic</th></tr><tr><td>1</td><td>Minimum value for the column or index.</td></tr><tr><td>2</td><td>Modal value for the column or index.</td></tr><tr><td>3</td><td>Number of occurrences of the modal value for the column or index.</td></tr><tr><td>4</td><td>Number of distinct values for the column or index.</td></tr><tr><td>5</td><td>Cardinality of the column or index.</td></tr></table></td></tr>
<tr><td>10 - 109</td><td>40 per interval</td><td>Intervals 1 - 100 contain information specific to their respective histogram.<br><br><table><tr><th>Field</th><th>Statistic</th></tr><tr><td>1</td><td>Minimum value for the interval.</td></tr><tr><td>2</td><td>Modal value for the interval.</td></tr><tr><td>3</td><td>Number of occurrences of the modal value in the interval.</td></tr><tr><td>4</td><td>Number of distinct non-modal values in the interval.</td></tr><tr><td>5</td><td>Number of rows containing distinct non-modal values for the interval.</td></tr></table></td></tr>
</table>

Rows for each histogram are separated into three categories:

- Nulls
- Loners
- Non-loners

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

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

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

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

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

where:

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

Consider the following very simple examples:

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

## 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 Reference: Data Types and Literals* and *Database Design* for more information about large DECIMAL numbers.

See the MaxDecimal field of the DBSControl field in *Utilities* and *SQL Reference: Data Types and Literals* for more information about how to control the precision and storage size of DECIMAL values.

# All-AMPs 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 multi-billion 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, for which it is not possible to collect full-table statistics, you can opt to collect statistics on a sample of table rows instead.[20]

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

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

## Sampled Statistics Are More Accurate Than Random AMP Statistical Samples

Do not confuse statistical sampling with the random AMP samples that the Optimizer collects when it has no statistics on which to base a query plan. Statistical samples taken across all AMPs are likely to be much more accurate than random AMP samples.

20.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.

The following table describes some of the differences between the two methods of collecting statistics:

| Statistical Sampling | Random AMP Sampling |
|---|---|
| Collects statistics on a small sample of rows from all AMPs.[a] | Collects statistics on a small sample of rows from a single AMP.[b] |
| Collects full statistics (see "Content and Storage of Statistical Histograms" on page 69) and stores them in histograms in DBC.TVFields. | Collects estimates for cardinality and number of distinct index values only. |
| Expands sample size dynamically to compensate for skew. | Sensitive to skew. |
| Provides fairly accurate estimates of all statistical parameters. | Provides fairly accurate estimates of base table and NUSI cardinality if the following conditions are met:<br>• Table is large<br>• Distribution of values is not skewed<br>• Data is not taken from an atypical sample<br>Other standard statistical parameters are less likely to be as accurate. |

a. If the columns are not indexed, then the rows are organized randomly, so the system just scans the first n percent it finds, where the value of n is determined by the relative presence or absence of skew in the data. 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 value. It is possible to change the number of AMPs from which a random AMP sample is taken by changing the setting of an internal DBSControl field. The full range of possibilities is 1, 2, 5, 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 receiving an error message, but the specified value is not honored, and the system automatically changes the sampling percentage to 100 (see the documentation for COLLECT STATISTICS in *SQL Reference: Data Definition Statements* for details).

## A Note on Recollecting Statistics

When you recollect statistics on a column or index, whether for the Optimizer or for a QCD analysis, the same method of collection, either full-table scan or sampling, is used to recollect the data 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 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. No message is returned 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.[21]

See the documentation for COLLECT STATISTICS in *SQL Reference: Data Definition Statements* for additional information.

21.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 or index, then 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.

Note, however, that when it resorts to collecting random AMP samples, the system does not collect the same complete set of statistics that are collected when you collect complete statistics using a COLLECT STATISTICS statement (see *SQL Reference: Data Definition Statements* for information about the COLLECT STATISTICS statement). Because the Optimizer knows less about the predicates of a request when it only has sampled statistics, it always pursues more conservative query plan choices than it would for the same query if it had a complete set of statistics from the entire population of rows from all AMPs.

## Assumptions Underlying the Method

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

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

Because these assumptions are usually valid, the statistics estimated using dynamic AMP sampling are generally fairly accurate.[22]

### 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 internal field RandomAMPSampling in the DBSControl 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 field in DBSControl, RowsSampling, controls the percentage of rows to read in order to make an estimate of the Rows Per Value statistic. Because this is an internal DBSControl field, 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.

22. Sampled statistics do not work well with PPI tables, however, and should not be used to collect statistics for them.

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 78.

**End of Aside**

## How Cardinality Is Estimated From a Random Single AMP Sample

The process used to gather statistics using a random AMP sample is as follows:

1. Randomly select an AMP to be used to gather a statistical sample to represent an estimation of population demographics.

   To do this, the system hashes on the value of the table ID to produce 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:

$$\text{Estimated cardinality of table} = \frac{\text{Average number of rows in sampled cylinder}}{\text{Datablock}} \times \sum \text{Number of data blocks in sampled cylinder} \times \sum \text{Number of cylinders with data for the table on this AMP} \times \sum \text{Number of AMPs in the system}$$

8. End of process.

Table cardinality, the average cardinality per value, and, for indexes, the 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.[23]

23. 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.
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.

The system does *not* create interval histograms (see "Interval Histograms" on page 64) as a result of random AMP sampling. The only other parameters available to the Optimizer after it collects a random AMP statistical sample is a set of default heuristics used to estimate the selectivity of various predicates.[24]

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[25] 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 87 for a list of those statistics).

Because the Optimizer understands that the information it collects using a single random AMP sample is less reliable, it assumes Low confidence[26] 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.[27] 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.

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

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

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

25. 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.

26. 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.

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.

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.

27. Sequential AMP IDs are selected to obtain a more efficient load distribution.

# Relative Accuracy of Residual Statistics Versus Random AMP Sample Statistics

## Introduction

The relative accuracy of residual[28] statistics with respect to a random AMP sample[29] depends on several factors.

For example, if the demographics for a column or index do not change, then residual statistics are a good representation of current demographics. It is possible for the cardinality of a table to grow by more than 10 percent,[30] but for the relative proportion of its rows with particular values not to change. As a result, even a change in the table demographics of this magnitude might not affect the query plan generated by the Optimizer. When this is true, residual statistics could still be more accurate than a newly collected random AMP sample.[31]

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

## Some Comparison Scenario

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

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

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

29. In this section, the term "random AMP sample" refers to random single AMP sampling only.

30. 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.

31. The accuracy of the statistics collected from a random AMP sample also depends on the number of AMPs sampled, which is determined by the setting of an internal DBSControl field. 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.

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.

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

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

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

Tables with a large[32] number of distinct values might see no significant negative performance effects with residual statistics, with the exception of new column values such as new dates. Tables with several orders of magnitude fewer distinct values can have a larger number of values nearer the crossover point that would affect their position within the statistics partitions if new statistics were collected.

## Effects of Cardinality on the Usefulness of Residual Statistics

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

Suppose a table has only 100 000 rows with 1 000 distinct values, an average of 100 rows per distinct value.

32. Large in this case is defined as greater than 100 000 distinct values.

The following list illustrates some of the potential issues the Optimizer faces when dealing with small tables, particularly with respect to generating query plans based on inaccurate statistics:

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

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

# Comparing the Relative Accuracies of Various Methods of Collecting Statistics

## Introduction

The important thing to understand when considering the comparative *a priori* likelihoods of the accuracy of statistics collected by the various available methods is the consistently higher accuracy of population statistics over all forms of sampled statistics.[33] Although this is undeniable, it is not possible to know *a priori* whether it is necessary to collect a full set of new statistics in order to ensure that the Optimizer produces the best query plans.

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

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

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

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

| Collection Method | Relative Elapsed Time to Collect | Accuracy Rank (Higher Number = Higher Accuracy) | Comments |
|---|---|---|---|
| Use residual statistics | None. | 1 | • Impossible to know *a priori* if residual statistics will produce a good query plan.<br>At worst, can produce a very poor query plan.<br>• Because statistics exist, Optimizer does not collect fresh statistics using a random AMP sample. |

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

| Collection Method | Relative Elapsed Time to Collect | Accuracy Rank (Higher Number = Higher Accuracy) | Comments |
|---|---|---|---|
| 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.<br>• 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.<br>• Because the sample size is small, there is a high bias in the statistics.<br>• Accuracy is very sensitive to skew.<br>• 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.<br>Other standard statistical parameters are less likely to be as accurate.<br>• Does not enable semantic query optimization. |
| All-AMPs sample | Approximately 5 percent of time to perform a full-table scan[a]. | 3 | • Data is collected from a system-determined subset of the rows on all AMPs.<br>• Collects identical statistics to full-table scan, including interval histogram creation.<br>• Percentage of sampled rows is small.<br>• Percentage of sampled rows is increased dynamically to enhance the accuracy of collected statistics if skew is detected in the samples.<br>• Does not enable semantic query optimization. |
| Full-table scan | Approximately 195 percent of the time to perform sampled statistics. | 4 | • Data is collected from all rows on all AMPs so there is no sample bias.<br>• Collects full statistics and creates interval histograms.<br>• Skew is accounted for using high-biased intervals in the statistical histogram for a column or index.<br>• Up to 100 histogram intervals are used, depending on the data.<br>• Required to initiate rule production for semantic query optimization. |

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

b. This is the default. Depending on an internal DBSControl 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 Sample Statistics" on page 80 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* for additional information about the uses and capabilities of this utility.

## Policies for Collecting Statistics

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

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

The following table provides more specific recommendations for collecting statistics. You should consider this set of recommendations to be both of the following:

- Minimal
- Essential

| FOR this category of table ... | You should collect statistics on ... |
|---|---|
| all | all columns used in join conditions. |
| large | all NUPIs. |
| small[a] | the primary index. |

a. In this context, a small table is defined as a table whose cardinality is less than 5 times the number of AMPs in the system. 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.
- Columns referenced in WHERE clause predicates, particularly if those columns contain skewed data.

# How the Optimizer Uses Statistical Profiles

## Introduction

The COLLECT STATISTICS (Optimizer Form) statement (see *SQL Reference: Data Definition Statements* 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 Reference: Data Definition Statements* 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 The Teradata Database" on page 64).

Different statistics are stored for a column depending on whether its values are highly skewed or not, as indicated by the following table:

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

Note the use of the term *estimate* in the 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.

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td colspan="2">Statistics Maintained for All Interval Types</td></tr>
<tr><td>Date collected</td><td>Reported by HELP STATISTICS as Date.<br>The date on which statistics were last collected.</td></tr>
<tr><td>Time collected</td><td>Reported by HELP STATISTICS as Time.<br>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.<br>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.<br>An estimate of the number of nulls for the specified column or index.</td></tr>
<tr><td>Number of intervals</td><td>Reported by HELP STATISTICS as Number of Intervals.<br>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.<br>Identifies whether the data type of the column set reported on is numeric or nonnumeric.<br><table><tr><th>Code</th><th>Definition</th></tr><tr><td>0</td><td>The data type is nonnumeric.</td></tr><tr><td>nonzero</td><td>The data type is numeric.</td></tr></table></td></tr>
<tr><td>Sampled flag</td><td>Reported by HELP STATISTICS as Sampled.<br>Identifies whether the statistics were collected from all rows in the table or from a sampled subset.<br><table><tr><th>Code</th><th>Definition</th></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></table></td></tr>
<tr><td>Sampled percent</td><td>Reported by HELP STATISTICS as Sampled Percent.<br>The approximate percentage of total rows in the table included in the sample.</td></tr>
<tr><td>Version number</td><td>Reported by HELP STATISTICS as Version.<br>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.<br>An estimate of the number of unique values for the column.</td></tr>
</table>

| Attribute | Description |
|---|---|
| Minimum value for the interval | Reported by HELP STATISTICS as Min Value.<br>An estimate of the smallest value for the specified column or index in the interval. |
| Maximum number of rows per value | Not reported by HELP STATISTICS.<br>An estimate of the maximum number of rows having the particular value for the column. |
| Typical number of rows per value | Not reported by HELP STATISTICS.<br>An estimate of the most common number of rows having the particular value for the column. |
| **Equal-Height Interval Statistics** | |
| Maximum value for the interval | Reported by HELP STATISTICS as Max Value.<br>An estimate of the largest value for the column or index in the interval. |
| Modal value for the interval | Reported by HELP STATISTICS as Mode Value.<br>An estimate of the most frequently occurring value or values for the column or index in the interval. |
| Number of rows having the modal value | Reported by HELP STATISTICS as Mode Frequency.<br>An estimate of the distinct number of rows in the interval having its modal value for the column or index. |
| Number of non-modal values | Reported by HELP STATISTICS as Non-Modal Values.<br>An estimate of the number of distinct non-modal values for the column or index in the interval. |
| Number of rows not having the modal value | Reported by HELP STATISTICS as Non-Modal Rows.<br>An estimate of the skewness of the distribution of the index or column values within the interval. |
| **High-Biased Interval Statistics** | |
| Maximum value for the interval | Reported by HELP STATISTICS as Max Value.<br>If two values are stored in the interval, then Max is the value for the second loner in a two-loner interval. |
| Modal value for the interval | Reported by HELP STATISTICS as Mode Value.<br>If two values are stored in the interval, then Mode is the value for the first loner.<br>No more than two modal values can be stored per interval for a skewed distribution. |
| Number of rows having the modal value | Reported by HELP STATISTICS as Mode Frequency.<br>The number of rows in the interval having the modal value. |

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>Number of non-modal values</td><td>Reported by HELP STATISTICS as Non-Modal Values.<br>A code indicating the number of loner values for the interval.<br><table><tr><th>Code</th><th>Definition</th></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></table></td></tr>
<tr><td>Number of rows not having the modal value</td><td>Reported by HELP STATISTICS as Non-Modal Rows.<br>The number of rows in the interval having the maximum value.</td></tr>
</table>

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

## Introduction

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

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

## Interval Histogram Data

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

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

  This number is constant across intervals when equal-height intervals are used.

  Its value is calculated as follows:

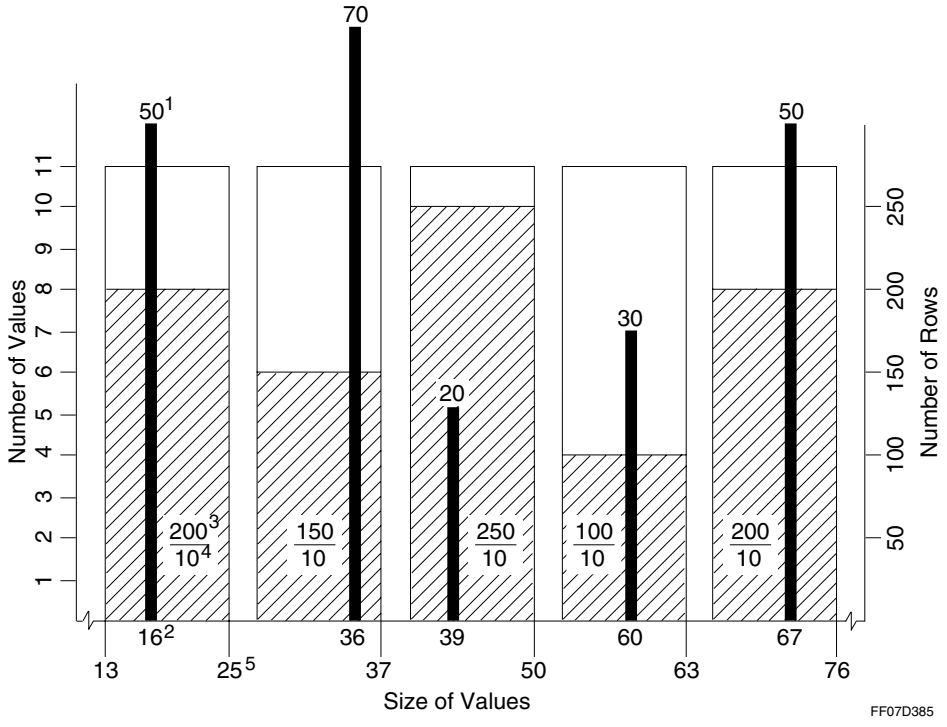
  ```
  Values not equal to most frequent  =  Number of values in interval - 1
  ```

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

The data used for the examples is tabulated 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 for equal-height interval histograms.

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

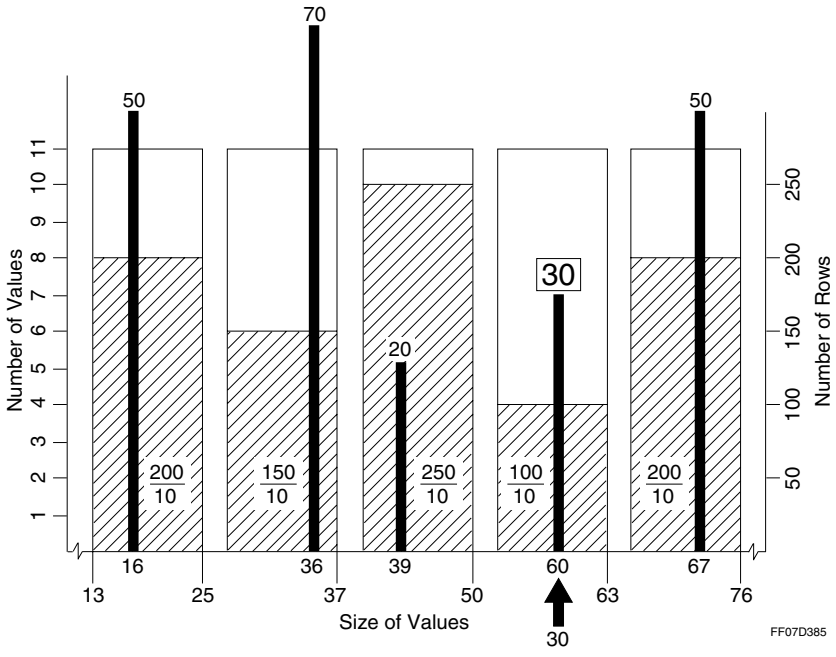
The following picture illustrates these numbers graphically. The footnotes are defined in the table following the illustration.



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

## 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*.



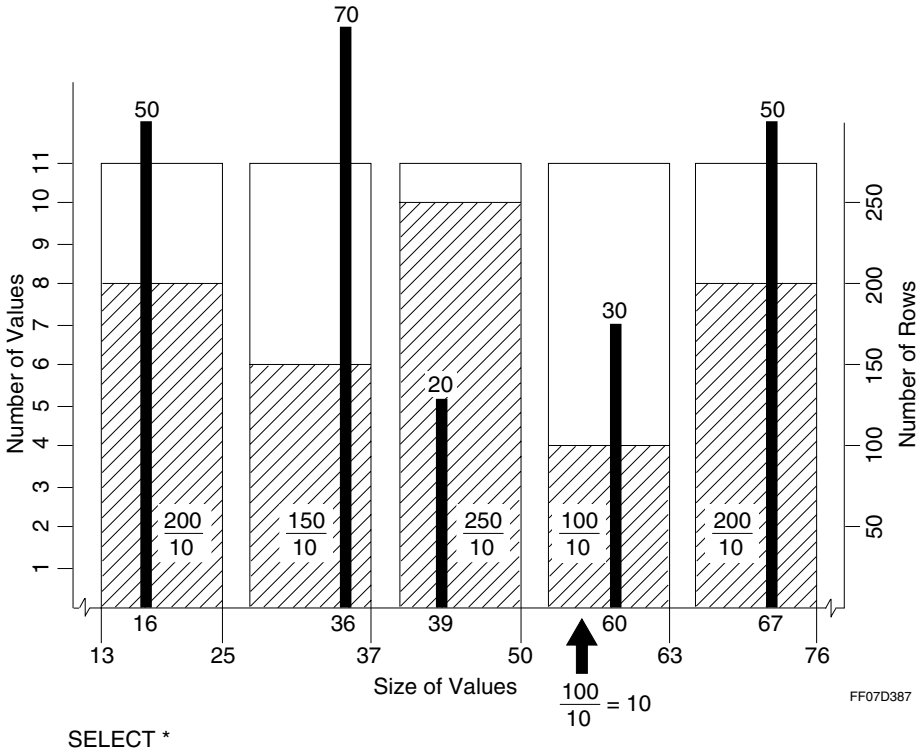

```
SELECT *
FROM table
WHERE condition = 60;

[ 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 93, there are no exact statistics to describe it. The heuristic used to estimate the response set cardinality is to divide the number of rows in the interval not having the most frequent value by the number of values in the interval not equal to the most frequent value.

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

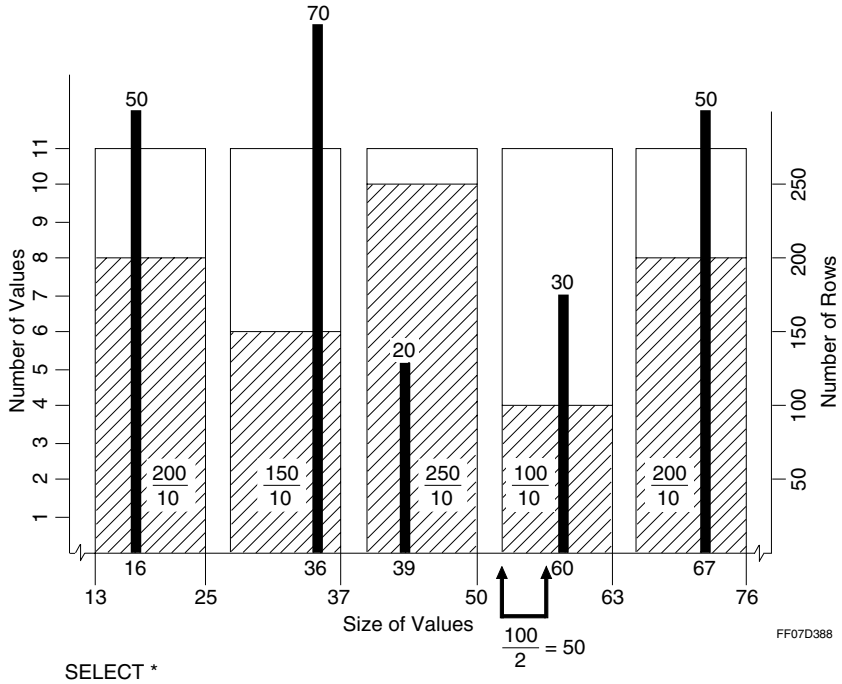
If the statistics are current, then there are 100 rows in this interval that do not have the value 30 for the column specified by *condition* and there are 10 values that are not equal to the most frequent value in the interval. The Optimizer divides the number of rows in the interval that do not have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

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

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

## Example 3

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




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

[ 50 rows ]
```

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

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

The Optimizer knows the following things about this condition:

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

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

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

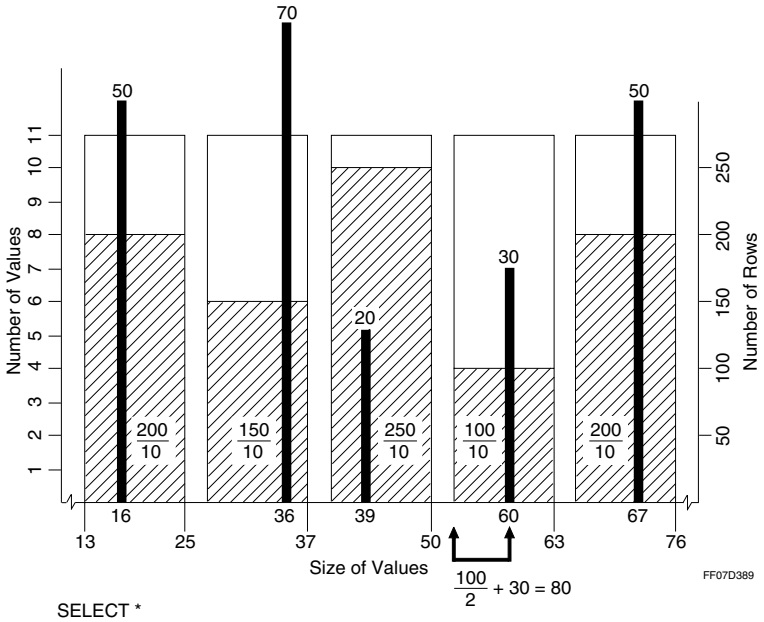
Assuming current statistics, there are 100 rows in the interval that do not have the value 30 for the column specified by *condition*, so the Optimizer divides the number of rows not having the value 60, which is 100, in half.

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

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

## Example 4

This example is slightly more sophisticated than "Example 3" on page 95 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.

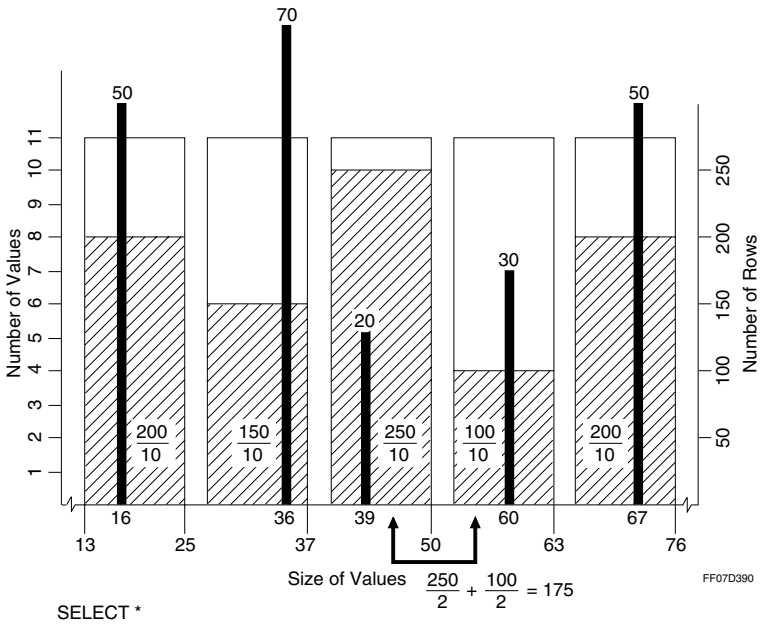
Because this quantity is known exactly if the statistics are current, the estimated cardinality of the response set should be more accurate than in the previous example.

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 96 because the response set spans two histogram intervals.




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

[ 175 rows ]
```

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

The Optimizer knows the following things about this condition:

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

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

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

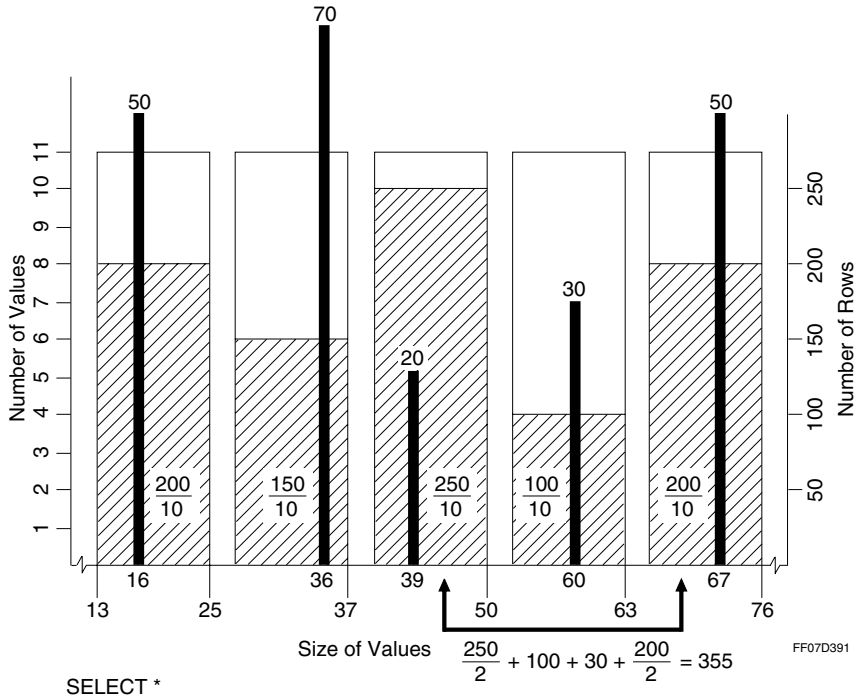
There are 100 rows in the higher interval that do not have a value 30 for the column specified by *condition*, so the Optimizer divides the number of rows not having the value 60, which is 100, in half, producing an estimate of 50 rows satisfying the condition for that interval.

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

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

## Example 6

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




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

[ 355 rows ]
```

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

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

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

The Optimizer knows the following things about this condition:

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

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

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.

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

# Cost Optimization

## Introduction

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

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

The column and index demographics gathered by the COLLECT STATISTICS (Optimizer Form) statement (see *SQL Reference: Data Definition Statements* for syntax and usage information) and the statistics it computes permit the Optimizer to better identify skew in tables, so it has more finely-tuned knowledge for generating plans using its cost estimator functions.

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

## Example

Consider the following table fragment:

**Table_X**

| State | SerialNum |
|---|---|
| 28 | 12345 |
| 28 | 23456 |
| 51 | 12345 |
| 51 | 23456 |

The following query resulted in a full file scan:

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

Now modify the query as follows:

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

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

## Path Selection

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

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

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

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

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

## Example 1

Consider the following expression:

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

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

## Example 2

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

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

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

## Example 3

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

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

Internally, that expression is converted to CNF.

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

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

## Example 4

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

## 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 non-default cost profile, but only Teradata field engineers have the access privileges required to use them.

## Statistics And Cost Estimation

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

- The Optimizer evaluates access selectivity based on demographic information.

  If statistics have recently been collected on the primary index for a table, then an accurate estimate of the total number of rows is available to the query Optimizer.
- The Optimizer uses primary index statistics to estimate the 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, 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 76). 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.

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

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

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

## Ensuring Indexed Access

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

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

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

## Guidelines For Indexed Access

*Always* use the EXPLAIN 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:

- UPIs are used for fastest access to table data.
- USIs are used only to process requests that employ equality constraints.
- The best performance in joins is achieved when the following is possible:
  - Matching UPI values in one table with unique index values, either UPI or USI, in another table.
  - Using *only* a primary index when satisfying an equality or IN condition for a join.
- NUPIs are used for single-AMP row selection or join processing to avoid sorting and redistribution of rows.
- When statistics are not available for a table, the estimated cost of using an index is based on information from a single AMP. This estimate assumes an even distribution of index values. An uneven distribution impacts performance negatively.
- Composite indexes are used only to process requests that employ equality constraints for *all* fields that comprise the index.

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

  For example, consider the following condition:

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

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

# Environmental Cost Factors

## Introduction

The 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.

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

## External Cost Parameters

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

| External Cost Parameter Group | Definition |
|---|---|
| Optimizer weights and scales | Weightings of CPU, disk, and network contributions to optimizing a request plus scaling factors to normalize disk array and instruction path length contributions.<br>Unitless decimal weighting factors. |
| CPU cost variables | CPU costs for accessing, sorting, or building rows using various methods. |
| Disk delay definitions | Elapsed times for various disk I/O operations.<br>Measured in milliseconds. |
| Disk array throughput | Throughputs for disk array I/O operations.<br>Measured in I/Os per second. |
| Network cost variables | Message distribution and duplication costs and overheads.<br>Measured in milliseconds. |

| External Cost Parameter Group | Definition |
|---|---|
| Optimizer environment variables | Hardware configuration information such as the number of CPUs and vprocs per node, maxima, minima, and average numbers of MIPS per CPU, nodes per clique, disk arrays per clique, and so on.<br>Unitless decimal values. |
| Miscellaneous cost parameters | DBSControl information such as free space percentages, maximum number of parse tree segments, dictionary cache size, and so on.<br>Measured in various units, depending on the individual cost parameter. |
| Performance constants | Various values used to assist the Optimizer to determine the best method of processing a given query.<br>Measured in various units, depending on the individual performance constant. |
| PDE system control files | Lists various system control files used by PDE. Used to initialize the appropriate GDOs with the appropriate values at startup. |
| DBS control files | Lists various database control files found in the DBSControl record. Used to initialize the appropriate GDOs with their respective appropriate values at startup. |
| TPA subsystem startup initializations | Initializes or recalculates various Optimizer parameters, including the optimizer target table GDO used by the target level emulation software (see Chapter 7: “Target Level Emulation”). |

# Optimizer Join Plans

## Introduction

Selecting the optimum method to join tables is as critical to the performance of a query as the selection of table access methods. Selecting an optimal join method, or mode, is a separate issue from that of determining the optimal join order for a query. Join order optimization is described in "Evaluating Join Orders" on page 135.

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

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

## 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 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[34] are joined can have a powerful impact on join cost.

The following table outlines some of the new terminology introduced by join planning:

| Term Type | Definition | Example |
|---|---|---|
| Bind | An equality constraint between two columns of different relations.<br>Normally used to determine whether there are conditions on an index. | `(t1.a = t2.a)` |

34. 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.

| Term Type | Definition | Example |
|---|---|---|
| Cross | A predicate with expressions on different relations.<br>A cross term can be generated for a condition of the form `column_name=expression`, which is referred to as a half cross term.<br>The Optimizer can use a half cross term as a constraint if it is specified on an index.<br>The form `conversion(column)=expression` can be used for the same purpose if `conversion(column)` and `column` are hashed the same. | `(t1.a = t2.a + 1)` |
| Exists | A predicate that specifies an EXISTS condition. | |
| Explicit | A predicate defined on a constant.<br>Normally used to determine whether there are conditions on an index. | `(t1.a = 1)` |
| Minus | A predicate that specifies a MINUS condition. | |
| Miscellaneous | Literally a group of terms that do not belong to any of the other categories.<br>The set of miscellaneous terms includes the following:<br>• Inequality terms.<br>• Equality terms on either the same set of tables or on non-disjoint sets of tables.<br>• ANDed terms.<br>• ORed terms.<br>• Other terms (for example, terms expressed over more than 2 relations).<br>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.<br>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) IN (SELECT t2.a, t2.b FROM t2)
→(t1.x IN Spool1.a AND (t1.y IN Spool1.b)

(t1.x, t1.y) IN (SELECT t2.a, constant FROM t2)
→(t1.x IN spool1.a) AND (t1.y=spool1.constant)
```

```
(t1.x, constant) NOT IN (SELECT t2.a, t2b FROM t2)
→ (t1.x 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 tables in a join operation:

- There is a direct connection between two tables if either of the following conditions is found:
  - An ANDed bind, misc, cross, outer join, or minus term that satisfies the dependent info between the two tables.
  - A spool file of a non-correlated subquery EXIST condition connects with any outer table.
- An ANDed miscellaneous or explicit term on a single table is pushed to the table.
- A term on no table is pushed to a table.
- An ORed term that references some subqueries and a single table is associated with that table as a complex term.
- All tables 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:

  ```
  SELECT t1.x1
  FROM t1, t2
  WHERE y1=1
  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.
  - 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 tables, for example:

    ```
    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 table row size.
  - The table 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 Chapter 3: "Join Optimizations," for more information about each of these join methods.

## Limit on the Number of Tables That Can Be Joined

Excluding self-joins, as many as 64 tables or views[35] can be joined per query block.

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

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

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

35. This assumes that the view is on a single table.

For example, consider a query with a single uncorrelated subquery. The subquery is limited to 64 tables and the outer query is limited to *63* tables, the 64$^{th}$ 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 64 tables, its outer query limited to 63 (63 plus the result of its inner query), and the parent outer query to 63 plus the result of *its* inner query.

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

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

## Join 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 Reference: Functions and Operators* 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.

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 Reference: UDF, UDM, and External Stored Procedure Programming* and *SQL Reference: Data Definition Statements* 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 tables in a request, use the DBS Control Utility (see *Utilities*) to increase the value for the DBSControl Record MaxParseTreeSegs performance field 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.

| FOR systems with this address size ... | THE minimum value for MaxParseTreeSegs is ... | THE default value for MaxParseTreeSegs is ... | AND the maximum value for MaxParseTreeSegs is ... |
|---|---|---|---|
| 32-bit | 12 | 1 000 | 3 000 |
| 64-bit | 24 | 2 000 | 6 000 |

The MaxParseTreeSegs field defines the number of 64 Kbyte parse tree memory segments the Parser allocates when parsing an SQL request.

## 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 144) to determine the least expensive method of joining two tables.

Product joins (see "Product Join" on page 145)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 151). For equality joins only, a hash join (see "Hash Join" on page 160) can be the least costly join method, particularly if the smaller table 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 167).

As an example, consider an equality join. In this case, the smaller table is duplicated on all AMPs and then every row in the smaller table is compared with every row in the larger table. The total number of comparisons to do the join is the product of the number of rows in the smaller table and the number of rows in the larger table.

An optimal solution for reducing the number of comparisons made by a Product Join is to assign the smaller table rows to a hash array and then use a Hash Join instead of a Product Join. In this case, each row in the right table does a simple table look up in the hash array to see if an entry exists or not.

| IF there is ... | THEN the system ... |
|---|---|
| no match in the hash array | discards the row and makes no further comparisons. |
| a match in the hash array | continues processing to determine if the remaining join conditions match. |

No other entries in the hash array need to be checked.

For an equality Product Join, if the estimated cardinality of the small table is 5 rows and the actual cardinality is 500 rows, then each row in the right table 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 table 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 table and generates the right table hash codes dynamically (see "Dynamic Hash Join" on page 167).

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 table in a join has 100 rows evenly distributed across the AMPs.

Consider the following three join options for these tables. The options are numbered for reference purposes, not for precedence:

1. The Optimizer can generate a join plan to redistribute the larger table and do a Product Join with approximately one row per AMP.

   The join can be on the primary index of the small table, which means redistributing the small table is unnecessary, or the join could be on a non-primary index set of columns. In the latter case, the small table will also have to be redistributed. After redistributing the larger table, only one comparison per row, on average, is needed because there is only one row per AMP in the small table.
2. The Optimizer can decide to duplicate the smaller table and then do a product join with 100 rows so that every row in the large table requires 100 comparisons.

   The usefulness of this option depends on the size of the larger table.
3. The Optimizer can decide to duplicate the smaller table and then sort both the large and small tables followed by a merge join.

Option 1 is faster than Option 2. In Option 1, rows from the large table 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 table needs to be sorted. However, by replacing the equality Product Join with a Hash Join, the smaller table can be duplicated instead of redistributing the larger table. The comparison costs are approximately the same because only one comparison per row is required in both cases. The larger table 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 135 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.

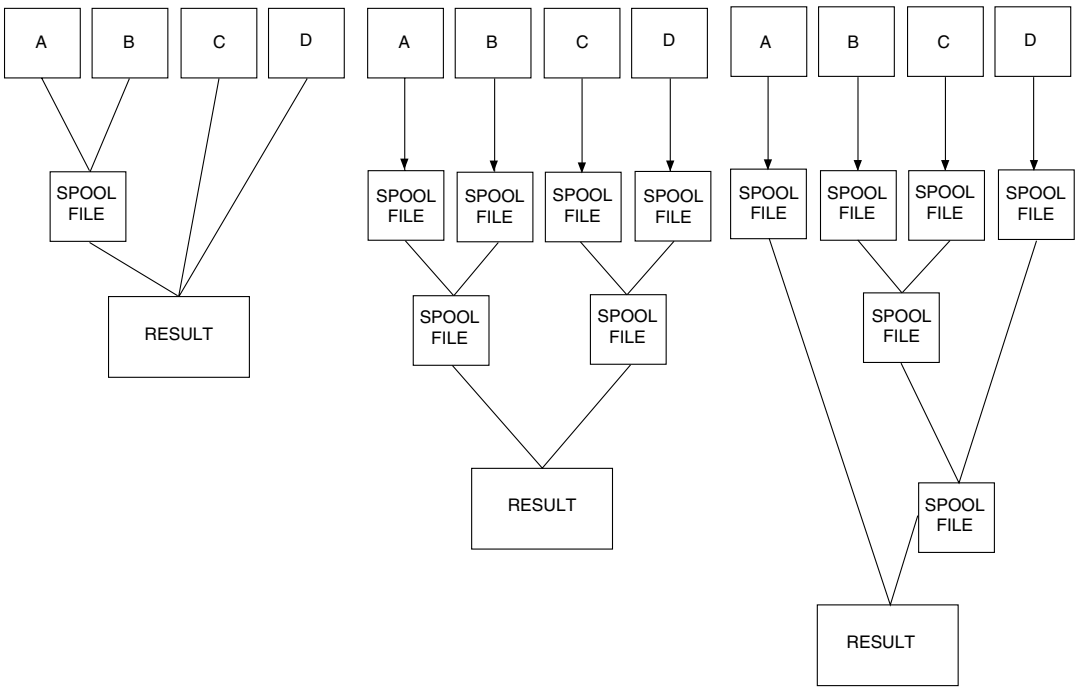
## How the Optimizer Plans Multitable Joins

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

For example, assume you submitted the following multitable query:

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

The Optimizer generates join plans like the three diagrammed in the following graphic.[36] Because the Optimizer uses column statistics to choose the least costly join plan from the candidate set it generates, the plans it generates are extremely dependent on the accuracy of those numbers.




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

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

36. 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.

## Join Order Search Trees

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

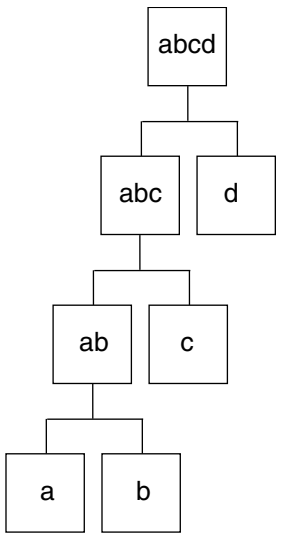
## Left-Deep Search Trees

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

```
Number of join orders = n!
```

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

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




ff07D413

## Bushy Search Trees

Bushy trees are an optimal method for generating more join order combinations. At the same time, the number of combinations generated can be prohibitive (see "Possible Join Orders as a Function of the Number of Tables To Be Joined" on page 119), 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:

| Equation element ... | Represents the ... |
|---|---|
| *O* | Landau complexity symbol of number theory and computation theory, meaning "order of." |
| *n* | number of relations in the join. |

Also see "Possible Join Orders as a Function of the Number of Tables To Be Joined" on page 119.

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* tables based on a bushy join tree search:

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

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:

| This variable ... | Represents this term ... |
|---|---|
| *n* | `2(n-1)` |
| *k* | `(n-1)` |

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

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

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

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

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

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

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

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

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

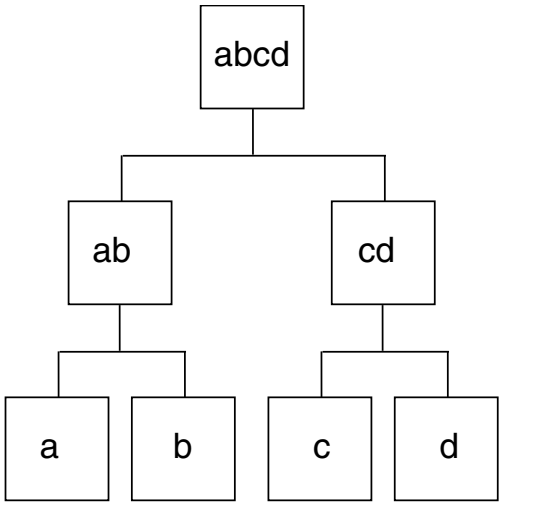
and so on.

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

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

The illustrated process can be described as follows:

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

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

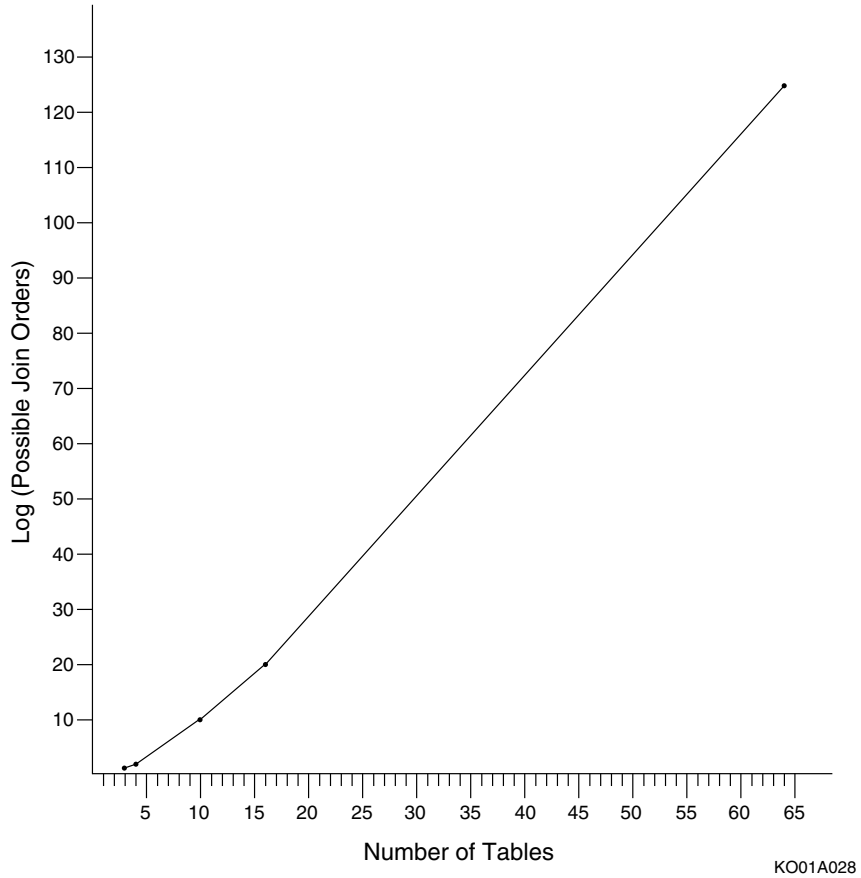
The following table illustrates the dilemma the Optimizer faces when it selects the order in which to join tables in a join plan. The table demonstrates how rapidly the possibilities for ordering binary joins escalates as the total number of tables joined increases.

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

| Number of Tables Joined | Number of Possible Join Orders | log(Possible Join Orders) | Number of Grains of Sand |
|---|---|---|---|
| 3 | 12.0 | 1.079 | 12 |
| 4 | 120.0 | 2.079 | 120 |
| 10 | $1.8*10^{10}$ | 10.255 | 6 sand boxes |
| 16 | $2.0*10^{20}$ | 20.301 | All the sand grains of all the beaches and all the deserts in the world |
| 64 | $1.2*10^{124}$ | 124.079 | All the sand grains required to fill the known universe |

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

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



## Reduce the Number of Participating Rows With Careful Query Coding

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

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

| Key | |
|---|---|
| **Abbreviation** | **Meaning** |
| PK | Primary key |
| FK | Foreign key |
| UPI | Unique primary index |
| SA | System-assigned |

| **Statistics** | **Shipments** | |
|---|---|---|
| 500 rows | **ShipNum** | ... |
| PK/FK | **PK, SA** | |
| | **UPI** | |
| Dist values | 500 | |
| Max rows/val | 1 | |
| Typ rows/val | 1 | |

| **Statistics** | **Parts** | | |
|---|---|---|---|
| 4 000 000 rows | **PartNum** | **ShipNum** | ... |
| PK/FK | **PK, SA** | | |
| | **UPI** | | |
| Dist values | $4.0*10^6$ | 501 | |
| Max rows/val | 1 | $3.99 \times 10^6$ | |
| Typ rows/val | 1 | 20 | |

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

```
SELECT ...
FROM Shipment, Part
WHERE Shipment.ShipNum = Part.ShipNum;
```

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

```
SELECT ...
FROM Shipment, Part
WHERE Part.ShipNum IS NOT NULL
AND Shipment.ShipNum = Part.ShipNum;
```

# Join Geography

## Introduction

Join geography is a term that describes the relocation, if any, of table 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, 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 | Cardinality When Statistics Last Collected | Cardinality Now |
|---|---|---|
| A | 1 000 | 50 000 |
| B | 75 000 | 3 500 |

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 pointedly 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 123.
- "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 126.
- "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 128.
- "Example 4: Duplicate the Smaller Table On Every AMP For a Product Join" on page 129.

Notice that the primary index is the major consideration used by the Optimizer in determining how to join two tables 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 133).

The query that generates a Merge 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:

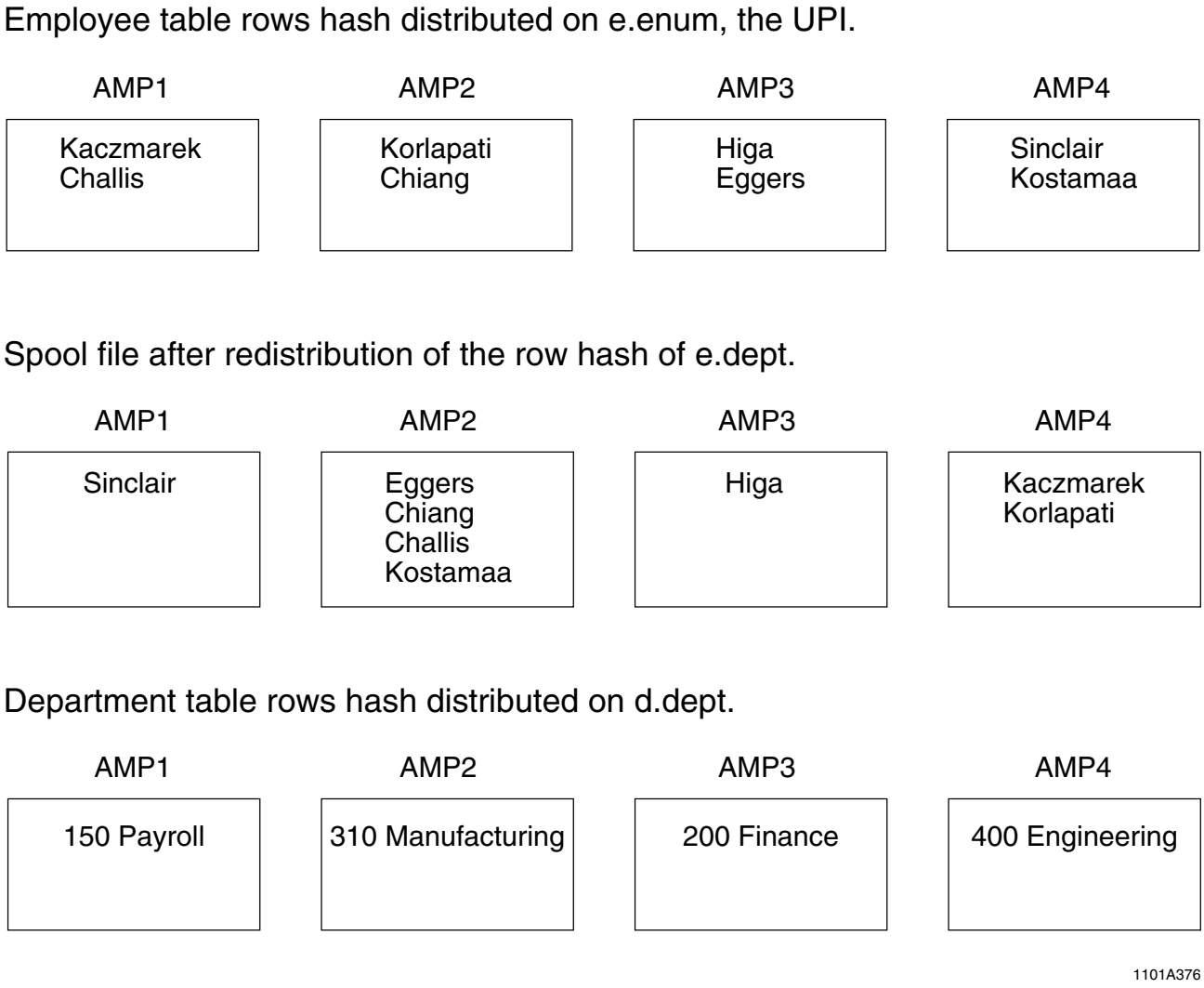
**employee**

| enum | name | dept |
|---|---|---|
| **PK** | | **FK** |
| **UPI** | | |
| 1 | Higa | 200 |
| 2 | Kostamaa | 310 |
| 3 | Chiang | 310 |
| 4 | Korlapati | 400 |
| 5 | Sinclair | 150 |
| 6 | Kaczmarek | 400 |
| 7 | Eggers | 310 |
| 8 | Challis | 310 |

**department**

| dept | name |
|---|---|
| **PK** | |
| **UPI** | |
| 150 | Payroll |
| 200 | Finance |
| 310 | Manufacturing |
| 400 | Engineering |

The following graphic shows how individual rows would be relocated to make this join for a four AMP system:




The example is run on a four AMP System. The query uses an equijoin condition on the dept columns of both tables. The system copies the employee table rows into spool and redistributes them on the row hash of 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 (Department), 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 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 133).

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 123, 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 will choose the strategy followed by this scenario.

The query that generates a Merge 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:

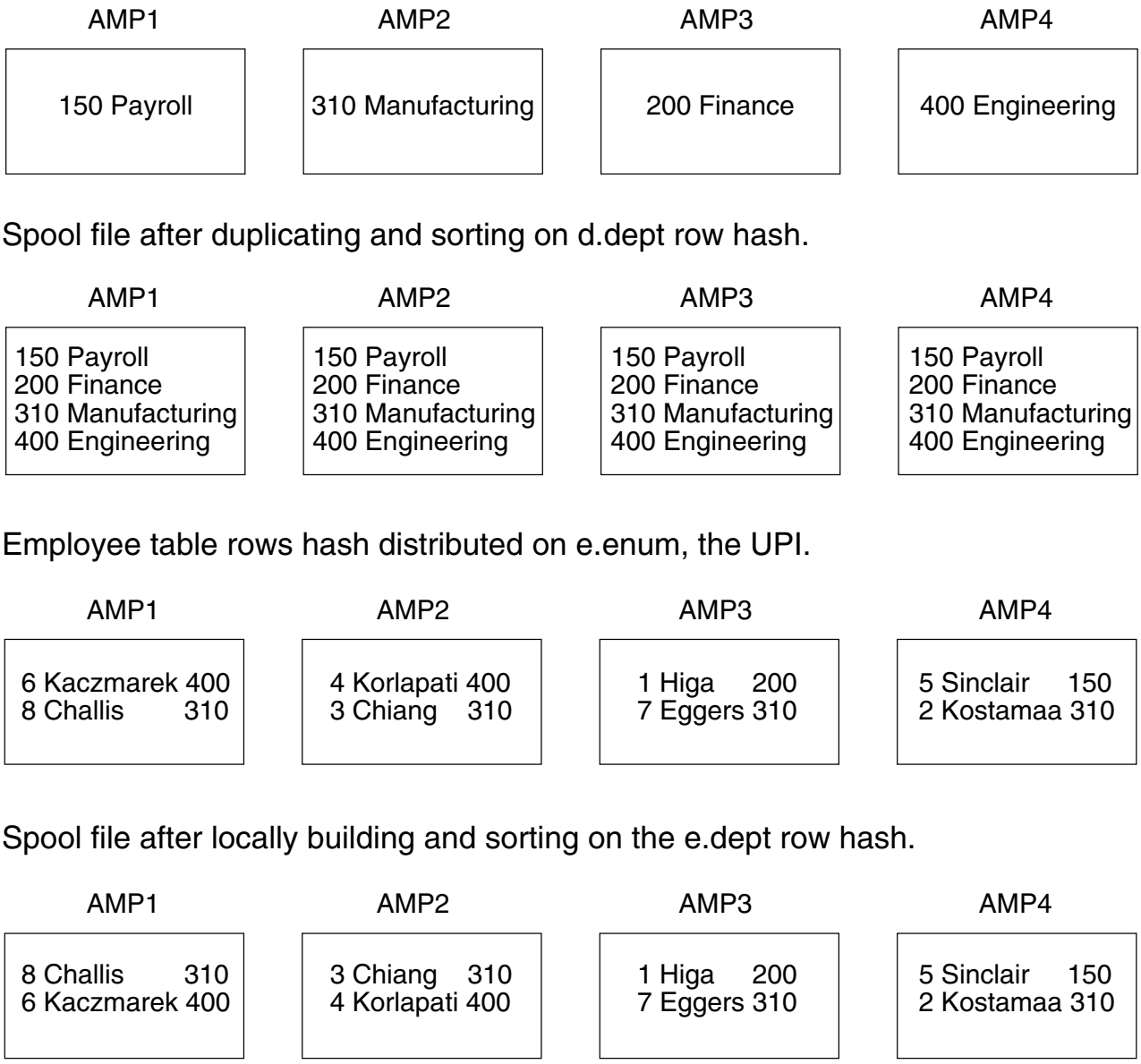
**employee**

| enum | name | dept |
|---|---|---|
| PK | | FK |
| UPI | | |
| 1 | Higa | 200 |
| 2 | Kostamaa | 310 |
| 3 | Chiang | 310 |
| 4 | Korlapati | 400 |
| 5 | Sinclair | 150 |
| 6 | Kaczmarek | 400 |
| 7 | Eggers | 310 |
| 8 | Challis | 310 |

**department**

| dept | name |
|---|---|
| PK | |
| UPI | |
| 150 | Payroll |
| 200 | Finance |
| 310 | Manufacturing |
| 400 | Engineering |

The following graphic shows how individual rows would be relocated to make this join for a four AMP system:




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 132). 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:

```
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:

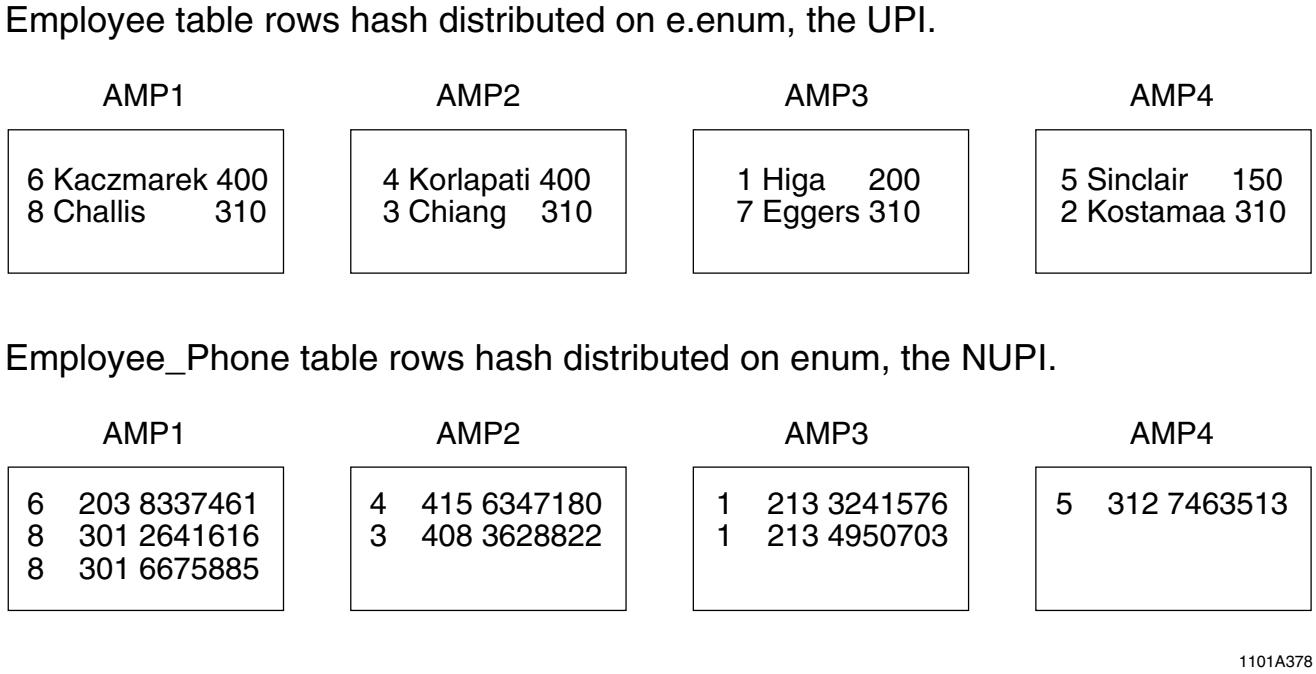
**employee**

| enum | name | dept |
|---|---|---|
| PK | | FK |
| UPI | | |
| 1 | Higa | 200 |
| 2 | Kostamaa | 310 |
| 3 | Chiang | 310 |
| 4 | Korlapati | 400 |
| 5 | Sinclair | 150 |
| 6 | Kaczmarek | 400 |
| 7 | Eggers | 310 |
| 8 | Challis | 310 |

**employee_phone**

| enum | area_code | phone |
|---|---|---|
| PK | | |
| FK | | |
| NUPI | | |
| 1 | 213 | 3241576 |
| 1 | 213 | 4950703 |
| 3 | 408 | 3628822 |
| 4 | 415 | 6347180 |
| 5 | 312 | 7463513 |
| 6 | 203 | 8337461 |
| 8 | 301 | 6675885 |
| 8 | 301 | 2641616 |

The following graphic shows how individual rows would be relocated to make this join for a four AMP system:




## 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 four 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 133, 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 θ join, where θ 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:

**employee**

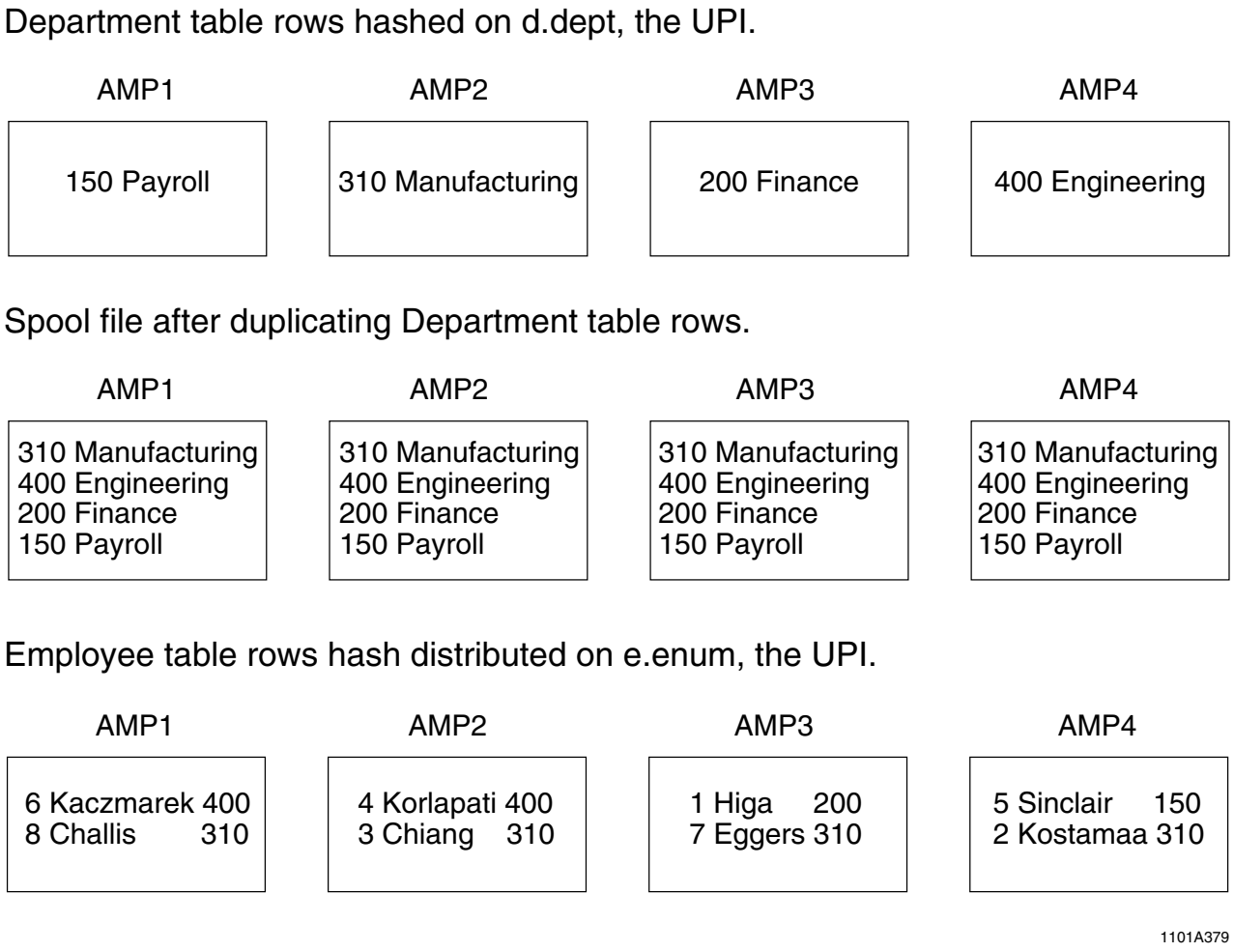
| enum | name | dept |
|---|---|---|
| PK | | FK |
| UPI | | |
| 1 | Higa | 200 |
| 2 | Kostamaa | 310 |
| 3 | Chiang | 310 |
| 4 | Korlapati | 400 |
| 5 | Sinclair | 150 |
| 6 | Kaczmarek | 400 |
| 7 | Eggers | 310 |
| 8 | Challis | 310 |

**department**

| dept | name |
|---|---|
| PK | |
| UPI | |
| 150 | Payroll |
| 200 | Finance |
| 310 | Manufacturing |
| 400 | Engineering |

The Product Join is caused by the non-equijoin condition `e.dept > d.dept` (see "Product Join" on page 145). 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 four AMP system:




## 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.

For example, three general scenarios can occur when two relations are joined using the Merge Join method (see "Merge Join" on page 151), 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:

| Rank | Scenario | WHEN the join column set is ... | FOR these relations in the join operation ... |
|---|---|---|---|
| 1 | 1 | the primary index | both. |
| 2 | 2 | the primary index | one. |
| 3 | 3 | not the primary index | neither. |

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:

```
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_1**

| col_1 | col_2 | col_3 |
|---|---|---|
| PI | | |
| 100 | 214 | 433 |

**table_2**

| col_1 | col_2 | col_3 |
|---|---|---|
| PI | | |
| 100 | 725 | 002 |

## 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,[37] 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 non-PI table.

For example, consider the following query:

```
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 non-primary 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.[38] 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_3**

| col_1 | col_2 | col_3 |
|---|---|---|
| PI | | |
| 255 | 345 | 225 |

**table_4**

| col_1 | col_2 | col_3 |
|---|---|---|
| PI | | |
| 867 | 255 | 566 |

**spool**

| col_1 | col_2 | col_3 |
|---|---|---|
| | PI | |
| 867 | 255 | 566 |

37. 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.

38. There is no rule that forces the rows of the non-primary 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:

```
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 non-primary index col_2 of table_5 and on the non-primary index col_3 of table_4, 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_5**

| col_1 | col_2 | col_3 |
|---|---|---|
| PI | | |
| 456 | 777 | 876 |

**table_6**

| col_1 | col_2 | col_3 |
|---|---|---|
| PI | | |
| 993 | 228 | 777 |

**spool**

| col_1 | col_2 | col_3 |
|---|---|---|
| PI | | |
| 456 | 777 | 876 |

**spool**

| col_1 | col_2 | col_3 |
|---|---|---|
| PI | | |
| 993 | 228 | 777 |

# Evaluating Join Orders

## Introduction

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

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.

| IF the following conditions are found ... | THEN ... |
|---|---|
| • The cost of the one-join lookahead plan exceeds a threshold value.<br>• There exists one table larger than the threshold.<br>• The query has no outer join requirement. | Generate a five-join lookahead plan and evaluate its cost. |

2 Keep the less costly plan generated.

3 Evaluate the new existing join conditions.

| IF the following conditions are found ... | THEN ... |
|---|---|
| • The cost of the current plan exceeds some threshold.<br>• A star join might be required. | Generate a better plan that uses star join optimization if such a plan can be found.<br>Otherwise, go to Stage 4. |

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 tables.
- Join a nonconnected table only after all connected tables 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.

| IF a unique ID is ... | THEN do the following ... |
|---|---|
| not used | 1 Build the assignment list for the temporary relation.<br>2 Use it to estimate the following things:<br>• Spool size<br>• Output row cost<br>3 Map the term information of the residual term list to refer to the new relation.<br>When this temporary relation is removed, the assignment list is reused and the term list is remapped to refer to the inputs. |
| used | Use existing map list information to estimate the following things:<br>• Spool size<br>• Output row cost |

6 Evaluate the best current join plan.

| IF ... | THEN ... |
|---|---|
| the following conditions are found for the first two joins of the best join plan:<br>• The joins involve one table connected with two tables not connected to each other<br>• Neither join is a Product Join | try a compromise join where the two unconnected tables are Product-joined in the first join and that result is then joined with a third table. |
| 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 tables in the join. Call this table_A.

The list of values associated with the set of IN conditions is product joined with the other table, table_B, and that result is then joined with table_A. If table_B happens to be the result of joining two unconnected tables, then the resultant join plan is a star join plan similar to what is described in "Star and Snowflake Join Optimization" on page 201.

Consider the following schema and query, for example:

| Table Name | Table Type | Relevant Index Columns | Index Type |
|---|---|---|---|
| DailySales | Fact | sku_id<br>locn_nmbr<br>day_dt | NUPI |
| | | locn_nmbr<br>day_dt | NUSI |
| CorpDay | Dimension | day_dt | UPI |
| Locn | Dimension | locn_nmbr | UPI |
| AllocSku | Dimension | sku_id | UPI |

Now consider how the Optimizer treats the following query against these tables:

```
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 condition, and then joining that result with DailySales.

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.

| IF the conditions are ... | THEN the ... |
|---|---|
| met | remaining joins of the best plan are also committed. |
| not met | second join is also committed if it joins another table with the result of the first join via the primary index for the newly joined table. |

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:

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

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

The explicit connections are these.

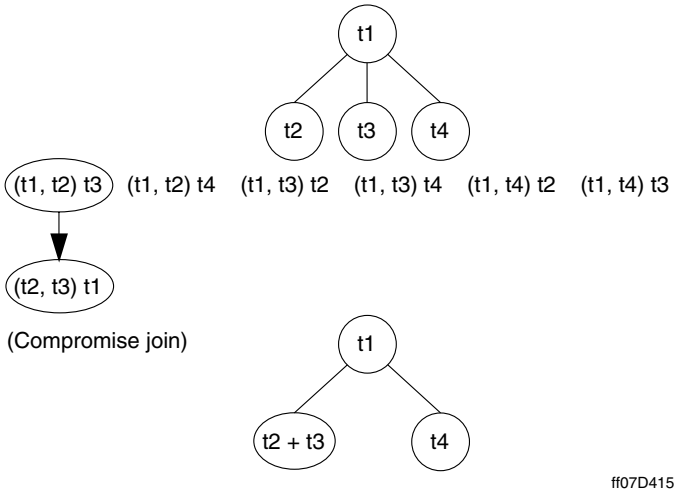
- t1 is connected to t2 because of the condition x1 = x2
- t1 is connected to t3 because of the condition y1 = y3
- t1 is connected to t4 because of the condition z1 = y4

We know from step 6 on page 137 that when the following conditions are true, the Optimizer tries to make a compromise join where the two unconnected tables are product-joined and the result is joined with the third table:

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

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

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



# Lookahead Join Planning

## Introduction

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

Depending on certain well-defined circumstances (see "Five-Join Lookahead Processing of n-Way Joins" on page 143), the Teradata Database uses both one-join and five-join lookahead methods to determine optimal join plans.

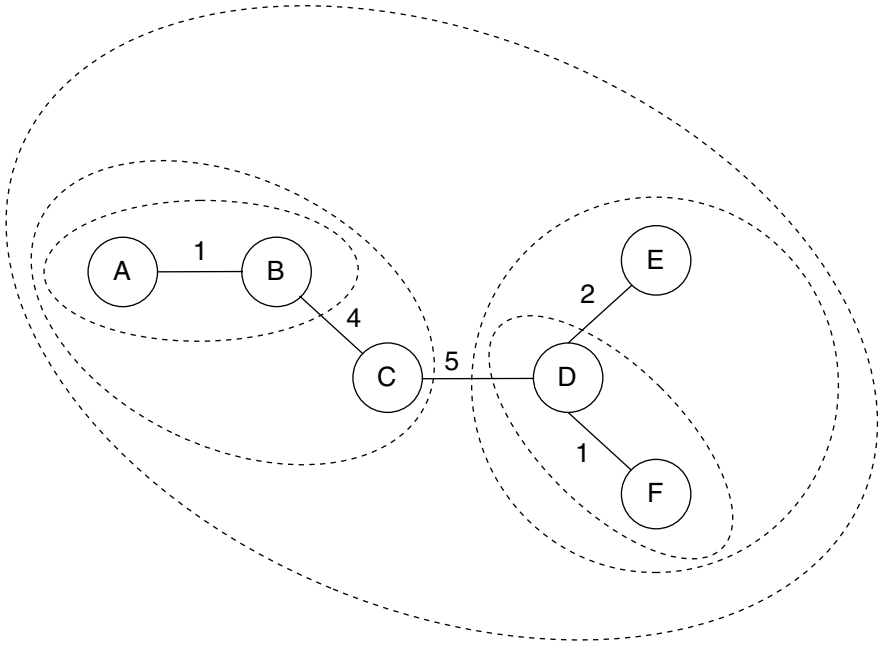
## One-Join Lookahead Processing of *n*-Way Joins

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, is uses a recursive greedy algorithm to search and evaluate connected relations. In this context, the term *greedy* means the search generates the next logical expression for evaluation based on the result of the costing determined by the previous step. From the perspective of search trees, a greedy algorithm always begins at the *bottom* of the tree and works its way toward the top.

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

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

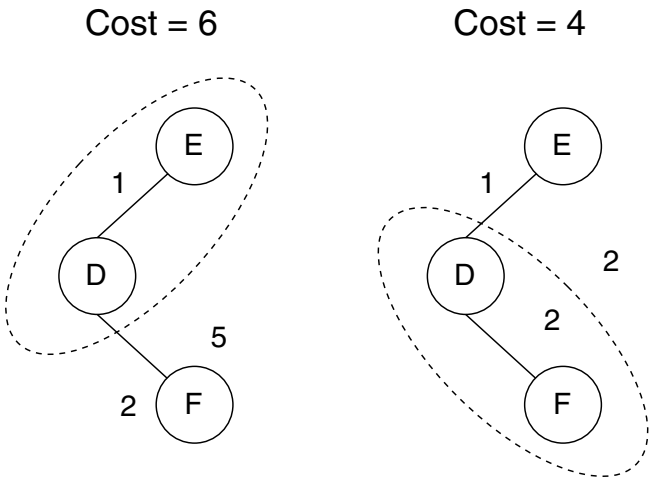
The following graphic illustrates the result of an Optimizer analysis of the join space for a 6-way join. The relative cost of each join is given by an integer within the selected join sequence.




These relationships are explicitly stated in the following table:

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

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




ff07D416

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

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

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

```
((DE)F) = 1 + 5 = 6

((DF)E) = 2 + 2 = 4
```

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

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

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

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

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

# Join Methods

## General Remarks

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

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

## Guidelines

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

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

  For further information on the Optimizer form of the COLLECT STATISTICS statement, see *SQL Reference: Data Definition Statements*.
- 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 Reference: Data Manipulation Statements*.

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

## Summary of the Most Commonly Used Join Algorithms

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

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

# Product Join

## Definition

The Product Join compares every qualifying row from one table to every qualifying row from the other table and saves the rows that match the WHERE 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 tables.

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.
- There are ORed join conditions.
- A referenced table 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, the Optimizer might substitute a form of Hash Join (see "Hash Join" on page 160 and "Dynamic Hash Join" on page 167) in place of an *equality* Product Join.[39]

## Product Join Types

Seven families of product join methods exist:

- Inner product join.
- Left outer product join.
- Right outer product join.
- Full outer product join.
- Piggybacked product join.
- Inner inclusion product join.

  Outer inclusion product join.
- Inner exclusion product join.

  Outer exclusion product join.

39. Hash Joins are only used for equality conditions.

## How a Product Join is Processed

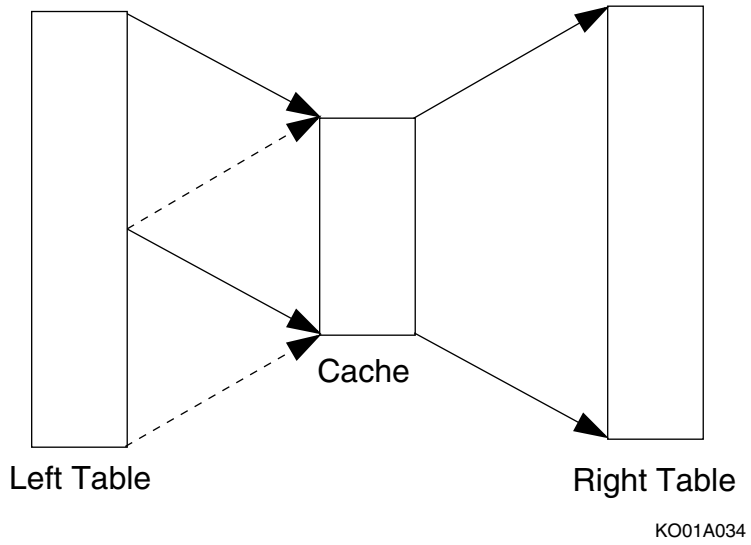
The following list outlines the Product Join process:

1. Cache the left table rows.
2. Join each row of the right table with each row from the cached left table.
3. End of process.

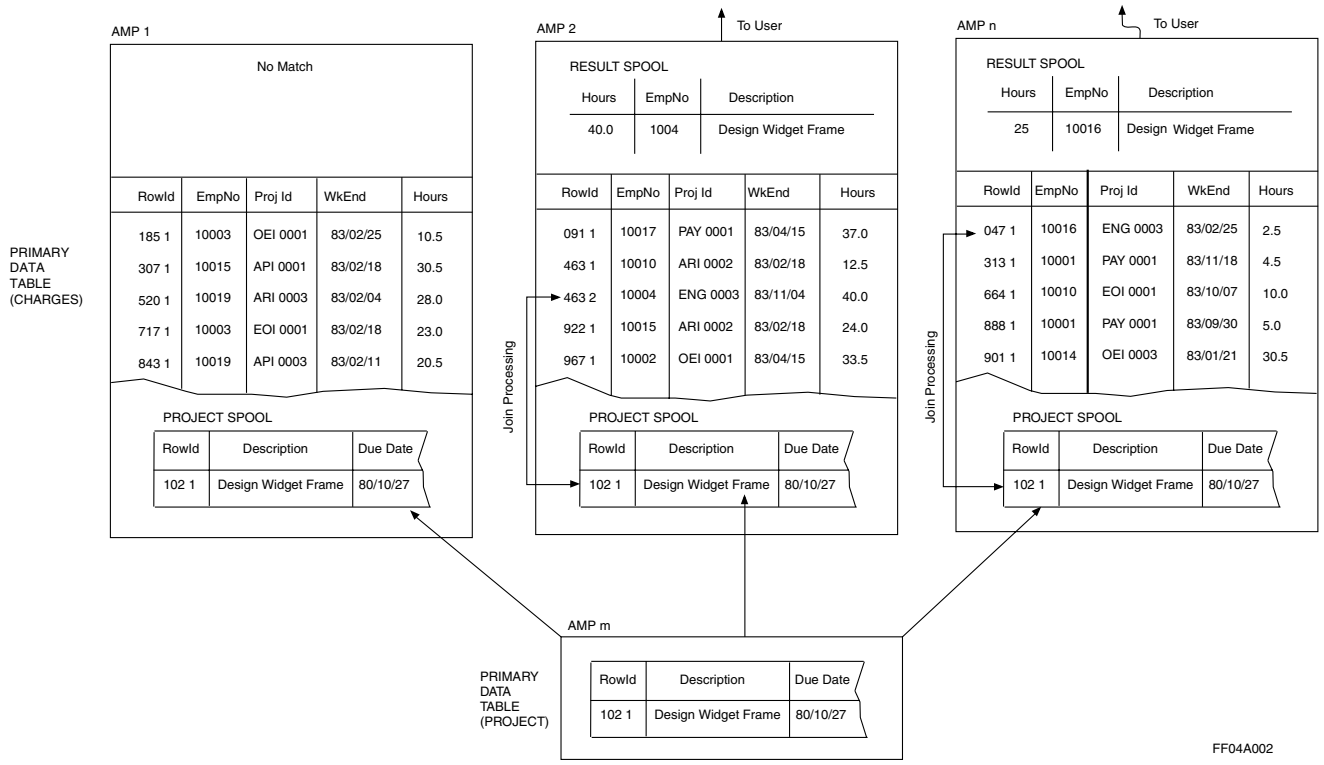
An overall join condition is a WHERE constraint that links tables to be joined on a column that is common to each, as shown in the following example:

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

The following graphic illustrates the generic Product Join process:



The following graphic illustrates a concrete Product Join:




## Cost of a Product Join

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

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

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

## Product Join With Dynamic Partition Elimination

Dynamic Partition Elimination is an option considered for a duplicated direct product join where at least one of the input relations is a PPI table and there is an equality join condition on the partitioning column.

For a product join with dynamic partition elimination, the left relation is sorted by rowkey using the same partition 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.

## 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 table.

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:

| Equation element ... | Specifies the ... |
|---|---|
| LeftRelCost | cost of reading the left relation. |
| RightRelCost | cost of reading the right relation. |
| Reduction | amount of I/O performed for the right relation, calculated as follows:<br>$\left(\frac{\left(\frac{\text{NumBlocks}}{\text{NonEmptyPartNum}}\right)}{\text{NumBlocks}}\right)$<br>where:<br>**Equation element ... / Specifies ...**<br>NumBlocks — the number of data blocks in the right relation.<br>NonEmptyPartNum — the number of non-empty partitions in the right relation. |
| NumCache | number of non-empty partitions in the left relation. |

## Example 1

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

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

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

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

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

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

Note that blocks from one table may have to be read several times if all of the rows of a Product Join cannot be 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 table definitions:

**Employee**

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

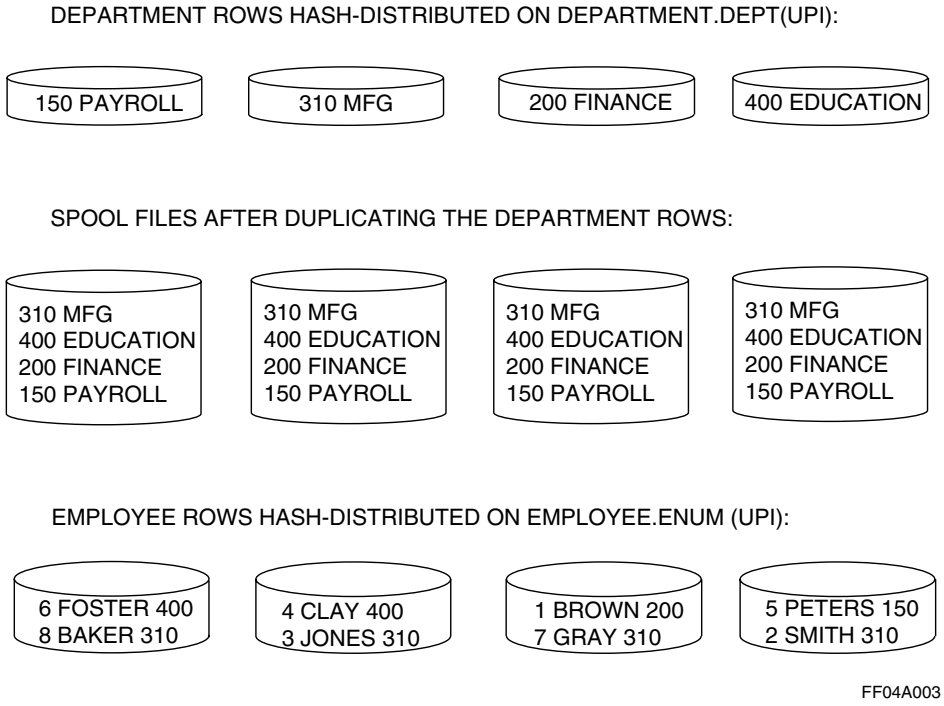
**Department**

| Dept | DeptName |
|---|---|
| PK | |
| UPI | |
| 400 | Education |
| 150 | Payroll |
| 200 | Finance |
| 310 | Manufacturing |

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

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

The rows are redistributed as shown in the following graphic:




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

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

# Merge Join

## Definition

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 145) 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 152)

  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 153)

  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 153)

  Fast path inner inclusion merge join (see "Inclusion Merge Join" on page 187)
- Slow path inner merge join (see "Slow Path Inner Merge Join" on page 152)

  Slow path inner inclusion merge join (see "Inclusion Merge Join" on page 187)
- Exclusion merge join (see "Exclusion Merge Join" on page 182)
- Fast path left outer join

  Fast path left outer inclusion merge join (see "Inclusion Merge Join" on page 187)
- Slow path left outer join

  Slow path left outer inclusion merge join (see "Inclusion Merge Join" on page 187)
- Fast path right outer merge join
- Slow path right outer merge join
- Full outer merge join

Depending on the relative costs, the Optimizer might substitute a form of Hash Join (see "Hash Join" on page 160 and "Dynamic Hash Join" on page 167) 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 $R_2$.

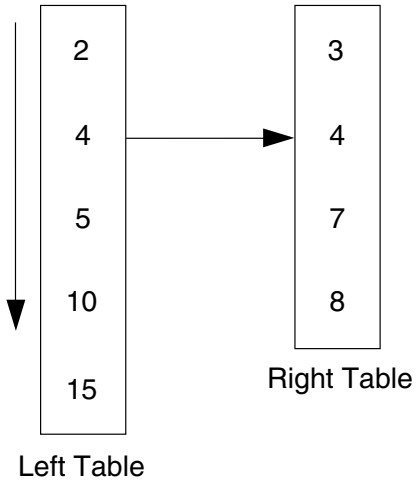
The Optimizer selects the least costly combination to be used for further comparisons.

## 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:




KO01A030

## Fast Path Inner Merge Join

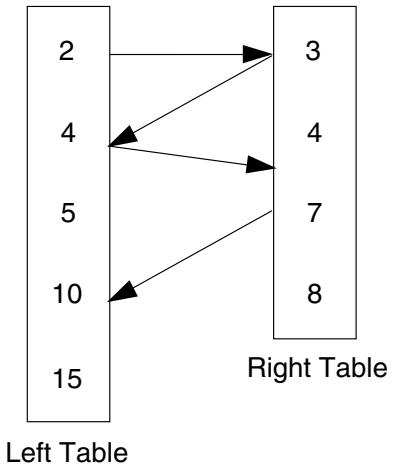
The process applied by the fast path Merge Join algorithm is:

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 >= to that of the left table row.

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

3. End of process.

The following graphic illustrates the generic fast path Merge Join process:




KO01A031

## PPI Sliding Window Merge Join

When either or both of the input relations for a merge join is a PPI table, the Optimizer uses a a sliding window merge join. Sliding window joins follow the general principle of structuring each of the left and right relations into windows of appropriate[40] 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 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.

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 Optimizer applies dynamic partition elimination to the operation.

## Merge Join Strategies

Merge Joins use one of the following distribution strategies:

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

40. Appropriate in terms of the number of partitions required as determined by the Optimizer based on the available memory.

| FOR a representation of this strategy ... | See this illustration ... |
|---|---|
| Hash redistribution | "Merge Join Row Distribution (Hash Redistribution)" on page 157. |
| Table duplication | "Merge Join Row Distribution (Duplicate Table)" on page 158. |
| Index matching | "Merge Join Row Distribution (Matching Indexes)" on page 159. |

## Example 1

The following SELECT statement determines who works in what location:

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

The following 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.

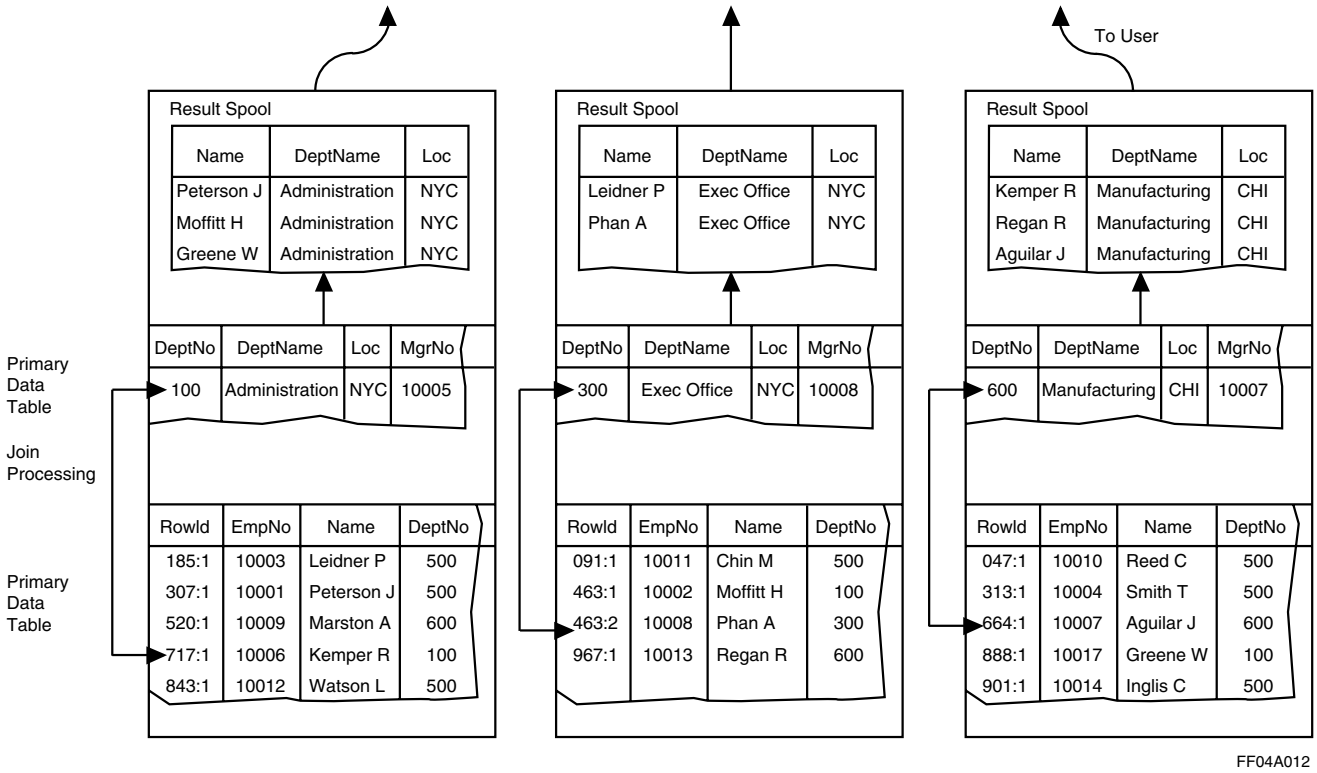
   This puts them in the same order as the Department rows on the AMP.
3. The hash value of the first row from the Department table will be used to read the first row with the same or bigger row hash from the Employee spool.

   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:

**Employee**

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

**Department**

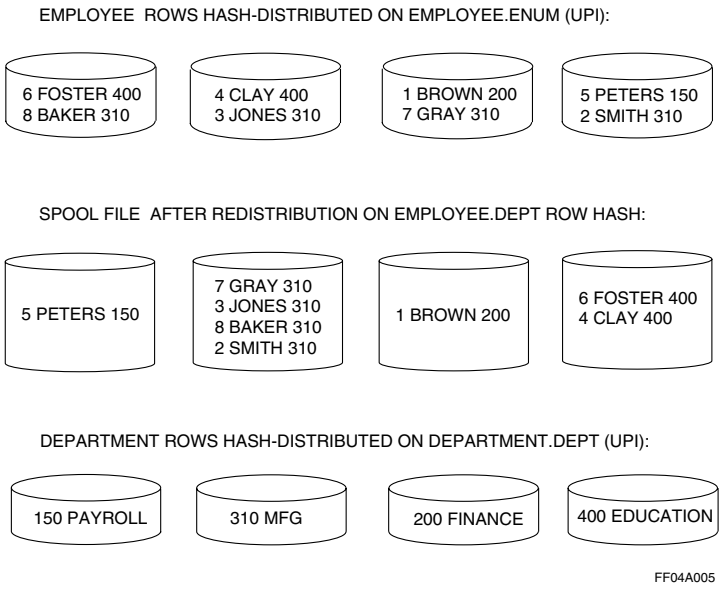
| Dept | DeptName |
|---|---|
| PK | |
| UPI | |
| 400 | Education |
| 150 | Payroll |
| 200 | Finance |
| 310 | Mfg |

One of the Merge Join row redistribution methods (see “Merge Join Row Distribution (Hash Redistribution)” on page 157 or “Merge Join Row Distribution (Duplicate Table)” on page 158) is used if you perform the following SELECT statement against these tables:

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

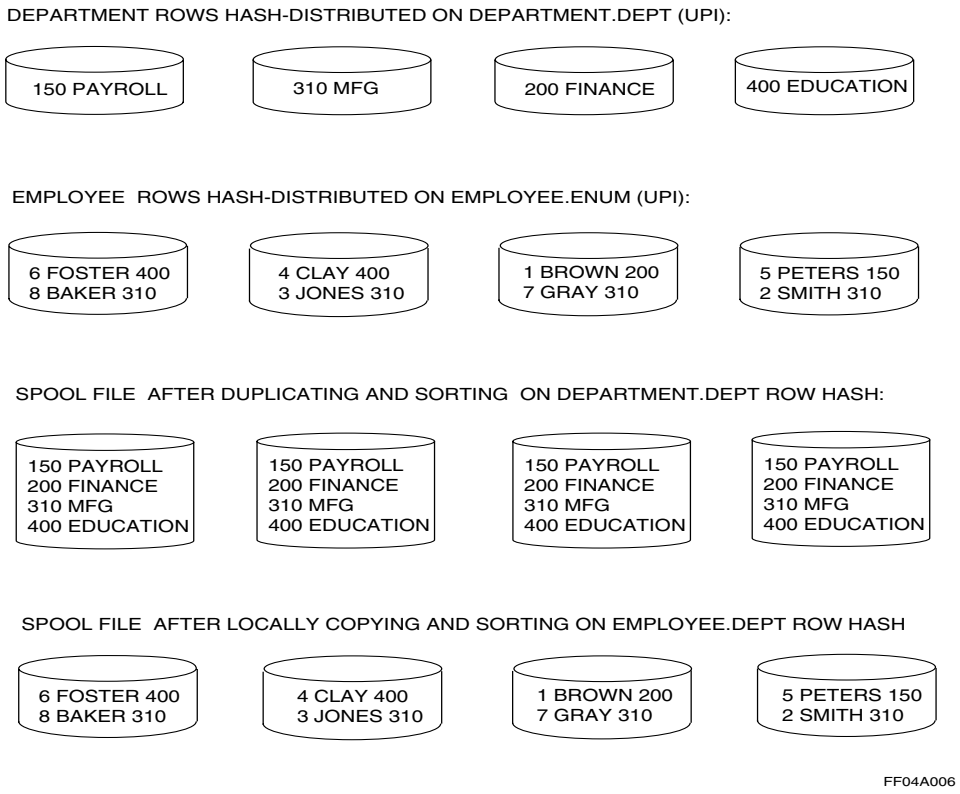
## Merge Join Row Distribution (Hash Redistribution)

The following graphic shows a Merge Join row distribution using hash redistribution:



## Merge Join Row Distribution (Duplicate Table)

The following graphic shows a Merge Join row distribution by duplicating a table:




## Example 3

This example uses the following Employee and Employee_phone table definitions.

**Employee**

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

**Employee_Phone**

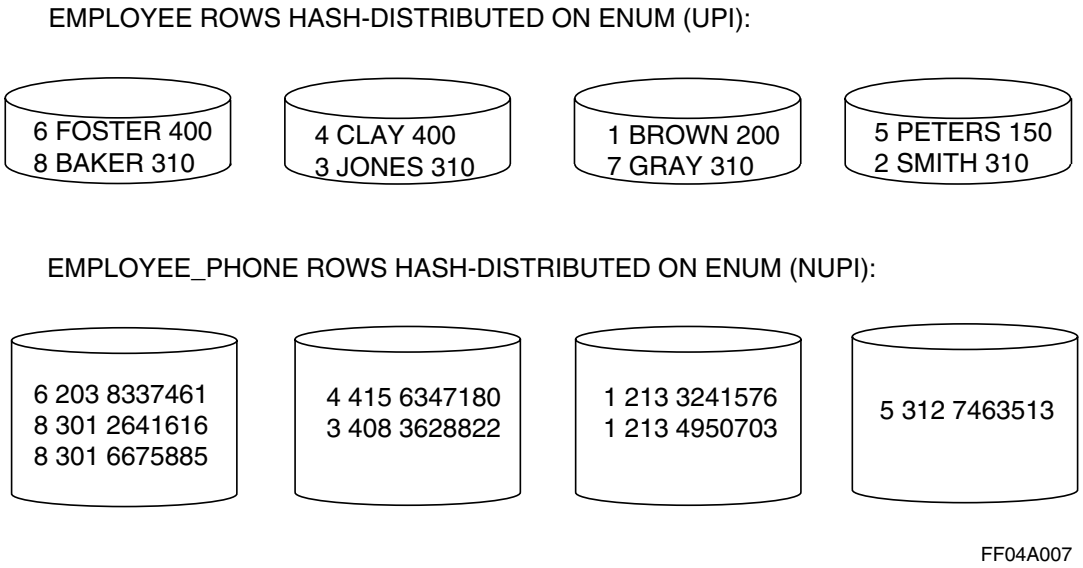
| Enum | AreaCode | Phone |
|---|---|---|
| PK | | |
| FK | | |
| NUPI | | |
| 1 | 213 | 4950703 |
| 1 | 408 | 3628822 |
| 3 | 415 | 6347180 |
| 4 | 312 | 7463513 |
| 5 | 203 | 8337461 |
| 6 | 301 | 6675885 |
| 8 | 301 | 2641616 |
| 8 | 213 | 4950703 |

The matching indexes row distribution strategy shown in “Merge Join Row Distribution (Matching Indexes)” on page 159 is used if you perform the following SELECT statement against these tables.

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

## Merge Join Row Distribution (Matching Indexes)

The following graphic shows a Merge Join row distribution by matching indexes:



# Hash Join

## Introduction

Hash Join is a method that performs better than Merge Join (see "Merge Join" on page 151) 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 167) 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 164 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:

| Term | Definition |
|---|---|
| Build Table | The smaller of the two join tables.<br>So named because it is used to build the hash table. |
| Fanout | The maximum number of partitions to be created at each partitioning level. |
| Hash Join | 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. |
| Hash Table | 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. |
| Probe Table | The larger of the two join tables in the hash join.<br>Rows from this table are used to probe rows in the hash table for a hash match before the join conditions are evaluated. |

| Term | Definition |
|---|---|
| Partition | A segment of a table in a hash join.<br>Tables are segmented into a number of partitions using the equation described in "Assigning Rows to a Hash Join Partition" on page 163).<br>Partitions can be memory-resident, in which case they also constitute the hash table, or a spool file.<br>The limit on the number of partitions for a hash join operation is 50.<br>Note that Hash Join partitions have no relationship to the partitions of a Partitioned Primary Index (see *SQL Reference: Data Definition Statements* and *Database Design*). They are entirely different, unrelated things. |

## Classic Hash Join

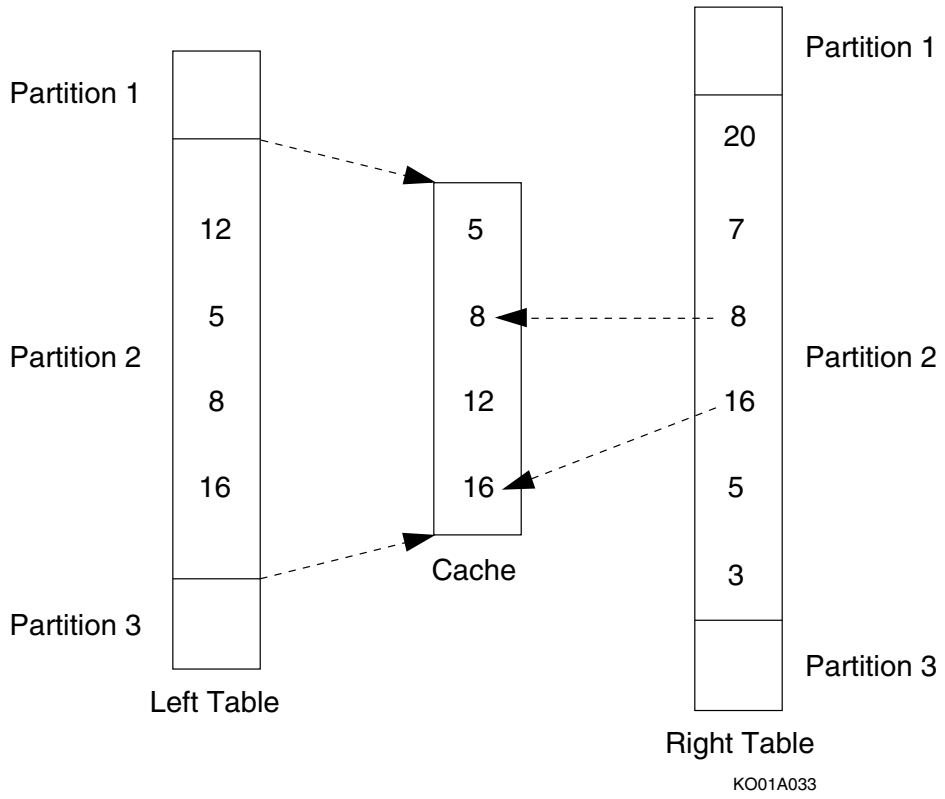
Classic Hash Join is applicable when the entire build table fits into available memory. The system reads rows from the build relation directly into a memory-resident hash table. The process then reads rows from the probe table and uses them to individually probe the hash table for a match. A result row 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.

The 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 167). 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 162 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 is 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 fields in the DBSControl record (see “Hash Join Control Variables” on page 164). The default partition size for a non-duplicated (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 163).

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 166, 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:

```
partition_number = (row_hash_value)(MOD(fanout))
```

where:

| Equation element ... | Specifies the ... |
|---|---|
| *partition_number* | number of the memory-resident Hash Join partition to which a row from the table is assigned. |
| *row_hash_value* | row hash value for the row in question. See *Database Design* for details. |
| MOD | modulo function. |
| *fanout* | maximum number of partitions to be created at each partitioning level in the Hash Join operation. |

## 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 DBSControl utility to size build table partitions proportionately to their parent hash table (see "SkewAllowance" on page 165).

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 fields of the DBSControl record (see "HTMemAlloc" on page 165 and "HTMemAllocBase" on page 165). 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 fields HTMemAlloc and SkewAllowance using the DBSControl utility (see *Utilities*). Use them to optimize the performance of your Hash Joins. HTMemAllocBase is an internal DBSControl field that can only be accessed by Teradata support personnel. Contact your Teradata technical support team if you suspect the value of this field needs to be changed.

| DBSControl Field | Function |
|---|---|
| HTMemAlloc | Varies the hash table size by calculating a percentage of the HTMemAllocBase value, and is used by the Optimizer in the following formula:<br>$\text{Hash table size} = \frac{\text{HTMemAlloc}}{100} \times \frac{\text{Amount of memory}}{\text{AMP}}$<br>The larger the specified percentage, the larger the hash table. For virtually all applications, you should not specify a value larger than 5. The recommended value for most applications is 2, which is the default.<br>Valid input values range from 0 to 10 (percent), where a value of 0 disables the hash join feature.<br>The recommended maximum value is in the range of 3 to 5. Higher values can allocate so much memory to Hash Join processing that the system either hangs or crashes. |
| HTMemAllocBase[a] | Varies the size of memory allocated per AMP for the hash table.<br>Valid input values range from 1 to 1024 Mbytes.<br>The default value is 10 Mbytes.<br>Note that HTMemAllocBase is the $\frac{\text{Amount of memory}}{\text{AMP}}$ factor in the equation used to determine hash table size in "HTMemAlloc" on page 165. |
| SkewAllowance | Provides for data skew by making the size of each partition smaller than the hash table.<br>Valid input values range from 20 to 80 (percent).<br>The default is 75 (percent), indicating that the partition size is set to 25 percent of the hash table to allow the actual partition to be four times more than the estimate made by the Optimizer before it is too large to fit into the hash table. |

a. This row is provided for information purposes only. HTMemAllocBase is an internal DBSControl field, so you can neither view nor change it. Please consult your Teradata support team to determine whether the value of HTMemAllocBase needs to be adjusted for your site.

## 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

LeftPrepCost + LeftRelCost + RightRelCost + HTBuildCost + HTProbCost

The cost of a regular hash join is computed as follows:

Cost of regular hash join = LeftPrepCost + RightPrepCost + LeftRelCost + RightRelCost + HTBuildCost + HTProbeCost

The cost of a dynamic hash join is computed as follows:

Cost of dynamic hash join = LeftPrepCost + LeftRelCost + RightRelCost + HTBuildCost + HTProbeCost

where:

| Costing term ... | Specifies the cost of ... |
|---|---|
| LeftPrepCost | preparing the left relation for the hash join. |
| RightPrepCost | preparing the right relation for the hash join. |
| LeftRelCost | reading the left relation. |
| RightRelCost | reading the right relation. |
| HTBuildCost | building the hash array. |
| HTProbeCost | probing the hash array. |

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 162).

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 164).

```
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%
```

## Dynamic Hash Join

Dynamic Hash Join[41] provides the ability to do an equality join directly between a small table and a large table on non-primary 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 non-primary 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.

41. Sometimes called Hash Join "On the Fly."

# 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").

## 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 169.

Remote Nested Join is described in "Remote Nested Join" on page 175.

## Process

The 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 table.
2. Use that row to locate the AMP having the matching rows on which the join is to be made.
3. End of process.

## 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 188). 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 170)
- Fast path (see "Fast Path Local Nested Join" on page 174)

## 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 188.

| IF the equality condition is on this index type … | THEN the left table is … |
|---|---|
| USI | 1 Hash-redistributed based on the joined field.<br>2 Nested Joined with the right table.<br>3 The resulting rows are redistributed by row hashing the rowID of the right table rows.<br>4 The rowID is used to retrieve the data rows from the right table to complete the join.<br>5 End of process. |
| NUSI | 1 Duplicated on all AMPs.<br>2 Nested Joined with the right table.<br>3 The resulting rows are redistributed by row hashing the rowID of the right table rows.<br>4 The rowID is used to retrieve the data rows from the right table to complete the join.<br>5 End of process. |

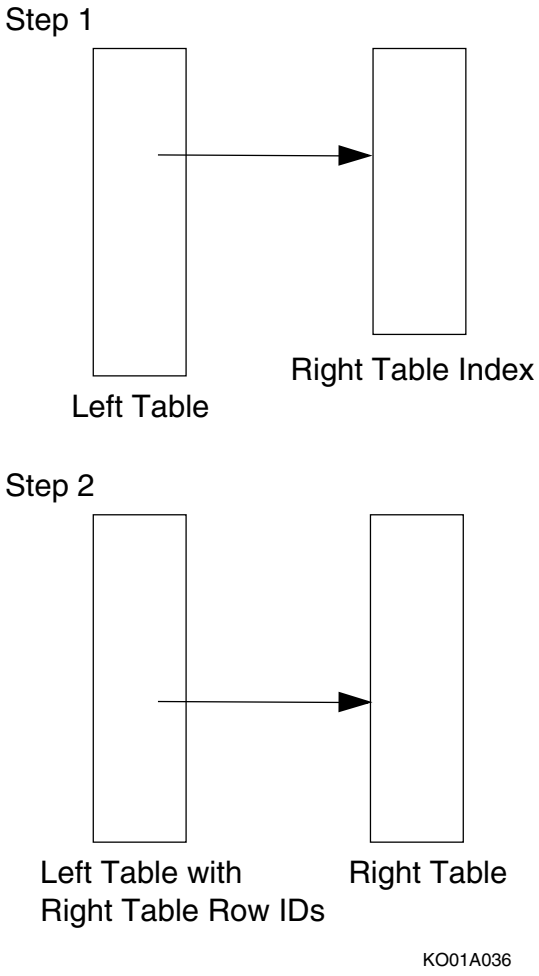
# Slow Path Local Nested Join

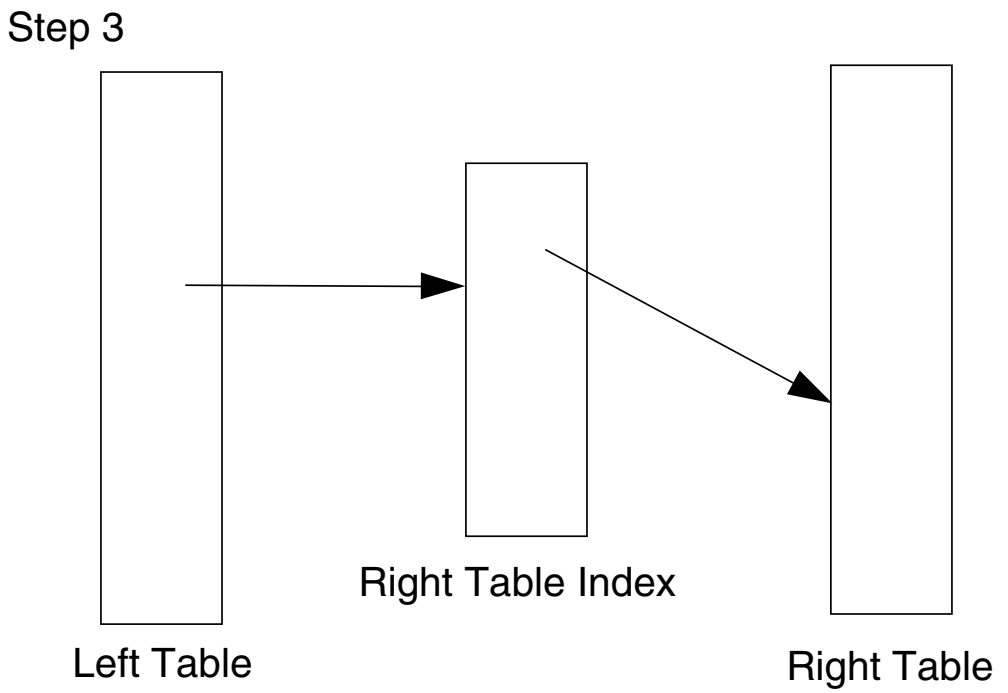
## Slow Path Local Nested Join Process

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:






## Example 1

The following is an example of a query processed using a slow path local Nested Join.

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

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

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

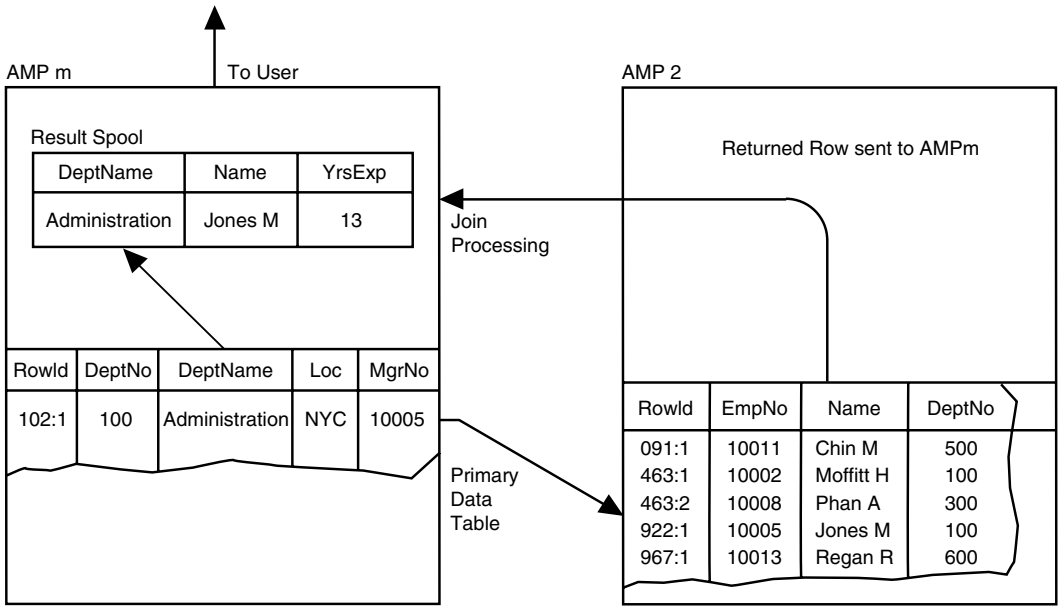
Note that this MgrNo value is the same as the value of EmpNo (the unique primary index of the Employee table) for the employee who manages Department 100. Thus, the hash code that is calculated for MgrNo is the same as the hash code for the equivalent EmpNo value.

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

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

This two-AMP process is illustrated in “Example of a Local Nested Join Operation” on page 173.

## Example of a Local Nested Join Operation




## Example 2

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

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

**Employee**

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

**Department**

| Dept | DeptName |
|---|---|
| PK | |
| UPI | |
| 400 | Education |
| 150 | Payroll |
| 200 | Finance |
| 310 | Mfg |

# Fast Path Local Nested Join

## Fast Path Local Nested Join Process

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

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

1 Read a row from the left base table and record its hash value.

2 Read the next row from the right NUSI subtable that has a row hash >= to that of the left base table row.

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

3 End of process.

## Example

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

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

# Remote Nested Join

## Definition

Remote Nested 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:

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

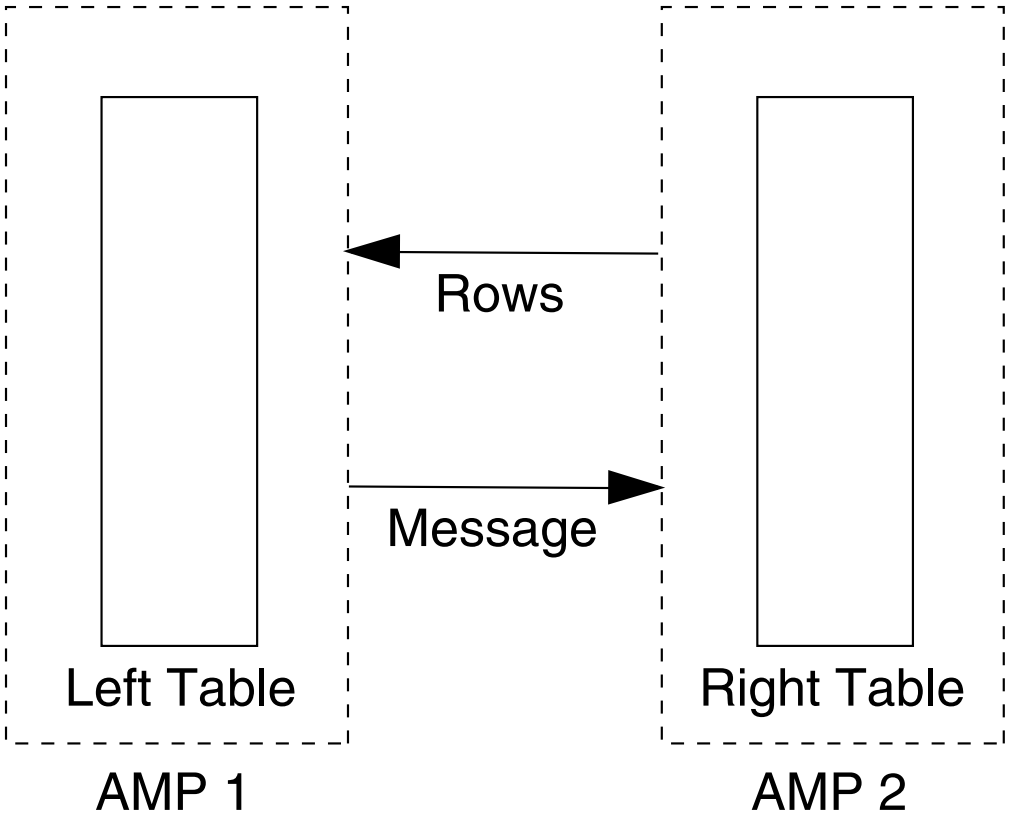
When there is no such join condition, then the index of the second (right) table must be defined with a constant as illustrated by Examples 1, 2, and 3 if a remote Nested Join is to be used.

## 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.

The following graphic illustrates the remote Nested Join process:




## 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)
- Non-indexed column that is matched to an index

If there is such a join condition, and the conditions of the first table match a column of the primary or secondary index of the second table, then the 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

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

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

## Example 2

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

## Example 3

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

# Nested Join Examples

## Introduction

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

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

## Table Definitions

For these examples, assume the following table definitions:

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

## Test Query

The following query is tested against this database, both using and without using Nested Join:

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

# Join Plan Without Nested Join

## Unoptimized Join Plan

Without using a Nested Join, the Optimizer generates the following join plan:

| Operation | Joined Tables | Total Processing Time |
|---|---|---|
| Spool 1:Product Join | duped table_1,<br>direct table_2 | 1 hour 15 minutes |

## Completion Time

The total estimated completion time is 1 hour 15 minutes.

## EXPLAIN Output for Unoptimized Join Plan

The following EXPLAIN output is generated when Nested Joins are not used:

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

# Join Plan With Nested Join

## Optimized Join Plan

Using Nested Joins, the Optimizer generates the following join plan:

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

## Completion Time

The total estimated completion time is 0.46 seconds.

The estimated performance improvement factor is 9782.

## EXPLAIN Output for Optimized Join Plan

The following EXPLAIN output is generated:

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

# Exclusion Join

## Definition

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.

## 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 182)
- Exclusion Product Join (see "Exclusion Product Join" on page 185)

# 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:

```
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 181), with the following exceptions:

- Multiple rows are retrieved in stage 2.
- Stages 2 and 4 are combined in the exclusion merge join, and redistribution occurs instead of duplication.
- Stage 3 is removed.

- Stage 5 is changed to an exclusion product.

## Example 2

Consider the following Employee and Customer tables:

**Employee**

| Enum | Name | Job_code |
|---|---|---|
| PK | | FK |
| UPI | | |
| 1 | Brown | 512101 |
| 2 | Smith | 412101 |
| 3 | Jones | 512101 |
| 4 | Clay | 412101 |
| 5 | Peters | 512101 |
| 6 | Foster | 512101 |
| 7 | Gray | 413201 |
| 8 | Baker | 512101 |

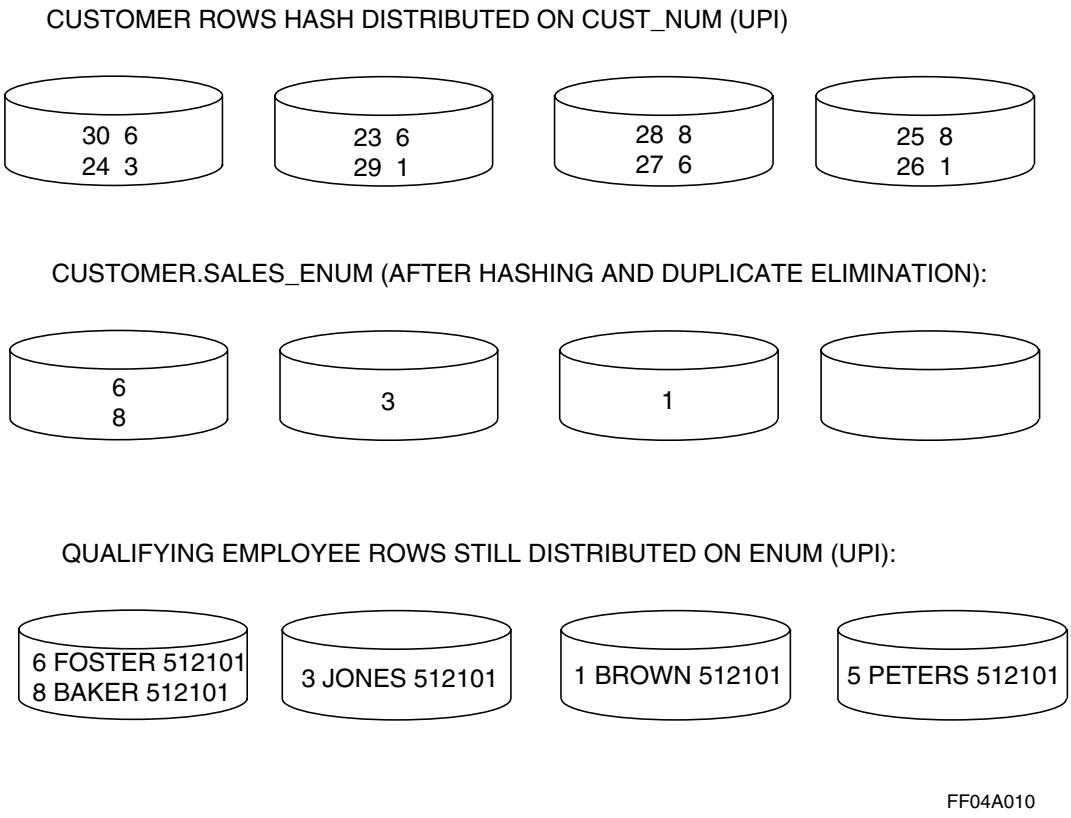
**Customer**

| Cust | Sales_enum |
|---|---|
| PK | FK |
| UPI | |
| 23 | 6 |
| 24 | 3 |
| 25 | 8 |
| 26 | 1 |
| 27 | 6 |
| 28 | 8 |
| 29 | 1 |
| 30 | 6 |

The graphic "Exclusion Merge Join Row Distribution" on page 184 illustrates the row redistribution caused by the following SELECT statement:

```
SELECT name
FROM employee
WHERE job_code = 512101
AND   enum NOT IN
 (SELECT sales_enum
  FROM customer);
```

## Exclusion Merge Join Row Distribution



# Exclusion Product Join

## Exclusion Product Join Process

The process applied by the Exclusion Product 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 requests names of employees who do not work in Chicago:

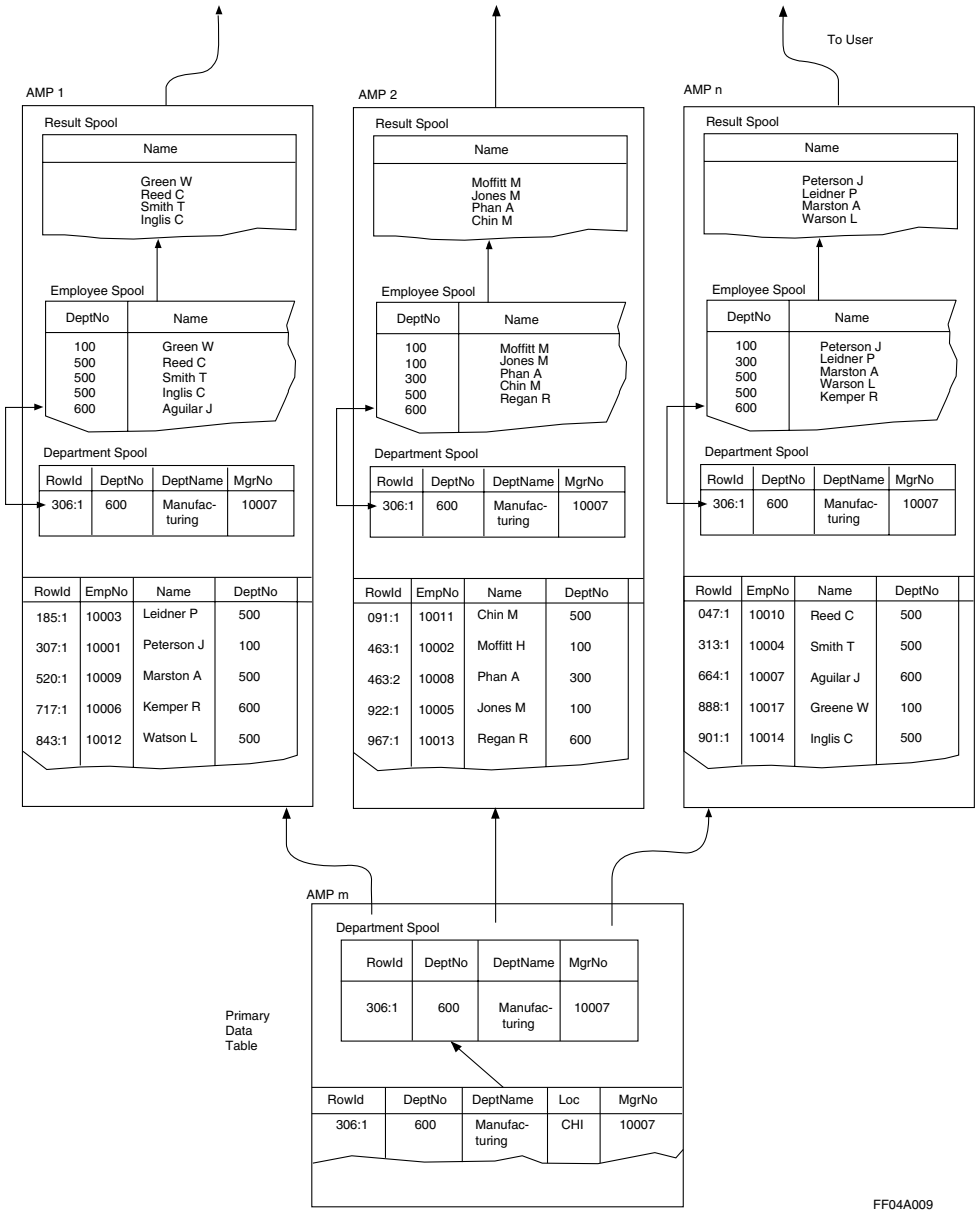
```
SELECT Name
FROM Employee
WHERE DeptNo NOT IN
 (SELECT DeptNo
  FROM Department
  WHERE Loc = 'CHI');
```

Because the subquery has only one row, an Exclusion Product Join is used with the following process:

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 186 illustrates this process.

## Exclusion Product Join Operation



# Inclusion Join

## Definition

An Inclusion Join is a Product or Merge Join where the first right table row that matches the left row is joined.

There are two types of Inclusion Join.

- Inclusion Merge Join
- Inclusion Product Join

## 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.

# RowID Join

## Introduction

The RowID Join is a special form of the Nested Join. The Optimizer selects a RowID Join instead of a Nested Join when the first condition in the query specifies a literal for the first table. This value is then used to select a small number of rows which are then equijoined with a secondary index from the second table.

## Rules

The Optimizer can select a RowID Join only if both of the following conditions are true:

- The WHERE clause condition must match another column of the first table to a NUSI or USI of the second table.
- Only a subset of the NUSI or USI values from the second table are qualified via the join condition (this is referred to as a weakly selective index condition), and a Nested Join is done between the two tables to retrieve the rowIDs from the second table.

## Process

Consider the following generic SQL query:

```
SELECT *
FROM table_1, table_2
WHERE table_1.NUPI = value
AND   table_1.column = table_2.weakly_selective_NUSI;
```

The process involved in solving this query is as follows:

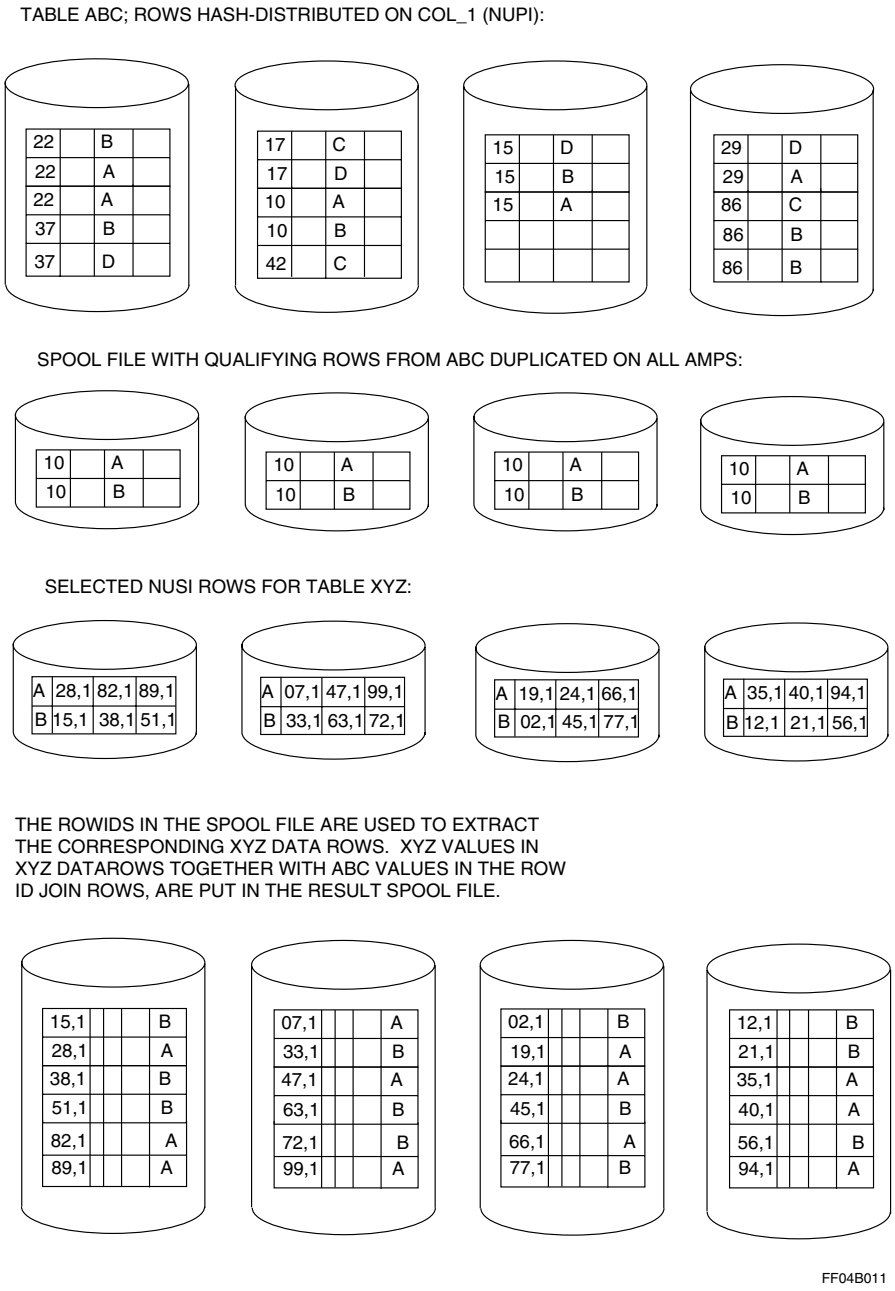
1. The qualifying table_1 rows are duplicated on all AMPS.
2. The value in the join column of a table_1 row is used to hash into the table_2 NUSI (similar to a Nested Join).
3. The rowIDs are extracted from the index subtable and placed into a spool file together with the corresponding table_1 columns. 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 189.

## RowID Join

The following graphic demonstrates a RowID Join:




## Example

Assume the following SELECT statement is performed. The first WHERE condition is on a NUPI and the second is on a NUSI. The Optimizer specifies a RowID Join to process the join.

```
SELECT *
FROM table_1, table_2
WHERE table_1.column_1 = 10
AND   table_1.column_3 = table_2.column_5;
```

# Correlated Joins

## Introduction

Correlated Join constitutes a class of join methods developed to process correlated subqueries.

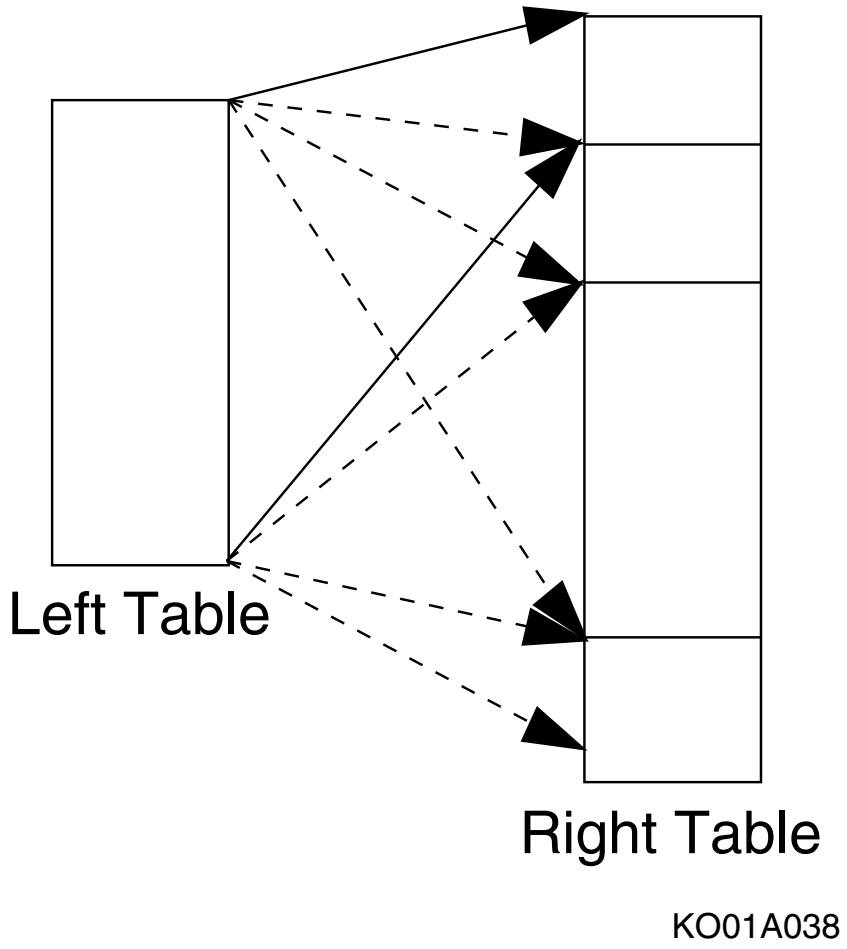
Some types of Correlated Join are extensions of the following more general join types:

- Inclusion Merge Join (see "Inclusion Merge Join" on page 187)
- Exclusion Merge Join (see "Exclusion Merge Join" on page 182)
- Inclusion Product Join (see "Inclusion Product Join" on page 187)
- Exclusion Product Join (see "Exclusion Product Join" on page 185)

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 187) 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 182.

  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 187.
- Correlated Exclusion Product Join

  Correlated version of standard Exclusion Product Join. See "Exclusion Product Join" on page 185.
- 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.

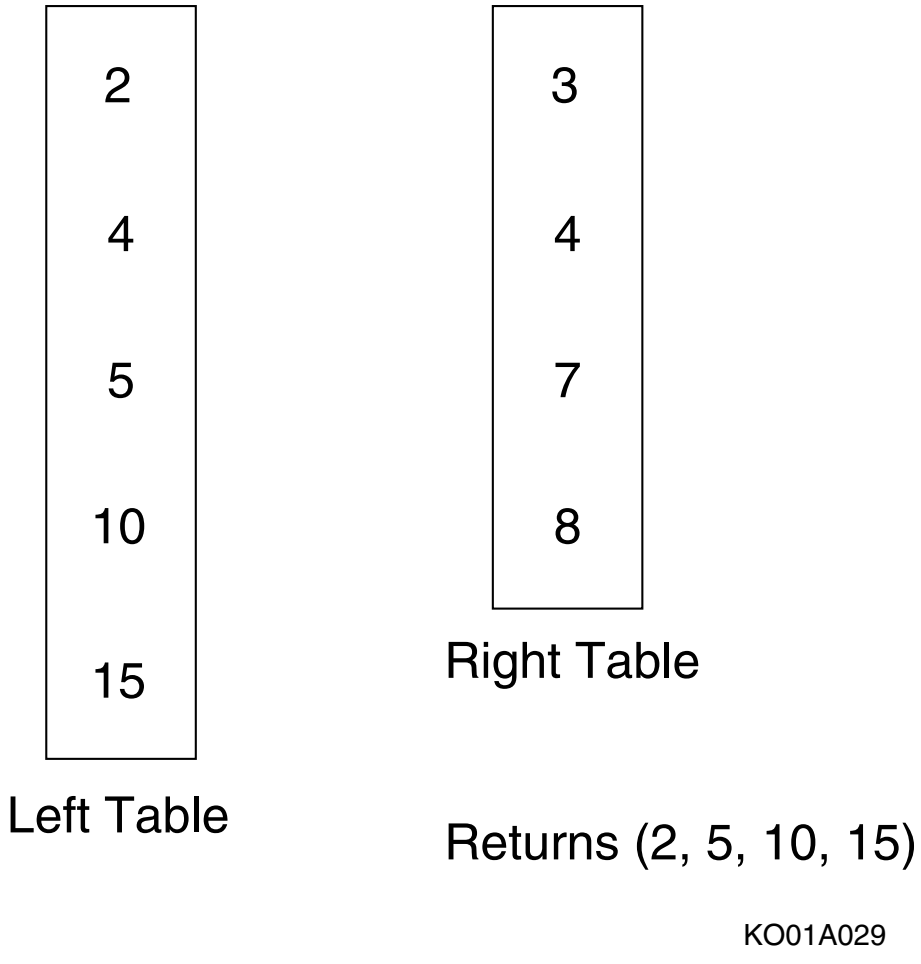
# Minus All Join

## 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.




4. End of process.

# Relational Query Optimization References

The following references are helpful for acquiring a basic understanding of how relational query optimizers work:

| Topic | Reference |
|---|---|
| Origins of Relational Query Optimizers: SQUIRAL | John Miles Smith and Philip Yen-Tang Chang, "Optimizing the Performance of a Relational Algebra Database Interface," *Communications of the ACM*, 18(10):568-579, 1975.<br>Philip Y. Chang, "Parallel Processing and Data Driven Implementation of a Relational Data Base System," *Proceedings of 1976 ACM National Conference*, 314-318, 1976.<br>SQUIRAL was a research project at the University of Utah based on a parallel implementation of the relational model. These papers describe its optimizer. |
| Origins of Relational Query Optimizers: Peterlee Relational Test Vehicle (PRTV) | P.A.V. Hall, "Optimization of Single Expressions in a Relational Data Base System," *IBM Journal of Research and Development*, 20 (3):244-257, 1976.<br>Available as a free PDF download from the following URL: http://domino.research.ibm.com/tchjr/journalindex.nsf/4ac37cf0bdc4dd6a85256547004d47e1/1f0b4f94986b761a85256bfa0067f7a6?OpenDocument<br>S.J.P. Todd, "The Peterlee Relational Test Vehicle—a system overview," *IBM Systems Journal*, 15(4):285-308, 1976.<br>Available as a free PDF download from the following URL: http://domino.research.ibm.com/tchjr/journalindex.nsf/e90fc5d047e64ebf85256bc80066919c/8ce0a2b6d80786d985256bfa00685ad6?OpenDocument<br>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.<br>Unfortunately, very little documentation of the project, most of which was in the form of internal IBM technical reports, is available.<br>These two papers describe the PRTV optimizer, which, like the rest of the PRTV architecture, was quite advanced for its time. |

| Topic | Reference |
|---|---|
| Origins of Relational Query Optimizers: System R | M.M. Astrahan, M. Schkolnick, and W. Kim, “Performance of the System R Access Path Selection Mechanism,” *Information Processing 80*, IFIP, North-Holland Publishing Co., 1980, pp. 487-491.<br><br>The first widely available report on the performance of the System R optimizer. |
| | M.W. Blasgen and K.P. Eswaran, “Storage and Access in Relational Data Bases,” *IBM Systems Journal*, 16(4):363-377, 1977.<br><br>Despite its deceptive title, this is the earliest widely available paper published on what was to become the System R cost-based query optimizer.<br><br>Published earlier as M.W. Blasgen and K.P. Eswaran, “On the Evaluation of Queries in a Data Base System,” *IBM Research Report* FJ 1745, April, 1976. |
| | P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of SIGMOD 1979*, 23-34, 1979.<br><br>Surveys the query optimizer designed for the IBM System R prototype relational system. This paper is generally regarded to be the starting point for all cost-based SQL query optimizers in commercially available relational database management systems. Implicitly divides query optimization into 3 components: cost estimation, plan selection, and cost comparison. |
| Origins of Relational Query Optimizers: INGRES | Eugene Wong and Karel Youssefi, “Decomposition—A Strategy for Query Processing,” *ACM Transactions on Database Systems*, 1(3):223-241, 1976.<br><br>Presents the decomposition strategy developed for the INGRES query optimizer. |
| | Karel Youssefi and Eugene Wong, “Query Processing In a Relational Database Management System,” *Proceedings of 5th International Conference on Very Large Data Bases*, 409-417, 1979.<br><br>Evaluates a number of different strategies used in the development of the INGRES query optimizer. |
| | Michael Stonebraker, Eugene Wong, and Peter Kreps, “The Design and Implementation of INGRES,” *ACM Transactions on Database Systems*, 1(3):189-222, 1976.<br><br>Presents the general approaches to designing and implementing INGRES, touching only briefly on query optimization. |
| | Michael Stonebraker, “Retrospection on a Database System,” *ACM Transactions on Database Systems*, 5(2):225-240, 1980.<br><br>Makes an honest assessment of the history of the development of INGRES, focusing on the mistakes. Includes a brief analysis of the query optimization approach that was implemented in the product. |

| Topic | Reference |
|---|---|
| Descendants of the System R Optimizer: The DB2 Family | Stephen Brobst and Dominic Sagar, “The New, Fully Loaded, Optimizer,” *DB2 Magazine*, 4(3), 1999. http://www.db2mag.com/db_area/archives/1999/q3/brobst.shtml.<br>Stephen Brobst and Bob Vecchione, “Starburst Grows Bright,” *Database Programming and Design*, 11(3), 1998.<br>Two fairly high-level product overviews of various newer IBM query optimizers. Stephen Brobst is now one of the Teradata database CTOs. |
| | Peter Gassner, Guy Lohman, and K. Bernhard Schiefer, “Query Optimization in the IBM DB2 Family,” *IEEE Data Engineering Bulletin*, 16(4):4-17, 1993.<br>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. |
| | Richard L. Cole, Mark J. Anderson, and Robert J. Bestgen, “Query Processing in the IBM Application System 400,” *IEEE Data Engineering Bulletin*, 16(4):18-27, 1993.<br>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. |
| Basic Principles of Query Optimization | Surajit Chaudhuri, "An Overview of Query Optimization in Relational Systems," *Proceedings of PODS 1998*, 34-43, 1998.<br>Presents a concise summary of the general principles of cost-based query optimization for relational database management systems.<br>C.J. Date, *An Introduction to Database Systems* (8th ed.), Reading, MA: Addison-Wesley, 2004.<br>This textbook presents an excellent overview of optimization in a rigorous, but less formal manner than Ullman (1982).<br>Goetz Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2):73-170, 1993.<br>Extends Jarke and Koch (1984) by adding a great deal of material on optimization in a parallel relational environment.<br>Waqar Hasan, *Optimization of SQL Queries for Parallel Machines*, Berlin: Springer-Verlag, 1996.<br>Also published as *Optimization of SQL Queries for Parallel Machines*, Ph.D. dissertation, Department of Computer Science, Stanford University, 1996.<br>Available as a free Postscript file at http://www-db.stanford.edu/pub/hasan/1995/<br>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. |

| Topic | Reference |
|---|---|
| Basic Principles of Query Optimization (continued) | Matthias Jarke and Jürgen Koch, "Query Optimization in Database Systems," *ACM Computing Surveys*, 16(2):111-152, 1984.<br><br>An excellent overview of the basic principles of query optimization in a relational environment.<br><br>David Maier, *The Theory of Relational Databases*, Rockville, MD: Computer Science Press, 1983.<br><br>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.<br><br>Jeffrey D. Ullman, *Principles of Database Systems* (2nd ed.), Rockville, MD: Computer Science Press, 1982.<br><br>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.<br><br>Clement T. Yu and Weiyi Meng, *Principles of Database Query Processing for Advanced Applications*, San Francisco, CA: Morgan Kaufmann Publishers, 1998.<br><br>An advanced textbook covering numerous special topics in relational and object/relational query processing. The first chapter is an excellent introductory overview of relational query optimization. |
| Statistical Profiles of Data Demographics | Michael V. Mannino, Paicheng Chu, and Thomas Sager, "Statistical Profile Estimation in Database Systems," *ACM Computing Surveys*, 20(3):191-221, 1988.<br><br>A basic, somewhat dated, overview of relational database statistics: how they are gathered and how they are used. |
| Interval Histograms | Yannis E. Ioannidis, “The History of Histograms (abridged),” *Proceedings of VLDB Conference*, 2003.<br><br>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.<br><br>Yannis E. Ioannidis and Stavros Christodoulakis, "Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results," *ACM Transactions on Database Systems*, 18(4):709-448, 1993.<br><br>Demonstrates that high-biased interval histograms are always optimal for estimating join cardinalities for a commonly used form of equijoin.<br><br>Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *Proceedings of SIGMOD 1996*, 294-305, 1996.<br><br>Provides a capsule review of older types of interval histograms used in relational systems and introduces a few new types, including compressed histograms. |

| Topic | Reference |
|---|---|
| High-Biased Interval Histograms | Philip B. Gibbons and Yossi Matias, "Synopsis Data Structures for Massive Data Sets," in: *External Memory Systems: DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, 1998.<br><br>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 the Teradata Database were developed by this group. |
| Propagation of Errors in Join Optimization Because of Bad Statistics | Yannis E. Ioannidis and Stavros Christodoulakis, "On the propagation of errors in the size of join results," *Proceedings of SIGMOD 1991*, 268-277, 1991.<br><br>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. |
| Join Processing | Priti Mishra and Margaret H. Eich, "Join Processing in Relational Databases," *ACM Computing Surveys*, 24(1):63-113, 1992.<br><br>A comprehensive, though slightly dated, review of the standard set of join issues for relational database management. The first two sections of the paper, "Join Operation" and "Implementation of Joins," are the most relevant to the material in this chapter. |
| | Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T.Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan, "Cost-Based Optimization for Magic: Algebra and Implementation," *Proceedings of SIGMOD 1996*, 435-446, 1996.<br><br>The Teradata optimizer actually uses an improvement over Magic Sets. What is relevant here is the complexity estimates for the space required to make join estimates. |
| | Toshihide Ibaraki and Tiko Kameda, "On the Optimal Nesting Order for Computing N-Relational Joins," *ACM Transactions on Database Systems*, 9(3):482-502, 1984.<br><br>Proves that join estimation is NP-complete. |
| | Kiyoshi Ono and Guy M. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization," *Proceedings of 16th International Conference on Very Large Data Bases*, 314-325, 1990.<br><br>Demonstrates for the first time that no particular join enumeration method is best for all queries. |

| Topic | Reference |
|---|---|
| Complexity Theory and Descriptive Complexity | Donald Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd. ed.), Reading, MA: Addison-Wesley, 1997.<br>Edmund Landau, *Foundations of Analysis*, New York: Chelsea Publishing Co., 1960.<br>Michael Sipser, *Introduction to the Theory of Computation* (2nd ed.), Boston, MA: Thomson Course Technology, 2005.<br>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. |
| | Neil Immerman, "Descriptive and Computational Complexity," *Proceedings of Symposia in Applied Mathematics*, 38:75-91, 1989.<br>Available as a PDF download from the following URL: http://www.cs.umass.edu/~immerman/pub/survey.pdf.<br>Neil Immerman, "Descriptive Complexity: a Logician's Approach to Computation," *Notices of the American Mathematical Society*, 42(10):1127-1133, 1995.<br>Available as a PDF download from the following URL: http://www.cs.umass.edu/~immerman/pub/ams_notices.pdf<br>Two survey papers on descriptive complexity by the leading computer scientist in the field. |
| Graph Theory | Béla Bollobás, *Modern Graph Theory*, Berlin: Springer-Verlag, 1998.<br>Gary Chartrand, *Introductory Graph Theory*, Mineola, NY: Dover Publishing Co., 1984.<br>Republication of Boston, MA: Prindle, Weber, and Schmidt, Inc., *Graphs as Mathematical Models*, 1977.<br>Reinhard Diestel, *Graph Theory* (2nd ed.), Berlin: Springer-Verlag, 2000.<br>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.<br>Note that the PDF version of the book cannot be printed.<br>Shimon Even, *Graph Algorithms*, Rockville, MD: Computer Science Press, 1979.<br>Richard J. Trudeau, *Introduction to Graph Theory*, Mineola, NY: Dover Publishing Co., 1994.<br>Corrected and enlarged edition of Kent, OH: The Kent State University Press, *Dots and Lines.*, 1976.<br>An assortment of introductory textbooks on graph theory. The Chartrand and Trudeau books are published as inexpensive paperbound editions. |

# CHAPTER 3 Join Optimizations

This chapter describes several join optimizations, including the following topics:

- Star and snowflake join optimization

  For additional information, see:

  - “Star and Snowflake Join Optimization” on page 201
  - “Miscellaneous Considerations for Star Join Optimization” on page 209
  - “Selecting Indexes for Star Joins” on page 211
- Classes of large table/small table joins

  For additional information, see:

  - “LT/ST-J1 Indexed Joins” on page 207
  - “LT-ST-J2 Unindexed Joins” on page 208
- Reasonable indexed joins (LT/ST-J1a)

  For additional information, see:

  - “Reasonable Indexed Join Plan Without Star Join Optimization” on page 217
  - “Reasonable Indexed Join Plan With Star Join Optimization: Large Table Primary Index Joined to Small Tables” on page 219
  - “Reasonable Indexed Join Plan With Star Join Optimization: Large Table USI Joined to Small Tables” on page 221
  - “Reasonable Indexed Join Plan With Star Join Optimization: Large Table NUSI Joined to Small Tables” on page 223
- Reasonable unindexed joins (LT/ST-J2a)

  For additional information, see:

  - “Reasonable Unindexed Join Without Join Optimization” on page 227
  - “Reasonable Unindexed Join With Join Optimization” on page 228
- Joins Using Join Indexes

  For additional information, see:

  - “Join Indexes” on page 229
  - “Maintaining Join Index During INSERT” on page 237
  - “General Method of Maintaining Join Index During DELETE” on page 235
  - “Optimized Method of Maintaining Join Index During DELETE” on page 236
  - “Maintaining Join Index During INSERT” on page 237
  - “General Method of Maintaining Join Index During UPDATE” on page 239
  - “Optimized Method of Maintaining Join Index During UPDATE” on page 241

The performance-oriented aspects of these topics 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*

# 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.

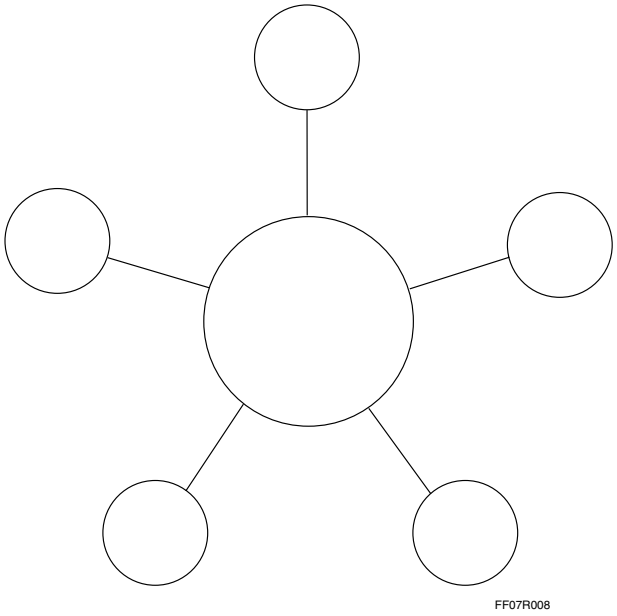
## Definition: Star Join

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:




The concept of star schemas in physical database design is described in *Database Design*.

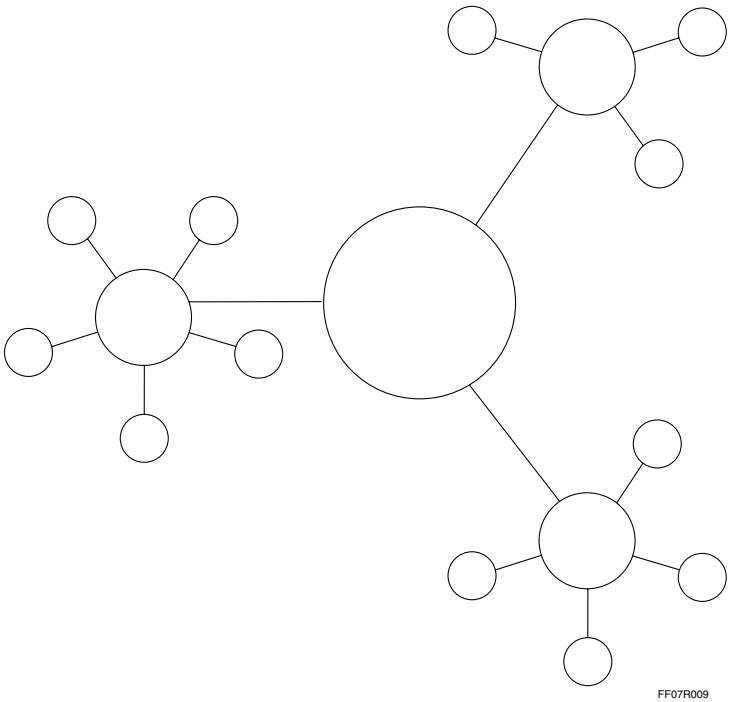
## Definition: Snowflake Join

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

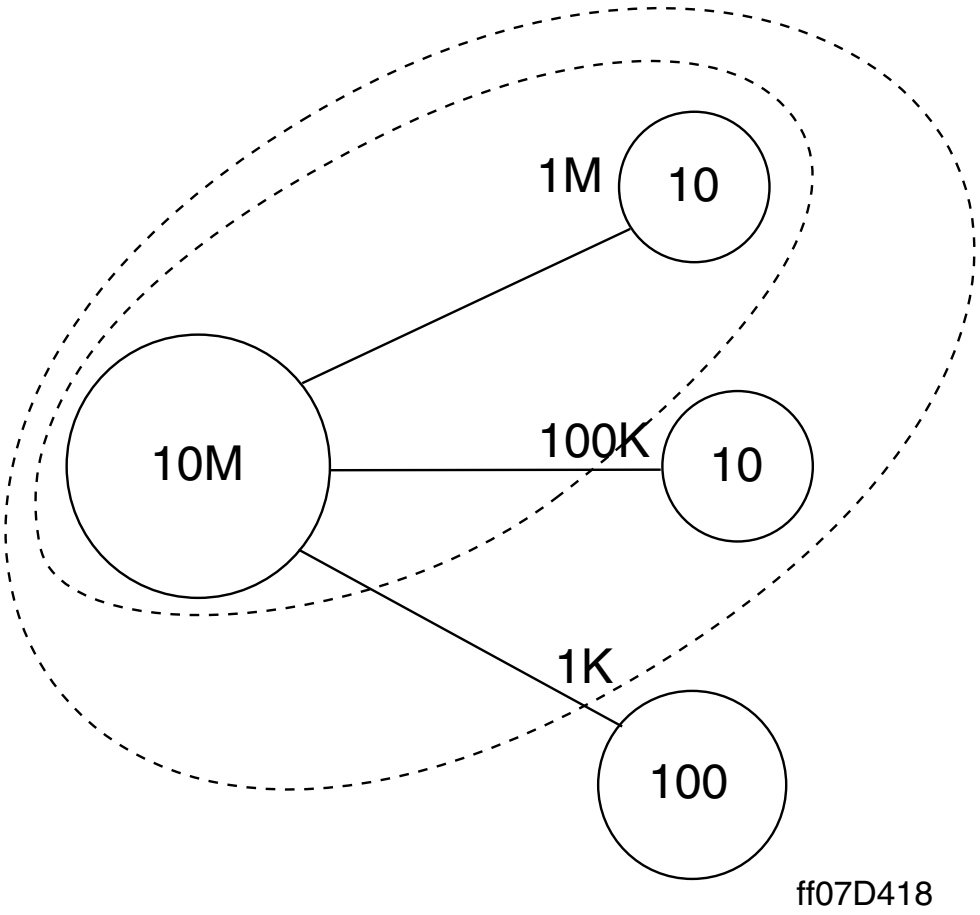
Optimizer usually generates a join plan in which a small table (or the join result of small tables) is joined directly with the large table.

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 step 6 on page 137 in "Evaluating Join Orders" on page 135).

## Example Star Join Optimization

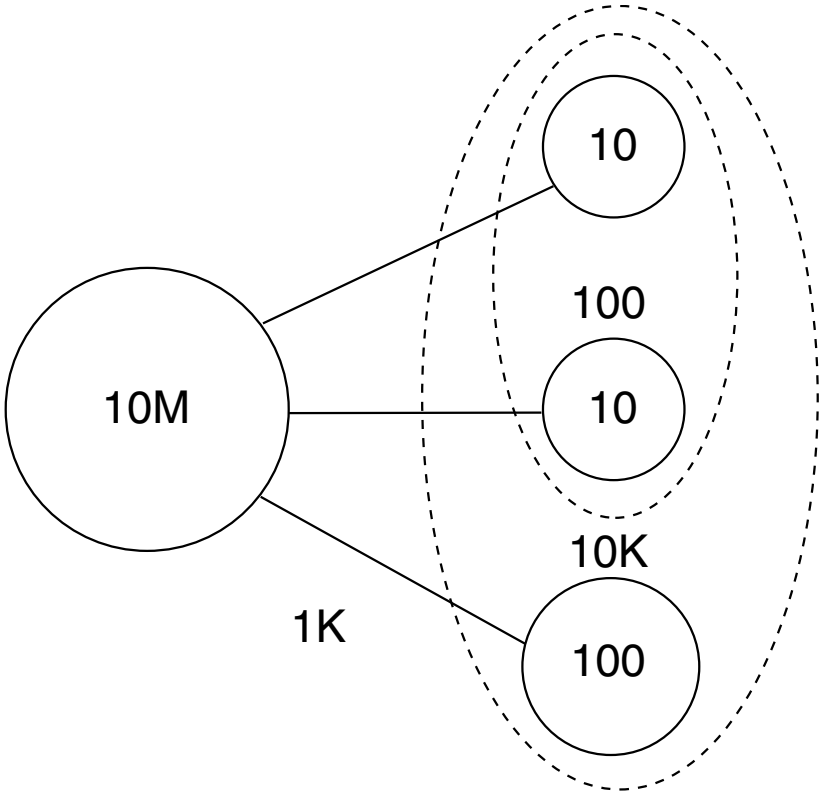
The following graphic illustrates a nonoptimal star join plan of four tables. The relative cardinalities of the tables are given as integers within each table (represented in each case by a circle). The relative cost of each join is given as an integer number on the line connecting the joined relations.

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 $1 \times 10^3$.



The next graphic presents an optimized join plan for the same set of tables. This plan uses a compromise join of all the unconnected (small) tables prior to making the join to the large table.

The first join has a relative cost of 1 x $10^2$, the second a cost of 1 x $10^4$, and the final join between the large table and its joined small tables relation has a cost of 1 x $10^3$.




ff07D419

$$\text{Cost of nonoptimized star join} = 1\times10^6 + 1\times10^5 + 1\times10^3 = 1\ 101\ 000$$

$$\text{Cost of optimized star join} = 1\times10^2 + 1\times10^4 + 1\times10^3 = 11\ 100$$

The results indicate that for this example, the optimized star join plan is two orders of magnitude cheaper than the nonoptimized star join plan for the same four tables.

## Star Join Categories

For purposes of the following descriptions, star joins are referred to as LT/ST joins. LT/ST joins always fall into one of the following categories:

- Indexed Joins, LT/ST-J1

  For more detailed information on LT/ST-J1 joins, see "LT/ST-J1 Indexed Joins" on page 207.
- Unindexed Joins, LT/ST-J2

  For more detailed information on LT/ST-J2 joins, see "LT-ST-J2 Unindexed Joins" on page 208.

# 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:

| IF LT/ST joins are … | THEN the primary index should be created using the … |
|---|---|
| used with the same frequency | star join that results in the most number of rows selected.<br>This leaves the star joins that select fewer rows for the NUSIs and USIs. |
| not used with the same frequency | collection of join columns associated with the most often used star join. |

For example, if the star join between Widgets and (period, state, country) is used more often than the star join between Widgets and (color, shape, size), the primary index should be created on (period, state, country).

However, if the former join selects far fewer number of rows than the latter join, it may be better to associate the primary index with the latter join (on columns color, shape, and size).

## Performance Modeling: Optimizing All Join Queries

To optimize the entire mix of join queries, you should design a more complete performance model for your database.

For example, a user might have a relatively short star join query that is used more frequently than an extremely long query.

In such cases, it might be better to select the *primary* index favoring the long query, even though the guidelines indicate otherwise. This is because the benefit of the long query may be very great compared to the cost of the short query, and the combination of joins results in a net gain in performance.

Not all join columns of the small tables must join with the large table index in an LT/ST-J1a (Reasonable Indexed) star join.

## Using a Common Set of Join Columns in the Primary Index

If more than one combination of the large table columns is used in different star joins, and if the combinations are overlapping, then the primary index should consist of the *common set* of these combinations (if the set is qualified for the primary index).

This has two advantages.

- Fewer indexes are required.
- More than one star join can share the same primary index.

For example, assume that the following conditions are true:

- The collection of columns (color, shape, size) of the large table Widgets is joined with the small tables (color, shape, and size) of a star join
- The collection of columns (shape, size, and period) is joined with the small tables (shape, size, and period) of another star join

In this case, the primary index of Widgets should consist of the columns (shape, size) if the set is qualified for the primary index.

# Star Join Examples

## Introduction

The sections that follow provide examples of these join expression types.

- LT/ST-J1a (Reasonable Indexed)
- LT/ST-J2a (Reasonable Unindexed)
- Nested join

## How the Examples Are Structured

For each type of query, two summaries of the join plans and estimated execution times are provided—one with and the other without star join optimization.

To be consistent with the EXPLAIN output, if the input table of a join is the result of a previous join, the cost of preparing the input table for the join is included in the cost of performing the previous join. Otherwise, the preparation cost is included into the cost of performing a join.

The total estimated cost for each query is taken directly from the EXPLAIN outputs, which take into account the parallelism of steps.

Costs are relative, and vary depending on the number of AMPs in the configuration. The example costs given are for a system with two AMPs.

The estimated percentage of performance improvement is provided for each example. Remember that these percentages are achieved only when the same join examples are performed under the identical conditions.

Other queries may achieve more or less performance improvement, depending on the join conditions and table statistics, but the general trends are consistently in the same direction.

The following table definitions are used in the examples:

## Small Table Definitions

| Table name | Columns | Primary index |
|---|---|---|
| Color | code, description | description |
| Size | code, description | description |
| Options | code, description | description |

## Large Table Definition

| Table name | Columns | Cardinality |
|---|---|---|
| Widgets | color, size, options, units, period | 1 000 000 rows |

## Star Join Example Types

Examples are provided for the following Large Table/Small Table join expression types:

# Cardinality and Uniqueness Statistics for the Reasonable Indexed Join Examples

## Small Table Cardinality Statistics

For all LT/ST-J1a (Reasonable Indexed Joins) examples, the following row information is given for the small tables:

| Attribute | Cardinality |
|---|---|
| color | 2 |
| size | 10 |
| options | 10 |

## Large Table Uniqueness Statistics

The following statistics information is given for the large table:

| Column Name | Number of Unique Values |
|---|---|
| color | 10 |
| size | 100 |
| options | 1000 |

## Test Query

These examples explain the join plans and estimated time for execution of the following query, when different types of indexes (primary index, unique secondary index, nonunique secondary index) are created on the various join columns (color, size, and options) of the large table.

```
SELECT …
WHERE widgets.color = color.code
AND   widgets.size = size.code
AND   widgets.options = options.code;
```

# Reasonable Indexed Join Plan Without Star Join Optimization

## Unoptimized Join Plan

Without join optimization, the following join plan is generated independently of the type of index created on the collection of join columns (color, size, and options) of the large table:

| Operation | Joined Tables | Total Processing Time (seconds) |
|---|---|---|
| Spool 4: Product Join | duped options, direct size | 2.67 |
| Spool 5: Product Join | duped color, direct widgets | 6 660.00[1] |
| Spool 1: Merge Join | duped 4, local 5 | 7.43 |

1. 1 hour, 51 minutes.

## Completion Time

Note that the total estimated completion time, including time for two product joins and a merge join, is 1 hour 52 minutes.

## EXPLAIN Output for Unoptimized Join Plan

The following EXPLAIN output is generated without star join optimization, independently of the type of index created on the collection of join columns (color, size, and options) of the large table:

```
Explanation
-----------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we execute the following steps in parallel.
   a) We do an all-AMPs RETRIEVE step from TEST.Color by way of
      an all-rows scan with no residual conditions into Spool 2, which
      is duplicated on all AMPs. The size of Spool 2 is estimated to be
      4 rows. The estimated time for this step is 0.08 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Options by way of an
      all-rows scan with no residual conditions into Spool 3, which is
      duplicated on all AMPs. The size of Spool 3 is estimated to be 20
      rows. The estimated time for this step is 0.24 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from TEST.Size by way of an all-rows
      scan with no residual conditions, which is joined to Spool 3
      (Last Use). TEST.Size and Spool 3 are joined using a product
      join. The result goes into Spool 4, which is duplicated on all
      AMPs. Then we do a SORT to order Spool 4 by row hash. The size
      of Spool 4 is estimated to be 200 rows. The estimated time for
      this step is 2.43 seconds.
   b) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of
      an all-rows scan, which is joined to TEST.Widgets. Spool 2 and
      TEST.Widgets are joined using a product join, with a join
      condition of ("TEST.Widgets.color = Spool_2.code"). The result
      goes into Spool 5, which is built locally on the AMPs. Then we do
      a SORT to order Spool 5 by row hash. The size of Spool 5 is
      estimated to be 200,000 rows. The estimated time for this step is
      1 hour and 51 minutes.
```

```
4) We do an all-AMPs JOIN step from Spool 4 (Last Use) by way of an
   all-rows scan, which is joined to Spool 5 (Last Use). Spool 4 and
   Spool 5 are joined using a merge join, with a join condition of
   ("(Spool_5.size=Spool_4.code)AND(Spool_5.options=Spool_4.code)").
   The result goes into Spool 1, which is built locally on the AMPs. The
   size of Spool 1 is estimated to be 200 rows. The estimated time for
   this step is 7.43 seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 1 hour and 52 seconds.
```

# Reasonable Indexed Join Plan With Star Join Optimization: Large Table Primary Index Joined to Small Tables

## Optimized Join Plan

With join optimization, the following join plan is generated when the collection of join columns (color, size, and options) makes up the primary index of the large table:

| Operation | Joined Tables | Total Processing Time (seconds) |
|---|---|---|
| Spool 3: Product Join | duped color, direct options | 0.31 |
| Spool 5: Product Join | duped size, direct 3 | 1.62 |
| Spool 1: Merge Join | hashed 5, direct widgets | 4.09 |

## Completion Time

Total estimated execution time is 5.80 seconds.

The estimated performance improvement factor is 1158.

## EXPLAIN Output for Optimized Join Plan

The EXPLAIN output for this optimized join plan is as follows:

```
Explanation
---------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we    lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we do an all-AMPs RETRIEVE step from TEST.Color by way of an
   all-rows scan with no residual conditions into Spool 2, which is
   duplicated on all AMPs. The size of Spool 2 is estimated to be 4
   rows. The estimated time for this step is 0.08 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
      all-rows scan, which is joined to TEST.Options. Spool 2 and
      TEST.Options are joined using a product join. The result goes
      into Spool 3, which is built locally on the AMPs. The size of
      Spool 3 is estimated to be 20 rows. The estimated time for this
      step is 0.23 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Size by way of an
      all-rows scan with no residual conditions into Spool 4, which is
      duplicated on all AMPs. The size of Spool 4 is estimated to be 20
      rows. The estimated time for this step is 0.24 seconds.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a product join. The result goes into
   Spool 5, which is redistributed by hash code to all AMPs. Then we do
   a SORT to order Spool 5 by row hash. The size of Spool 5 is
   estimated to be 200 rows. The estimated time for this step is
   1.38 seconds.
5) We do an all-AMPs JOIN step from TEST.Widgets by way of an all-rows
   scan with no residual conditions, which is joined to Spool 5 (Last
   Use). TEST.Widgets and Spool 5 are joined using a merge join, with a
   join condition of ("(TEST.Widgets.size = Spool_5.code) AND
   ((TEST.Widgets.color = Spool_5.code) AND (TEST.Widgets.options =
   Spool_5.code ))"). The result goes into Spool 1, which is built
   locally on the AMPs. The size of Spool 1 is estimated to be 200 rows.
   The estimated time for this step is 4.09 seconds.
```

```
6) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 5.80 seconds.
```

# Reasonable Indexed Join Plan With Star Join Optimization: Large Table USI Joined to Small Tables

## Optimized Join Plan

With join optimization, the following join plan is generated when the collection of join columns (color, size, and options) makes up a unique secondary index of the large table:

| Operation | Joined Tables | Total Processing Time (seconds) |
|---|---|---|
| Spool 3: Product Join | duped color, direct options | 0.31 |
| Spool 5: Product Join | duped size, direct 3 | 1.62 |
| Spool 6: Nested Join | hashed 5, index widgets | 2.71 |
| Spool 1: rowID Join | hashed 6, index widgets | 5.65 |

## Completion Time

The total estimated time is 10.07 seconds.

The estimated performance improvement factor is 667.

## EXPLAIN Output for Optimized Join Plan

The EXPLAIN output for this optimized join plan is as follows:

```
Explanation
---------------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we do an all-AMPs RETRIEVE step from TEST.Color by way of an
   all-rows scan with no residual conditions into Spool 2, which is
   duplicated on all AMPs. The size of Spool 2 is estimated to be 4
   rows. The estimated time for this step is 0.08 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
      all-rows scan, which is joined to TEST.Options. Spool 2 and
      TEST.Options are joined using a product join. The result goes
      into Spool 3, which is built locally on the AMPs. The size of
      Spool 3 is estimated to be 20 rows. The estimated time for this
      step is 0.23 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Size by way of an
      all-rows scan with no residual conditions into Spool 4, which is
      duplicated on all AMPs. The size of Spool 4 is estimated to be 20
      rows. The estimated time for this step is 0.24 seconds.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a product join. The result goes into Spool
   5, which is redistributed by hash code to all AMPs. Then we do a
   SORT to order Spool 5 by row hash. The size of Spool 5 is estimated
   to be 200 rows. The estimated time for this step is 1.38 seconds.
5) We do a all-AMP JOIN step from Spool 5 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets by way of unique
   index # 4 extracting row ids only. Spool 5 and TEST.Widgets are
   joined using a nested join. The result goes into Spool 6, which is
   redistributed by hash code to all AMPs. Then we do a SORT to order
   Spool 6 by row hash. The size of Spool 6 is estimated to be 200
   rows. The estimated time for this step is 2.71 seconds.
```

```
6) We do an all-AMPs JOIN step from Spool 6 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets. Spool 6 and
   TEST.Widgets are joined using a row id join. The result goes into
   Spool 1, which is built locally on the AMPs. The size of Spool 1 is
   estimated to be 200 rows. The estimated time for this step is 5.65
   seconds.
7) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 10.07 seconds.
```

# Reasonable Indexed Join Plan With Star Join Optimization: Large Table NUSI Joined to Small Tables

## Optimized Join Plan

With join optimization, the following join plan is generated when the collection of join columns (color, size, and options) makes up a nonunique secondary index of the large table:

| Operation | Joined Tables | Total Processing Time (seconds) |
|---|---|---|
| Spool 3: Product Join | duped color, direct options | 0.31 |
| Spool 5: Product Join | duped size, direct 3 | 4.43 |
| Spool 1: Nested Join | duped 5, index widgets | 22.73 |

## Completion Time

The total estimated execution time is 27.26 seconds.

The estimated performance improvement factor is 246.

## EXPLAIN Output for Optimized Join Plan

The EXPLAIN output for this optimized join plan is as follows:

```
Explanation
---------------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we do an all-AMPs RETRIEVE step from TEST.Color by way of an
   all-rows scan with no residual conditions into Spool 2, which is
   duplicated on all AMPs. The size of Spool 2 is estimated to be 4
   rows. The estimated time for this step is 0.08 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
      all-rows scan, which is joined to TEST.Options. Spool 2 and
      TEST.Options are joined using a product join. The result goes
      into Spool 3, which is built locally on the AMPs. The size of
      Spool 3 is estimated to be 20 rows. The estimated time for this
      step is 0.23 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Size by way of an
      all-rows scan with no residual conditions into Spool 4, which is
      duplicated on all AMPs. The size of Spool 4 is estimated to be 20
      rows. The estimated time for this step is 0.24 seconds.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a product join. The result goes into Spool
   5, which is duplicated on all AMPs. The size of Spool 5 is estimated
   to be 400 rows. The estimated time for this step is 4.19 seconds.
5) We do an all-AMPs JOIN step from Spool 5 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets by way of index # 4.
   Spool 5 and TEST.Widgets are joined using a nested join. The result
   goes into Spool 1, which is built locally on the AMPs. The size of
   Spool 1 is estimated to be 200 rows. The estimated time for this
   step is 22.73 seconds.
6) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 27.26 seconds.
```

# Join Plan With Star Join Optimization: Large Table Subquery Join

## Example Query

The following query is used for this example:

```
SELECT ...
WHERE Widgets.color=COLOR.code
AND   Widgets.size=SIZE.code
AND   Widgets.options IN (SELECT OPTIONS.code);
```

## Optimized Join Plan

With join optimization, the following join plan is generated when the collection of join columns (color, size, and options) makes up a nonunique secondary index of the large table:

| Operation | Joined Tables | Total Processing Time (seconds) |
|---|---|---|
| Spool 4: Product Join | duped color, direct size | 0.31 |
| Spool 6: Product Join | local 4, duped options | 4.46 |
| Spool 1: Nested Join | duped 6, index widgets | 22.73 |

## Completion Time

The total estimated completion time is 27.40 seconds.

The estimated performance improvement factor is 245.

## EXPLAIN Output for Optimized Join Plan

The EXPLAIN output for this optimized join plan is as follows:

```
Explanation
-----------------------------------------------------------------------
1) First, we lock TEST.Options for read, we lock TEST.Color for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we execute the following steps in parallel.
   a) We do an all-AMPs RETRIEVE step from TEST.Options by way of
      an all-rows scan with no residual conditions into Spool 2, which
      is redistributed by hash code to all AMPs. Then we do a SORT to
      order Spool 2 by the sort key in spool field1 eliminating
      duplicate rows. The size of Spool 2 is estimated to be 10 rows.
      The estimated time for this step is 0.19 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Color by way of an
      all-rows scan with no residual conditions into Spool 3, which is
      duplicated on all AMPs. The size of Spool 3 is estimated to be 4
      rows. The estimated time for this step is 0.08 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
      all-rows scan, which is joined to TEST.Size. Spool 3 and
      TEST.Size are joined using a product join. The result goes into
      Spool 4, which is built locally on the AMPs. The size of Spool 4
      is estimated to be 20 rows. The estimated time for this step is
      0.23 seconds.
   b) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of
      an all-rows scan into Spool 5, which is duplicated on all AMPs.
      The size of Spool 5 is estimated to be 20 rows. The estimated time
      for this step is 0.27 seconds.
```

```
4) We do an all-AMPs JOIN step from Spool 4 (Last Use) by way of an
   all-rows scan, which is joined to Spool 5 (Last Use). Spool 4 and
   Spool 5 are joined using a product join. The result goes into Spool
   6, which is duplicated on all AMPs. The size of Spool 6 is estimated
   to be 400 rows. The estimated time for this step is 4.19 seconds.
5) We do an all-AMPs JOIN step from Spool 6 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets by way of index # 4.
   Spool 6 and TEST.Widgets are joined using a nested join. The result
   goes into Spool 1, which is built locally on the AMPs. The size of
   Spool 1 is estimated to be 200 rows. The estimated time for this
   step is 22.73 seconds.
6) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 27.40 seconds.
```

# Cardinality and Uniqueness Statistics for the Reasonable Unindexed Join Examples

## Small Table Cardinality Statistics

For LT/ST-J2a (Reasonable Unindexed) join example, the following row information is given for the small tables:

| Attribute | Cardinality (rows) |
|---|---|
| color | 3 |
| size | 10 |
| options | 10 |

## Large Table Uniqueness Statistics

The following statistics information is given for the large table:

| Column Name | Number of Unique Values |
|---|---|
| color | 6 |
| size | 30 |
| options | 300 |

No index is created on the collection of join columns (color, size, and options) of the large table.

## Test Query

The following query is used for the LT/ST-J2a (Reasonable Unindexed) join in this example:

```
SELECT *
FROM widget, color, size, options
WHERE widgets.color = color.code
AND   widgets.size = size.code
AND   widgets.options = options.code
AND   size.description = options.description;
```

# Reasonable Unindexed Join Without Join Optimization

## Unoptimized Join Plan

Without star join optimization, the following join plan is generated:

| Operation | Joined Tables | Total Processing Time (seconds) |
|---|---|---|
| Spool 3: Merge Join | direct options, direct size | 0.46 |
| Spool 4: Product Join | duped color, direct widgets | 12 491.00[1] |
| Spool 1: Merge Join | duped 3, local 4 | 21.05 |

1. 3 hours, 28 minutes, 11 seconds.

## Completion Time

The total estimated completion time is 3 hours 28 minutes.

## EXPLAIN Output for Unoptimized Join Plan

The following EXPLAIN output is generated without LT/ST optimization:

```
Explanation
-----------------------------------------------------------------------
1) First, we lock TEST.Color for read, we lock TEST.Options for read,
   we lock TEST.Size for read, and we lock TEST.Widgets for read.
2) Next, we execute the following steps in parallel.
   a) We do an all-AMPs RETRIEVE step from TEST.Color by way of an
      all-rows scan with no residual conditions into Spool 2, which is
      duplicated on all AMPs. The size of Spool 2 is estimated to be 6
      rows. The estimated time for this step is 0.11 seconds.
   b) We do an all-AMPs JOIN step from TEST.Options by way of an
      all-rows scan with no residual conditions, which is joined to
      TEST.Size. TEST.Options and TEST.Size are joined using a merge
      join, with a join condition of ("TEST.Widgets.description =
      TEST.Options.description"). The result goes into Spool 3, which
      is duplicated on all AMPs. Then we do a SORT to order Spool 3 by
      row hash. The size of Spool 3 is estimated to be 20 rows. The
      estimated time for this step is 0.46 seconds.
3) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
   all-rows scan, which is joined to TEST.Widgets. Spool 2 and
   TEST.Widgets are joined using a product join, with a join condition
   of ("TEST.Widgets.color = Spool_2.code"). The result goes into
   Spool 4, which is built locally on the AMPs. Then we do a SORT to
   order Spool 4 by row hash. The size of Spool 4 is estimated to be
   500,000 rows. The estimated time for this step is 3 hours and 28
   minutes.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a merge join, with a join condition of
   ("(Spool_4.options = Spool_3.code) AND (Spool_4.size =
   Spool_3.code)"). The result goes into Spool 1, which is built
   locally on the AMPs. The size of Spool 1 is estimated to be 555
   rows. The estimated time for this step is 21.05 seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 3 hours and 28 minutes.
```

# Reasonable Unindexed Join With Join Optimization

## Optimized Join Plan

With star join optimization, the following join plan is generated:

| Operation | Joined Tables | Total Processing Time (seconds) |
|---|---|---|
| Spool 2: Merge Join | direct options, direct size | 0.44 |
| Spool 3: Product Join | direct color, duped 2 | 1.24 |
| Spool 1: Merge Join | local widgets, duped 3 | 7 761.00[1] |

1. 2 hours, 9 minutes, 21 seconds.

## Completion Time

The total estimated completion time is 2 hours 9 minutes.

The estimated performance improvement factor is 1.6.

## EXPLAIN Output for Optimized Join Plan

The following EXPLAIN output is generated:

```
Explanation
-----------------------------------------------------------------------
1) First, we lock TEST.Options for read, we lock TEST.Size for read, we
   lock TEST.Color for read, and we lock TEST.Widgets for read.
2) Next, we do an all-AMPs JOIN step from TEST.Options by way of an
   all-rows scan with no residual conditions, which is joined to
   TEST.Size. TEST.Options and TEST.Size are joined using a merge join,
   with a join condition of ("TEST.Size.description =
   TEST.Options.description"). The result goes into Spool 2, which is
   duplicated on all AMPs. The size of Spool 2 is estimated to be 20
   rows. The estimated time for this step is 0.44 seconds.
3) We execute the following steps in parallel.
   a) We do an all-AMPs JOIN step from TEST.Color by way of an
      all-rows scan with no residual conditions, which is joined to
      Spool 2 (Last Use). TEST.Color and Spool 2 are joined using a
      product join. The result goes into Spool 3, which is duplicated
      on all AMPs. Then we do a SORT to order Spool 3 by row hash. The
      size of Spool 3 is estimated to be 60 rows. The estimated time for
      this step is 1.24 seconds.
   b) We do an all-AMPs RETRIEVE step from TEST.Widgets by way of an
      all-rows scan with no residual conditions into Spool 4, which is
      built locally on the AMPs. Then we do a SORT to order Spool 4 by
      row hash. The size of Spool 4 is estimated to be 1,000,000 rows.
      The estimated time for this step is 2 hours and 9 minutes.
4) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an
   all-rows scan, which is joined to Spool 4 (Last Use). Spool 3 and
   Spool 4 are joined using a merge join, with a join condition of (
   "(Spool_4.color = Spool_3.code) AND ((Spool_4.size = Spool_3.code
   AND (Spool_4.options = Spool_3.code ))"). The result goes into Spool
   1, which is built locally on the AMPs. The size of Spool 1 is
   estimated to be 556 rows. The estimated time for this step is 21.94
   seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1. The total estimated time is 2 hours and 9 minutes.
```

# Join Indexes

## Introduction

The following sections demonstrate the performance optimization achieved on table selects, deletes, inserts, and updates resulting from the use of join indexes.

For additional information about join indexes, see SQL Reference: Fundamentals and *SQL Reference: Data Definition Statements*.

## Example 1: Simple Join Query

The following is an example of a simple join query:

```
EXPLAIN
SELECT o_orderdate, o_custkey, 1_partkey, 1_quantity, 1_extendedprice
FROM Lineitem, Ordertbl
WHERE 1_orderkey = o_orderkey;

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

## Example 2: Search on Join Indexed Rows

The following is an example of a search condition on the join indexed rows:

```
EXPLAIN
SELECT o_orderdate, o_custkey, 1_partkey, 1_quantity,1_extendedprice
FROM Lineitem, Ordertbl
WHERE 1_orderkey = o_orderkey
AND o_orderdate > '1997-11-01';

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

## Example 3: Aggregation Against Join-Indexed Rows

The following is an example of an aggregation against the Join-Indexed rows:

```
EXPLAIN
SELECT 1_partkey, AVG(1_quantity), AVG(1_extendedprice)
FROM Lineitem, Ordertbl
WHERE 1_ordkey=o_orderkey
AND o_orderdate > '1997-11-01'
GROUP BY 1_partkey;

Explanation

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

## Example 4: Join Indexed Rows Used to Join With Another Base Table

The following is an example of a join indexed rows used to join with another base table:

```
EXPLAIN
SELECT o_orderdate, c_name, c_phone, 1_partkey,1_quantity,
       1_extendedprice
FROM Lineitem, Ordertbl, Customer
WHERE 1_orderkey = o_orderkey
AND o_custkey = c_custkey;

Explanation

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

## Example 5: Join Indexed Row Used to Resolve Single Table Query

The following is an example of a join index rows used to resolve single table query:

```
EXPLAIN
SELECT 1_orderkey, 1_partkey, 1_quantity, 1_extendedprice
FROM Lineitem
WHERE 1_partkey = 1001;

Explanation

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

## Example 6: Creating and Using Secondary Index on Top of Join Index

The following is an example creating and using secondary index on top of the join index:

```
CREATE INDEX shipidx(1_shipdate) ON OrderJoinLine;

***Index has been created.
***Total elapsed time was 5 seconds.
EXPLAIN
SELECT o_orderdate, o_custkey, 1_partkey, 1_quantity,
       1_extendedprice
FROM Lineitem, Ordertbl
WHERE 1_ordkey=o_orderkey
AND l_shipdate='1997-09-18';

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

## Example 7: Using Join Index Defined With Multiway Join Result

The following is an example defining and using a join index defined with a multiway join result:

```
CREATE JOIN INDEX CustOrderJoinLine
AS SELECT (1_orderkey, o_orderdate, c_nationkey,  o_totalprice),
          (1_partkey, 1_quantity, 1_extendedprice,  1_shipdate)
FROM (Lineitem
LEFT OUTER JOIN Ordertbl ON 1_orderkey=o_orderkey)
INNER JOIN Customer ON o_custkey=c_custkey
PRIMARY INDEX (1_orderkey);

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

EXPLAIN
SELECT (1_orderkey, o_orderdate, c_nationkey, o_totalprice),
       (1_partkey, 1_quantity, 1_extendedprice, 1_shipdate)
FROM Lineitem, Ordertbl, Customer
WHERE 1_orderkey = o_custkey
AND o_custkey = c_custkey
AND c_nationkey = 10;

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

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

# Maintenance of Join Index for DELETE, INSERT, and UPDATE

## Introduction

As with other indexes (for example, secondary indexes), join indexes are automatically maintained by the system when DELETE, INSERT, or UPDATE statements are issued against the underlying base tables of a join index.

## Overhead Costs

When considering the use of join indexes, carefully analyze the overhead cost associated with maintaining them during updates and weigh these costs against the benefits to query performance.

Join indexes are maintained by generating additional AMP steps in the execution plan.

The general case method involves first reproducing the affected portion of the join index rows. This is accomplished by re-executing the join query, as defined in the join index, using only those base table rows that are relevant to the update at hand.

The entire join index result is not reproduced for each update statement.

| FOR this category of database operation ... | Join indexes are maintained in this way ... |
|---|---|
| DELETE | The corresponding rows in the join index are located and then deleted with a Merge Delete step. |
| INSERT | Newly formed join result rows are added to the join index with a Merge step. |
| UPDATE | The necessary modifications are performed and the corresponding rows in the join index are then replaced by first deleting the old rows (with a Merge Delete) and then inserting the new rows (with a Merge).<br>Join indexes defined with outer joins usually require additional steps to maintain unmatched rows. |

As with secondary indexes, updates can cause a physical join index row to split into multiple rows. Each newly formed row has the same fixed field value but contains a different list of repeated field values.

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:

```
CREATE JOIN INDEX OrderJoinLine AS
SELECT (l_orderkey, o_orderdate, o_custkey, o_totalprice),
       (l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM Lineitem
LEFT OUTER JOIN Ordertbl ON l_orderkey=o_orderkey
ORDER BY o_orderdate
PRIMARY INDEX (l_orderkey);
```

# General Method of Maintaining Join Index During DELETE

The following is an example of a general case method for maintaining join index during a DELETE.

Note the following items in the EXPLAIN output:

| This step ... | Does this ... |
|---|---|
| 5 | Reproduces the affected portion of the join index rows. |
| 6.1 | Deletes the corresponding rows from the join index. |
| 8 | Inserts new non-matching outer join rows into the join index. |

```
   EXPLAIN
   DELETE FROM Ordertbl
   WHERE o_custkey = 1001;

    *** Help information returned. 37 rows.
    *** Total elapsed time was 2 seconds.

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for write on a
     RowHash to prevent global deadlock for LOUISB.OrderJoinLine.
  2) Next, we lock a distinct LOUISB."pseudo table" for write on a
     RowHash to prevent global deadlock for LOUISB.Ordertbl.
  3) We lock a distinct LOUISB."pseudo table" for read on a RowHash to
     prevent global deadlock for LOUISB.Lineitem.
  4) We lock LOUISB.OrderJoinLine for write, we lock LOUISB.Ordertbl
     for write, and we lock LOUISB.Lineitem for read.
  5) We do an all-AMPs JOIN step from LOUISB.Ordertbl by way of a
     RowHash match scan with a condition of (
     "LOUISB.Ordertbl.o_custkey = 1001"), which is joined to
     LOUISB.Lineitem.  LOUISB.Ordertbl and LOUISB.Lineitem are joined
     using a merge join, with a join condition of (
     "LOUISB.Lineitem.l_orderkey = LOUISB.Ordertbl.o_orderkey").  The
     input tables LOUISB.Ordertbl and LOUISB.Lineitem will not be
     cached in memory, but they are eligible for synchronized scanning.
     The result goes into Spool 1, which is built locally on the AMPs.
     Then we do a SORT to order Spool 1 by row hash.  The result spool
     file will not be cached in memory.  The size of Spool 1 is
     estimated to be 100,000 rows.  The estimated time for this step is
     6 minutes and 5 seconds.
  6) We execute the following steps in parallel.
       1) We do a MERGE DELETE to LOUISB.OrderJoinLine from Spool 1.
       2) We do an all-AMPs DELETE from LOUISB.Ordertbl by way of an
          all-rows scan with a condition of
          ("LOUISB.Ordertbl.o_custkey = 1001").
  7) We do an all-AMPs RETRIEVE step from Spool 1 (Last Use) by way of
     an all-rows scan into Spool 2, which is redistributed by hash code
     to all AMPs.  Then we do a SORT to order Spool 2 by join index.
     The result spool file will not be cached in memory.  The size of
     Spool 2 is estimated to be 100,000 rows.  The estimated time for
     this step is 35 minutes and 59 seconds.
  8) We do a MERGE into LOUISB.OrderJoinLine from Spool 2 (Last Use).
  9) We spoil the parser's dictionary cache for the table.
 10) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

# Optimized Method of Maintaining Join Index During DELETE

## Direct Access Index Updates

Optimizations have been made for update statements that allow the affected join index rows to be located via direct access. For example, if a DELETE specifies a search condition on the primary or secondary of a join index, the affected join index rows are not reproduced. Instead, the join index may be directly searched for the qualifying rows and modified accordingly.

## Preconditions for Optimization

To use this optimized method (that is, the direct update approach), the following conditions must be present:

- A primary or secondary access path to the join index.
- If *join_index_field_2* is defined, no modifications to *join_index_field_1* columns.
- No modifications to the join condition columns appearing in the join index definition.
- No modifications to the primary index columns of the join index.

## Example

The following is an example of an optimized method for maintaining join index during a DELETE:

```
   EXPLAIN
   DELETE FROM LineItem
   WHERE l_orderkey = 10;

    *** Help information returned. 11 rows.
    *** Total elapsed time was 2 seconds.

Explanation
-----------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP DELETE from join index table
          LOUISB.OrderJoinLine by way of the primary index
          "LOUISB.OrderJoinLine.l_orderkey = 10" with a residual
          condition of ("LOUISB.OrderJoinLine.l_orderkey = 10").
       2) We do a single-AMP DELETE from LOUISB.LineItem by way of the
          primary index "LOUISB.LineItem.l_orderkey = 10" with no
          residual conditions.
  2) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

# Maintaining Join Index During INSERT

The following is an example of maintaining join index during an INSERT.

Note the following items in the EXPLAIN output:

| This step ... | Does this ... |
|---|---|
| 9 | Produces the new join result rows for the join index. |
| 11 | Deletes any formerly unmatched outer join rows in the join index. |
| 12 | Inserts the new join result rows into the join index. |

```
   EXPLAIN
   INSERT Ordertbl
   SELECT *
   FROM NewOrders;

    *** Help information returned. 46 rows.
    *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     RowHash to prevent global deadlock for LOUISB.NewOrders.
  2) Next, we lock a distinct LOUISB."pseudo table" for write on a
     RowHash to prevent global deadlock for LOUISB.OrderJoinLine.
  3) We lock a distinct LOUISB."pseudo table" for write on a RowHash to
     prevent global deadlock for LOUISB.Ordertbl.
  4) We lock a distinct LOUISB."pseudo table" for read on a RowHash to
     prevent global deadlock for LOUISB.Lineitem.
  5) We lock LOUISB.NewOrders for read, we lock LOUISB.OrderJoinLine
     for write, we lock LOUISB.Ordertbl for write, and we lock
     LOUISB.Lineitem for read.
  6) We do an all-AMPs RETRIEVE step from LOUISB.NewOrders by way of an
     all-rows scan with no residual conditions into Spool 1, which is
     built locally on the AMPs.  Then we do a SORT to order Spool 1 by
     row hash.  The input table will not be cached in memory, but it is
     eligible for synchronized scanning.  The result spool file will
     not be cached in memory.  The size of Spool 1 is estimated to be 2
     rows.  The estimated time for this step is 0.04 seconds.
  7) We do a MERGE into LOUISB.Ordertbl from Spool 1.
  8) We do an all-AMPs RETRIEVE step from Spool 1 (Last Use) by way of
     an all-rows scan into Spool 3, which is redistributed by hash code
     to all AMPs.  Then we do a SORT to order Spool 3 by row hash.  The
     result spool file will not be cached in memory.  The size of Spool
     3 is estimated to be 2 rows.  The estimated time for this step is
     0.07 seconds.
  9) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of a
     RowHash match scan, which is joined to LOUISB.Lineitem.  Spool 3
     and LOUISB.Lineitem are joined using a merge join, with a join
     condition of ("LOUISB.Lineitem.l_orderkey = Spool_3.o_orderkey").
     The input table LOUISB.Lineitem will not be cached in memory, but
     it is eligible for synchronized scanning.  The result goes into
     Spool 2, which is built locally on the AMPs.  Then we do a SORT to
     order Spool 2 by row hash.  The result spool file will not be
     cached in memory.  The size of Spool 2 is estimated to be 20 rows.
     The estimated time for this step is 0.37 seconds.
 10) We do an all-AMPs RETRIEVE step from Spool 2 by way of an all-rows
     scan into Spool 4, which is built locally on the AMPs.  Then we do
     a SORT to order Spool 4 by join index.
```

```
11) We do a MERGE DELETE to LOUISB.OrderJoinLine from Spool 2 (Last
    Use).
12) We do a MERGE into LOUISB.OrderJoinLine from Spool 4 (Last Use).
13) We spoil the parser's dictionary cache for the table.
14) We spoil the parser's dictionary cache for the table.
15) Finally, we send out an END TRANSACTION step to all AMPs involved
    in processing the request.
 -> No rows are returned to the user as the result of statement 1.
```

# General Method of Maintaining Join Index During UPDATE

The following is an example of a general method of maintaining join index during an UPDATE.

Note the following items in the EXPLAIN output:

| This step ... | Does this ... |
|---|---|
| 5 | reproduces the affected portion of the join index rows. |
| 6.1 | deletes the corresponding rows from the join index. |
| 6.2 | reproduces the affected portion of the join index rows and makes the necessary modifications to form the new rows. |
| 7.1 | inserts the newly modified rows into the join index. |

```
   EXPLAIN
   UPDATE Lineitem
   SET l_extendedprice = l_extendedprice * .80
   WHERE l_partkey = 50;

    *** Help information returned. 47 rows.
    *** Total elapsed time was 3 seconds.

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for write on a
     RowHash to prevent global deadlock for LOUISB.OrderJoinLine.
  2) Next, we lock a distinct LOUISB."pseudo table" for read on a
     RowHash to prevent global deadlock for LOUISB.Ordertbl.
  3) We lock a distinct LOUISB."pseudo table" for write on a RowHash to
     prevent global deadlock for LOUISB.Lineitem.
  4) We lock LOUISB.OrderJoinLine for write, we lock LOUISB.Ordertbl
     for read, and we lock LOUISB.Lineitem for write.
  5) We do an all-AMPs JOIN step from LOUISB.Lineitem by way of a
     RowHash match scan with a condition of (
     "LOUISB.Lineitem.l_partkey = 50"), which is joined to
     LOUISB.Ordertbl.  LOUISB.Lineitem and LOUISB.Ordertbl are
     left outer joined using a merge join, with a join condition of (
     "LOUISB.Lineitem.l_orderkey = LOUISB.Ordertbl.o_orderkey").  The
     input tables LOUISB.Lineitem and LOUISB.Ordertbl will not be
     cached in memory, but they are eligible for synchronized scanning.
     The result goes into Spool 1, which is built locally on the AMPs.
     Then we do a SORT to order Spool 1 by row hash.  The result spool
     file will not be cached in memory.  The size of Spool 1 is
     estimated to be 100,000 rows.  The estimated time for this step is
     6 minutes and 14 seconds.
```

```
6) We execute the following steps in parallel.
     1) We do a MERGE DELETE to LOUISB.OrderJoinLine from Spool 1
        (Last Use).
     2) We do an all-AMPs JOIN step from LOUISB.Lineitem by way of a
        RowHash match scan with a condition of (
        "LOUISB.Lineitem.l_partkey = 50"), which is joined to
        LOUISB.Ordertbl.  LOUISB.Lineitem and LOUISB.Ordertbl are
        left outer joined using a merge join, with a join condition
        of ("LOUISB.Lineitem.l_orderkey = LOUISB.Ordertbl.o_orderkey").
        The input tables LOUISB.Lineitem and LOUISB.Ordertbl will not
        be cached in memory, but they are eligible for synchronized
        scanning.  The result goes into Spool 2, which is built
        locally on the AMPs.  Then we do a SORT to order Spool 2 by
        join index.  The result spool file will not be cached in
        memory.  The size of Spool 2 is estimated to be 100,000 rows.
        The estimated time for this step is 5 minutes and 7 seconds.
7) We execute the following steps in parallel.
     1) We do a MERGE into LOUISB.OrderJoinLine from Spool 2 (Last
        Use).
     2) We do an all-AMPs UPDATE from LOUISB.Lineitem by way of an
        all-rows scan with a condition of (
        "LOUISB.Lineitem.l_partkey = 50").
8) We spoil the parser's dictionary cache for the table.
9) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
-> No rows are returned to the user as the result of statement 1.
```

# Optimized Method of Maintaining Join Index During UPDATE

## Direct Access Index Updates

Optimizations have been made for update statements that allow the affected join index rows to be located via direct access. For example, if an UPDATE specifies a search condition on the primary or secondary of a join index, the affected join index rows are not reproduced. Instead, the join index may be directly searched for the qualifying rows and modified accordingly.

## Preconditions for Optimization

To use this optimized method (that is, the direct update approach), the following conditions must be present:

- A primary or secondary access path to the join index.
- If *join_index_field_2* is defined, no modifications to *join_index_field_1* columns.
- No modifications to the join condition columns appearing in the join index definition.
- No modifications to the primary index columns of the join index.

## Example

The following is an example of an optimized method for maintaining join index during an UPDATE:

```
   EXPLAIN
   UPDATE Lineitem
   SET l_quantity = l_quantity - 5
   WHERE l_orderkey = 10;

    *** Help information returned. 11 rows.
    *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP UPDATE from join index table
          LOUISB.OrderJoinLine by way of the primary index
          "LOUISB.OrderJoinLine.l_orderkey = 10" with a residual
          condition of ("LOUISB.OrderJoinLine.l_orderkey = 10").
       2) We do a single-AMP UPDATE from LOUISB.Lineitem by way of the
          primary index "LOUISB.Lineitem.l_orderkey = 10" with no
          residual conditions.
  2) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

# CHAPTER 4 Interpreting the Output of the EXPLAIN Request Modifier

This chapter describes several aspects of the EXPLAIN request modifier. See *SQL Reference: Data Manipulation Statements* 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 0-32767. 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 Reference: Data Definition Statements*.

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 Reference: Data Manipulation Statements*), a WITH RECURSIVE modifier (see *SQL Reference: Data Manipulation Statements*), or another EXPLAIN request modifier. Nor can you explain individual functions, stored procedures, or methods.

## 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 in *SQL Reference: Statement and Transaction Processing*.

# 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 Reference: Statement 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[1]

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]

1. The system reports Index Join confidence only for join operations.
2. 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.[3]

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

## 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:

| Confidence Level | Meaning | Reason |
|---|---|---|
| No | The system has neither Low nor High confidence in the cardinality and distinct value estimates for the relation.<br>The Optimizer pursues conservative strategies to optimize the relevant steps. | Any of the following situations exists:<br>• There are no statistics on the column or index sets specified in the predicate.<br>• The predicate contains complex expressions for which statistics cannot be collected.<br>For example, statistics cannot be collected for either of the following two expressions:<br>• `SUBSTR(col1,5,10)`<br>• `CASE WHEN x=10 THEN x+10 ELSE x-10`<br>• For an aggregate estimation, there are no statistics on the grouping columns. |

3. 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.

| Confidence Level | Meaning | Reason |
|---|---|---|
| Low | The system is moderately certain that the estimated cardinality and distinct value estimates for the relation are accurate.<br>The Optimizer pursues more aggressive strategies to optimize the relevant steps. | One of the following states exists for the relation:<br>• There are conditions in the query on an index set that has no statistics.<br>Cardinality estimates can be made based on sampling the index set.<br>• There are conditions in the query on an index or column set with statistics that is ANDed with conditions on non-indexed columns.<br>• There are conditions in the query on an index or column set with statistics that is ORed with other conditions.<br>• 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.<br>The confidence for single-AMP random AMP statistical samples is always Low.<br>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. |

| Confidence Level | Meaning | Reason |
|---|---|---|
| High | The system is fairly certain that the estimated cardinality and distinct value estimates for the relation are accurate.<br>The Optimizer pursues more aggressive strategies to optimize the relevant steps. | • Retrieval from a single relation with no predicates:<br>  • There are conditions in the query on the primary index and there are statistics on the primary index.<br>  • There are conditions in the query on the primary index, but there are no statistics on the primary index.<br>  The confidence is high under any of the following situations:<br>  • 5 or more AMPs are sampled.<br>  • Rows per value are sampled.<br>  • No skew is detected.<br>• Retrieval from a single relation with predicates:<br>  • There are statistics on the predicate columns or indexes and there is no skew.<br>  • There are multiple equality predicates with covering multitable statistics.<br>• For an aggregate estimation, the confidence is high under any of the following situations:<br>  • The grouping columns are constants.<br>  • The grouping columns have equality predicates.<br>  • The grouping columns are all covered by a single multicolumn statistics set.<br>  • There is only one grouping column and it has statistics.<br>The confidence for retrieval from volatile tables is always High because the system performs all AMPs sampling for volatile tables by default. |

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,[4] the join plan determines the best order for joining the relations. Because you can join as many as 64 tables per join clause, it is critical to minimize join cardinality estimation errors to ensure that an optimal query plan is generated.

4. Such as join method and join geography.

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 (see *SQL Reference: Statement and Transaction Processing* for details).

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:[5]

| Confidence Level | Meaning | Reason |
|---|---|---|
| No | The system has neither Low nor High nor Index Join confidence in the estimated join cardinality. | One (or both) relation in the join does *not* have statistics on the join columns. |
| 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.<br>• One relation in the join has Low confidence and the other has any of the following three confidence levels:<br>  • Low<br>  • High<br>  • 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:<br>• High<br>• 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.<br>• There is a foreign key relationship between the two relations in the join.<br>Note that because of the way Teradata implements PRIMARY KEY and UNIQUE INDEX constraints, these are essentially two ways of saying the same thing. |

5. 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 Reference: Statement 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.

# EXPLAIN Request Modifier Terminology

Terms used in EXPLAIN phrases are described in the following list:

| Phrase | Explanation |
|---|---|
| ***n* partitions of ...** | |
| | Only *n* of the partitions are accessed. *n* is greater than one. |
| **a single partition of ...** | |
| | Only a single partition is accessed. |
| **all-AMPs JOIN step by way of an all-rows scan ...** | |
| | 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. |
| **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.<br>**2** Each row-hash match is located and processed as follows:<br>**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.<br>**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.<br>**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. |

| Phrase | Explanation |
|---|---|
| **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.<br>BMSMS (bit map set manipulation step);<br>intersects the following row id bit maps:<br>1) The bit map built for … by way of index # *n*...<br>2) The bit map built for … by way of index # *n*...<br>…<br>The resulting bit map is placed in Spool *n*...<br>BMSMS...<br>Indicates that two or more bit maps are intersected by ANDing them to form one large bit map.<br>index # *n*...<br>Identifies, in the order in which they are ANDed, each nonunique secondary index used in the intersection.<br>resulting bit map is placed in Spool *n*...<br>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.<br>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.<br>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.<br>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 way of index # *n* and the bit map in Spool *n* ...** | |
| | 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). |

| Phrase | Explanation |
|---|---|
| **by way of the sort key in spool field1 ...** | |
| | Field1 is created to allow a tag sort. |
| **computed globally ...** | |
| | The computation involves all the intermediate spool file data. |
| **condition ...** | |
| | An intermediate condition that joins table rows (as compared with the overall join condition). |
| **duplicated on all AMPs ...** | |
| | 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. |
| **eliminating duplicate rows ...** | |
| | 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. |
| **<END ROW TRIGGER LOOP for step *n*.>** | |
| | Delimits the running of the last step in the row trigger loop.<br>Control moves to the next step outside the trigger. |
| **END TRANSACTION step ...** | |
| | This indicates that processing is complete and that any locks on the data may be released. Changes made by the transaction are committed. |
| **enhanced by dynamic partition...** | |
| | This indicates a join condition where dynamic partition elimination has been used. |
| **estimated size ...** | |
| | 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 *SQL Reference: Data Definition Statements*). |
| **estimated time ...** | |
| | 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. |

| Phrase | Explanation |
|---|---|
| **execute the following steps in parallel ...** | |
| | This identifies a set of steps that are processed concurrently. The explanatory text immediately following the list describes the execution of each step. |
| **join condition ...** | |
| | The overall constraint that governs the join.<br>In the following statement, "Employee.EmpNo = Department.MgrNo" is the overall constraint governing the join.<br>SELECT DeptName, Name FROM Employee, Department<br>WHERE Employee.EmpNo = Department.Mgrno; |
| **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 Teradata Database 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 types of join processing performed by the Teradata Database. |
| **nested join ...** | |
| | One of the types of join processing performed by the Teradata Database. |
| **no residual conditions ...** | |
| | Rows are selected in their entirety; there are no specific search conditions. All applicable conditions have been applied to the rows. |
| **of *n* partitions ...** | |
| | 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. |
| **of a single partition ...** | |

| Phrase | Explanation |
|---|---|
| | 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. |
| **product join ...** | |
| | One of the types of join processing performed by the Teradata Database. |
| **redistributed by hash code to all AMPs ...** | |
| | 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. |
| **single-AMP JOIN step by way of the unique primary index ...** | |
| | 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. |
| **single-AMP RETRIEVE by way of unique index # *n* ...** | |
| | 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. |
| **single-AMP RETRIEVE step by way of the unique primary index ...** | |
| | 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.<br>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. |
| **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.<br>The contents of field1 depend on the query and may comprise any of the following:<br>• 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).<br>• A concatenation of the row IDs that identify the data rows from which the spool row was formed (used for complex queries involving subqueries).<br>Some other value that uniquely defines the spool row (used for complex queries involving aggregates and subqueries). |

| Phrase | Explanation |
|---|---|
| **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 or columns defined in the ORDER BY or WITH...BY clause of the request being processed. |
| **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. |
| **spool *n* ...** | |
| | Identifies a spool file, which is a temporary file used to contain data during an operation.<br>• Spool 1 is normally used to hold a result before it is returned to the user.<br>• Spools 2, 3, etc., contain intermediate data that produces a result. |
| **spool_n.Field ...** | |
| | Identifies the field of the spooled rows that is being used in a join constraint or comparison operation.<br>For example:<br>PERSONNEL.Employee.EmpNo = Spool_2.MgrNo<br>Statement 1...<br>This term refers to the initiating request. |
| **the estimated time is nn seconds.** | |
| | The time shown is not clock seconds; you should consider it to be an arbitrary unit of measure that compares different operations. |
| **two-AMP RETRIEVE step by way of unique index #*n* ...** | |
| | A row of a table is selected based on a unique secondary index:<br>• A single row in the unique secondary index subtable is selected using the index value that hashes to the AMP on which the subtable row is stored.<br>• 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. |

| Phrase | Explanation |
|---|---|
| **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. |
| **which is duplicated on all AMPs …** | |
| | Relocating data in preparation for a join. |
| **which is redistributed by hash code to all AMPs …** | |
| | Relocating data in preparation for a join. |

# 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 244.

The following topics are described here.

# 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 an all-rows scan
   with a condition of ("PERSONNEL.employee.Name = 'Smith t'") 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 0.03 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 0.03 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.
-> The row is sent directly back to the user as the result of statement 1. The total
   estimated time 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.MgrNo"). 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.
-> 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 0.19 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
---------------------------------------------------------------------
1) First, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to prevent
   global deadlock for PERSONNEL.charges.
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.charges for read, and we lock PERSONNEL.employee for read.
```

```
4) We do an all-AMPs RETRIEVE step from PERSONNEL.charges 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 and the sort key in spool
   field1 eliminating duplicate rows.  The size of Spool 2 is estimated to be 12 rows.
   The estimated time for this step is 0.15 seconds.
5) We do an all-AMPs JOIN step from PERSONNEL.employee by way of an all-rows scan with
   no residual conditions, which is joined to Spool 2 (Last Use).  PERSONNEL.employee
   and Spool 2 are joined using an inclusion 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.  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 12 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.
```

## Example 6: Recursive Query

The following example demonstrates a recursive query:

```
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 0.08 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 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.
```

## 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', 890811, 2.5)
; INSERT Charges (30004, 'AP2-0004', 890831, 37.5
; INSERT Charges (30005, 'AP2-0004', 890825, 11.0)
; INSERT Charges (30006, 'AP2-0004', 890721, 24.5)
; INSERT Charges (30007, 'AP2-0004', 890811, 40.5)
; INSERT Charges (30008, 'AP2-0004', 890831, 32.0
; INSERT Charges (30009, 'AP2-0004', 890825, 41.5)
; INSERT Charges (30010, '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;
```

In ANSI mode, EXPLAIN generates the following response for this request:

```
Explanation
-----------------------------------------------------------------------
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.

```
Explanation
-----------------------------------------------------------------------
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.
```

```
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 field1
   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.
    <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.
```

# 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 268).

```
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-003'" 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:

| Phrase | Definition |
|---|---|
| single-AMP RETRIEVE step from … by way of the unique primary index | 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. |
| duplicated on all AMPs | 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. |

| Phrase | Definition |
|---|---|
| all-AMPs JOIN step … by way of an all-rows scan | 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. |
| condition of … | An intermediate condition used to qualify the joining of selected rows with spooled rows (as compared with an overall join condition). |
| product join | One of the types of join processing performed by the Teradata Database. See *SQL Reference: Statement and Transaction 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
-----------------------------------------------------------
1) First, we lock PERSONNEL.Department for read, and we lock
PERSONNEL.Employee for read.
2) Next, we do an all-AMPs RETRIEVE step from PERSONNEL.Employee 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 0.10 seconds.
3) We do an all-AMPs JOIN step from PERSONNEL.Department by way of a RowHash match
scan with no residual conditions, which is joined to Spool 2 (Last Use).
PERSONNEL.Department and Spool 2 are joined using a merge join, with a join condition
of ("Spool_2.DeptNo = PERSONNEL.Department.DeptNo"). 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 0.11 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 0.22 seconds.
```

## Terminology

New terminology in this explanation is defined as follows.

| Phrase | Definition |
|---|---|
| merge join | One of the types of join processing performed by the Teradata Database. See *SQL Reference: Statement and Transaction Processing*. |

## 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:

| Phrase | Definition |
|---|---|
| hash join | One of the types of join processing performed by the Teradata Database. See *SQL Reference: Statement and Transaction Processing*. |

## 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;

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:

| Phrase | Definition |
|---|---|
| single-AMP JOIN step from … by way of the unique primary index | A single row on one AMP is selected using the value of a unique index defined for its table.<br>Using the value in that row which hashes to a unique index value in a row of a second table on another AMP, the first row is joined with the second row. (Note that when a table is accessed by its unique primary index, the need for a rowhash lock on the table is implied, even though it is not explicitly stated in the Explain text.) |
| nested join | One of the types of join processing performed by the Teradata Database. See *SQL Reference: Statement and Transaction Processing*. |

## 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:

| Phrase | Definition |
|---|---|
| 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.<br>The contents of field1 depend on the query and may comprise any of the following:<br>• 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).<br>• A concatenation of the row IDs that identify the data rows from which the spool row was formed (used for complex queries involving subqueries).<br>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 the Teradata Database. See *SQL Reference: Statement and Transaction Processing*. |

# 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.
```

## Terminology

New terminology in this explanation is defined as follows:

| Phrase | Definition |
|---|---|
| SUM step to aggregate | 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. |
| computed globally | The final computation involves all the intermediate spool-file data. |

## Example 2: Indexed Access With Bit Mapping

This request returns the following EXPLAIN report:

```
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:

| Phrase | Definition |
|---|---|
| A BMSMS (bit map set manipulation step) that intersects the following Row-Id bit maps:<br>**1** The bit map built for … by way of index # *n* …<br>**2** The bit map built for … by way of index # *n* … | 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.<br>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. |
| residual condition | Selected rows are further qualified by one or more conditional expressions. |

# 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.

```
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 5.
    No rows are returned to the user as the result of statement 6.
```

# 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:

| Phrase | Definition |
|---|---|
| *n* partitions of | Only *n* of the partitions are accessed, where $n > 1$. In this case, $n = 8$. |

## 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.

```
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:

| Phrase | Definition |
|---|---|
| a single partition of | Only one partition is accessed in processing this query. |

## 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.

```
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:

| Phrase | Definition |
|---|---|
| all partitions of | All partitions are accessed for primary index access in processing this query. |

## 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.

```
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.

```
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:

| Phrase | Definition |
|---|---|
| of a single partition | 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. |
| of *n* partitions | The optimizer determined that all rows in each of *n* partitions can be deleted, where $n > 1$. In some cases, this allows for faster deletion of entire partitions. |

## 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.

```
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:

| Phrase | Definition |
|---|---|
| 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.<br>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. |
| a rowkey-based | 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.<br>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.<br>Note that with either method, the join conditions must still be validated. |

## 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',
      l_commitdate    DATE FORMAT 'yyyy-mm-dd',
      l_receiptdate   DATE FORMAT 'yyyy-mm-dd',
      l_shipinstruct  VARCHAR(25),
      l_shipmode      VARCHAR(10),
      l_comment       VARCHAR(44))
   PRIMARY INDEX (l_orderkey)
PARTITION BY RANGE_N(
  l_shipdate BETWEEN DATE '1992-01-01' AND DATE '1998-12-31'
  EACH INTERVAL '1' MONTH);

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')"),
   which is joined to TH.LINEITEM with a condition of (
   "TH.LINEITEM.L_COMMITDATE < TH.LINEITEM.L_RECEIPTDATE").
   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.

```
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.[6]

## Terminology

New terminology in this set of EXPLAIN reports is defined as follows:

| Phrase | Definition |
|---|---|
| if no update in <*step number*> | The match condition for the update phase was not met, so the conditional insert phase will be performed.<br><*step number*> indicates the MERGE update step. |
| if <*step number*> not executed, | Indicates the action taken if the first step in an UPDATE block in not performed.<br>Reported only if <*step number*> is greater than 1. |

## Database Object DDL for Examples

The following DDL statements create the tables accessed by the EXPLAINed DML statement examples:

```
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);
```

6. 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);
```

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:

```
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:

```
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:

```
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.[7]

## Terminology

New terminology in this set of EXPLAIN reports is defined as follows:

| Phrase | Definition |
|---|---|
| if no update in <*step number*> | The match condition for the update phase was not met, so the conditional insert phase will be performed.<br><*step number*> indicates the MERGE update step. |
| if <*step number*> not executed, | Indicates the action taken if the first step in an UPDATE block in not performed.<br>Reported only if <*step number*> is greater than 1. |

## Database Object DDL for Examples

The following DDL statements create the tables accessed by the EXPLAINed DML statement examples.

```
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:

```
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.
```

7. Atomic upsert conditional steps are insert operations that are performed after an unconditional update operation does not meet its matching condition.

```
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:

```
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:

```
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 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
-----------------------------------------------------------------------------
  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 = 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:

```
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.
```

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 295, as the EXPLAIN report at the end confirms.

```
ALTER TRIGGER r16 DISABLED;
ALTER TRIGGER r17 DISABLED;
ALTER TRIGGER r18 DISABLED;
ALTER TRIGGER r19 DISABLED;
ALTER TRIGGER r20 DISABLED;

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););

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.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.
2) 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.
3) If no update in 1.2, we do an INSERT into Test.t17.
4) 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.
5) 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.
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 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 = 7 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 5 Query Capture Facility

This chapter describes the Query Capture Database (QCD) that supports the various Database Query Analysis tools such as the Teradata Database 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.

Among the topics documented are the following:

# 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 release 5.0 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 release 5.0 must be examined carefully to ensure that they continue to work and must be converted if they are no longer compatible with the QCD schema.

# Functional Overview of the Query Capture Facility

## Introduction

The Query Capture Facility, or QCF, provides a method to capture and store the steps from any query plan in a set of predefined relational tables called the Query Capture Database, or QCD.

You create your QCD databases using the procedures described in "Creating the QCD Tables" on page 308.

## 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 Reference: Data Definition Statements* for further information.

## Applications of QCF and QCD

The 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.

- 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.

# 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 308) or BTEQ using the CreateQCF table (see "Procedure Using BTEQ" on page 309) 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.

# 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 310.

## 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 309.

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 Reference: Data Definition Statements* 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.

   ```
   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 Reference: Data Definition Statements* 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.
6. End of procedure.

# 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 310.

1. Start Visual Explain.
2. Pull down the Tools menu and select Control Center.
3. Select the Manage QCD tab.
4. Select the Clean up QCD button.
5. Select the database to be cleaned up from the Database scroll bar menu.
6. Select the Delete Database Objects radio button.
7. Select the OK button.
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.

   ```
   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.

# Querying QCD

You can analyze the information in QCD using several different approaches.

- To create and compare graphical EXPLAIN reports, use the Visual EXPLAIN tool.
- To perform ad hoc queries of QCD, use interactive SQL DML statements.
- To perform 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 313 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 Reference: Data Definition Statements* for the syntax of the DIAGNOSTIC "Validate Index” statement.

# QCD Query Macros and Views

## Introduction

The 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.

The 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:

| Version | Definition |
|---|---|
| X | Access restricted to information inserted by the current user in the QCD. |
| NonX | Access permitted to information inserted by any user of the QCD. |

These macros and views are usually performed via the Visual EXPLAIN and Teradata Index Wizard client utilities.

## User Categories

The following categories of users are defined on a QCD to enhance its security:

| User Category | Description |
|---|---|
| Normal | Loads, views, and deletes own plans or workloads only. |
| Power | Loads and views plans or workloads inserted by any user.<br>Deletes own plans or workloads only. |
| Administrator | Loads, view, and deletes any plan created by any user.<br>Drops and deletes QCD tables.<br>QCD creator has Administrator privileges granted by default. |

## Specific User Category Privileges

The following table indicates the specific privileges granted to each user category:

| User Category | Database Object Type | Privileges Granted |
|---|---|---|
| Normal | QCD tables | • INSERT<br>• UPDATE<br>• SELECT on the following tables:<br>  • IndexRecommendations<br>  • SeqNumber<br>  • TableStatistics<br>• DELETE on the following table:<br>  AnalysisLog |
| | X views | SELECT |
| | X macros | EXEC |
| | Non-X views | none |
| | Non-X macros | none |
| Power | QCD tables | • INSERT<br>• UPDATE<br>• SELECT on the following tables:<br>  • IndexRecommendations<br>  • SeqNumber<br>  • TableStatistics<br>• DELETE on the following table:<br>  AnalysisLog |
| | X views | SELECT |
| | X macros | EXEC |
| | Non-X views | SELECT |
| | Non-X macros | EXEC (excluding DELETE plan and workload macros) |

| User Category | Database Object Type | Privileges Granted |
|---|---|---|
| Administrator | QCD tables | • DELETE<br>• DROP<br>• INSERT<br>• SELECT<br>• UPDATE |
| | X views[1] | • DELETE<br>• DROP<br>• INSERT<br>• SELECT<br>• UPDATE |
| | X macros | • DROP<br>• EXECUTE |
| | Non-X views | • DELETE<br>• DROP<br>• INSERT<br>• SELECT<br>• UPDATE |
| | Non-X macros | • DROP<br>• EXECUTE |

1. X views use the value for UDB_Key in the Query table (see "Query" on page 345) 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

## Function

Records the log information for index analysis by the Teradata Index Wizard utility.

The table maintains the checkpoint information that is recorded when you enable the CHECKPOINT option for the INITIATE INDEX ANALYSIS statement.

## Table Definition

```
CREATE TABLE AnalysisLog (
 WorkLoadID         INTEGER NOT NULL,
 RecommendationID   INTEGER NOT NULL,
 IndexNameTag       VARCHAR(30) CHARACTER SET UNICODE
                    NOT CASESPECIFIC NOT NULL,
 SeqNumber          SMALLINT NOT NULL,
 Cflag              CHARACTER(1) CHARACTER SET LATIN
                    NOT CASESPECIFIC NOT NULL,
 UserName           VARCHAR(30) CHARACTER SET UNICODE
                    NOT CASESPECIFIC NOT NULL,
 AnalysisStatus     VARBYTE(32000) NOT NULL,
 StartTime          TIMESTAMP(6) NOT NULL,
 UpdateTime         TIMESTAMP(6) NOT NULL)
PRIMARY INDEX (WorkloadID, IndexNameTag);
```

## Attribute Definitions

The following table defines the AnalysisLog table attributes:

| Attribute | Definition |
|---|---|
| WorkloadID | • Uniquely identifies the workload.<br>• Partial NUPI for the table. |
| RecommendationID | Uniquely identifies the index recommendation generated for the specific index analysis. |
| IndexNameTag | • User-specified name for the index analysis.<br>• Partial NUPI for the table. |
| SeqNumber | A sequence number to identify individual rows that belong to a multirow AnalysisStatus.<br>The sequence begins at 1. |

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>Cflag</td><td>Used with multirow AnalysisStatus.<br><br><table><tr><th>IF the row is …</th><th>THEN Cflag is set to …</th></tr><tr><td>the last in the sequence</td><td>F</td></tr><tr><td>not the last in the sequence</td><td>T</td></tr></table></td></tr>
<tr><td>UserName</td><td>Name of the user performing the index analysis.</td></tr>
<tr><td>AnalysisStatus</td><td>Identifies the status of the index analysis.<br><br>This column contains encoded information that the index wizard infrastructure uses. During a checkpoint operation, this information is updated.</td></tr>
<tr><td>StartTime</td><td>Timestamp for the beginning of the index analysis.</td></tr>
<tr><td>UpdateTime</td><td>Timestamp for the moment the AnalysisStatus column is most recently updated.</td></tr>
</table>

# DataDemographics

## Function

Contains table demographic information for use by the Teradata Index Wizard and Visual EXPLAIN client utilities.

## Table Definition

The following CREATE TABLE statement defines the DataDemographics table:

```
CREATE TABLE DataDemographics (
 MachineName     VARCHAR(30) CHARACTER SET UNICODE
                 UPPERCASE NOT CASESPECIFIC NOT NULL,
 TableName       VARCHAR(30) CHARACTER SET UNICODE
                 UPPERCASE NOT CASESPECIFIC NOT NULL,
 DatabaseName    VARCHAR(30) CHARACTER SET UNICODE
                 UPPERCASE NOT CASESPECIFIC NOT NULL,
 DBSize          INTEGER NOT NULL,
 CollectedTime   TIMESTAMP(6) NOT NULL,
 AMPNumber       INTEGER NOT NULL,
 ClusterNumber   INTEGER NOT NULL,
 SubTableID      SMALLINT NOT NULL,
 SubTableType    VARCHAR(120),
 RowCount        DECIMAL(18,0) NOT NULL,
 AvgRowSize      INTEGER NOT NULL,
 QueryID         INTEGER,
 IndexName       VARCHAR(2048) CHARACTER SET UNICODE
                 UPPERCASE NOT CASESPECIFIC NOT NULL,
 DemographicsID INTEGER)
PRIMARY INDEX (MachineName, DatabaseName, TableName);
```

## Attribute Definitions

The following table defines the DataDemographics table attributes:

| Attribute | Definition |
|---|---|
| MachineName | • The name of the system to which TableName belongs.<br>• Partial NUPI for the table. |
| TableName | • The name of the table on which demographic data has been collected.<br>• Partial NUPI for the table. |
| DatabaseName | • The name of the containing database for TableName.<br>• Partial NUPI for the table. |
| DBSize | The size in KB of data blocks in TableName. |
| CollectedTime | The timestamp value when the data demographics were collected. |

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>AMPNumber</td><td>The AMP Vproc number to which the row information pertains.</td></tr>
<tr><td>ClusterNumber</td><td>The number of the cluster to which AMPNumber belongs.</td></tr>
<tr><td>SubTableID</td><td>The unique identifier for the subtable in which the data the row describes is stored.</td></tr>
<tr><td>SubTableType</td><td>The subtable data in text format.</td></tr>
<tr><td>RowCount</td><td>The cardinality of the subtable.</td></tr>
<tr><td>AvgRowSize</td><td>The average size of a row in the subtable.</td></tr>
<tr><td>QueryID</td><td>The value for QueryID depends on how the demographics are captured.
<table>
<tr><th>IF demographics are captured by this statement ...</th><th>THEN QueryID is ...</th></tr>
<tr><td>COLLECT DEMOGRAPHICS</td><td>null.</td></tr>
<tr><td>INSERT EXPLAIN</td><td>the unique ID of the query.</td></tr>
</table></td></tr>
<tr><td>IndexName</td><td>A comma-separated list of the names of the primary and secondary indexes for the table.<br>If an index has no name, then it is represented in this field by a comma-separated list of the names of the columns that compose it.</td></tr>
<tr><td>DemographicsID</td><td>Set to 1 if the demographics are captured by a COLLECT DEMOGRAPHICS or INSERT EXPLAIN AND DEMOGRAPHICS statement.<br>1 indicates that the capture is on the system on which the row is inserted.<br>DemographicsID has different values if the demographics are imported rather than captured directly.</td></tr>
</table>

## Spatial Distribution of a Table Across AMPs

DataDemographics does not contain information about the spatial distribution of tables across the AMPs.

You can view those details by querying the system view DBC.TableSizeX.

# Field

## Function

Captures all the columns (fields) used or referenced in a captured query plan.

## Table Definition

The following CREATE TABLE statement defines the Field table:

```
CREATE TABLE Field (
   FieldID              INTEGER NOT NULL,
   RelationKey          INTEGER NOT NULL,
   Name                 VARCHAR(30) CHARACTER SET UNICODE
                        NOT CASESPECIFIC,
   FldAlias             VARCHAR(30) CHARACTER SET UNICODE
                        NOT CASESPECIFIC,
   QueryID              INTEGER NOT NULL,
   ValueAccessFrequency INTEGER,
   JoinAccessFrequency  INTEGER,
   RangeAccessFrequency INTEGER,
   ChangeRate           INTEGER,
   DataLength           INTEGER,
   StatsKind            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:

| Attribute | Definition |
|---|---|
| FieldID | • Unique field identifier within a specific table.<br>• Partial USI for the table. |
| RelationKey | • Unique identifier for the relation in which the field is defined within a certain database.<br>• NUPI for the table.<br>• Partial USI for the table. |
| Name | The name of the captured column. |

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>FldAlias</td><td>• Alias name for the field.<br>• Null if none exists.</td></tr>
<tr><td>QueryID</td><td>The unique ID for the query.</td></tr>
<tr><td>ValueAccessFrequency</td><td>The number of times the column is used with an equality condition.</td></tr>
<tr><td>JoinAccessFrequency</td><td>The number of times the column is used in a join condition.</td></tr>
<tr><td>RangeAccessFrequency</td><td>The number of times the column is used in a range condition.</td></tr>
<tr><td>ChangeRate</td><td>The change rating value for the column.<br><table><tr><th>IF the column is ...</th><th>THEN ChangeRate is set to this value ...</th></tr><tr><td>modified</td><td>1</td></tr><tr><td>not modified</td><td>0</td></tr></table></td></tr>
<tr><td>DataLength</td><td>The maximum length of the column.</td></tr>
<tr><td>StatsKind</td><td>Defines whether the statistics are collected from the dictionary or from a QCD.<br>The data for this column is retrieved from interval 0 of the available statistics.<br>The field is set to null if no statistics are available for the column or index.<br>The value is set to S during index validation by the Teradata Index Wizard. Statistics for the column being analyzed are retrieved from the QCD.<br><table><tr><th>Code</th><th>Description</th></tr><tr><td>P</td><td>Dictionary statistics.</td></tr><tr><td>S</td><td>QCD statistics.</td></tr><tr><td>null</td><td>No statistics.</td></tr></table></td></tr>
<tr><td>NumNulls</td><td>Number of nulls for the column or index.<br>The data for this column is retrieved from interval 0 of the available statistics. The field is set to null if no statistics are available.</td></tr>
<tr><td>NumIntervals</td><td>Number of intervals for the column or index.<br>The data for this column is retrieved from interval 0 of the available statistics. The field is set to null if no statistics are available.</td></tr>
<tr><td>MinValue</td><td>Minimum value for the interval.<br>The data for this column is retrieved from interval 0 of the available statistics. The field is set to null if no statistics are available.</td></tr>
</table>

| Attribute | Definition |
|---|---|
| ModeValue | Modal value for the interval.<br>The data for this column is retrieved from interval 0 of the available statistics. The field is set to null if no statistics are available. |
| ModeFreq | Number of occurrences of the modal value in the interval.<br>The data for this column is retrieved from interval 0 of the available statistics. The field is set to null if no statistics are available. |
| TotalValues | Number of unique values for the column or index in the table.<br>The data for this column is retrieved from interval 0 of the available statistics. The field is set to null if no statistics are available. |
| TotalRows | Cardinality of the table.<br>The data for this column is retrieved from interval 0 of the available statistics. The field is set to null if no statistics are available. |

# Index_Field

## Function

Captures all the index fields used in the query plan.

## Table Definition

The following CREATE TABLE statement defines the Index_Field table:

```
CREATE TABLE Index_Field(
 RelationKey  INTEGER NOT NULL,
 IndexNum     INTEGER NOT NULL,
 FieldID      INTEGER NOT NULL)
PRIMARY INDEX PK_Relationkey_IdxNum(RelationKey, IndexNum)
UNIQUE INDEX USK_RelationKey_IdxNum_FieldID (RelationKey,
IndexNum, FieldID);
```

## Attribute Definitions

The following table defines the Index_Field table attributes:

| Attribute | Definition |
|---|---|
| RelationKey | • Unique identifier for the relation in which the captured index fields are defined.<br>• Partial NUPI for the table.<br>• Partial USI for the table. |
| IndexNum | • Unique identifier for the captured index.<br>• Partial NUPI for the table.<br>• Partial USI for the table. |
| FieldID | • Unique field identifier within a specific table.<br>• Partial USI for the table. |

# IndexColumns

## Function

Captures the columns that form the index identified by the IndexID during index analysis by the Teradata Index Wizard.

See "JoinIndexColumns" on page 337 for the definition of a QCD table that captures similar information for join index recommendations.

## Table Definition

The following CREATE TABLE statement defines the IndexColumns table:

```
CREATE TABLE IndexColumns(
 WorkLoadID       INTEGER NOT NULL,
 RecommendationID INTEGER NOT NULL,
 TableID          BYTE(6) NOT NULL,
 IndexID          INTEGER NOT NULL,
 ColumnName       VARCHAR(30) CHARACTER SET UNICODE
                  NOT CASESPECIFIC NOT NULL)
PRIMARY INDEX ( RecommendationID, TableID, IndexID );
```

## Attribute Definitions

The following table defines the IndexColumns table attributes:

| Attribute | Definition |
|---|---|
| WorkloadID | Uniquely identifies the workload analyzed to create this secondary index recommendation. |
| RecommendationID | • Uniquely identifies the recommendation ID in the IndexRecommendations table.<br>• Partial NUPI for the table. |
| TableID | • Uniquely identifies the table ID in the IndexRecommendations table.<br>• Partial NUPI for the table. |
| IndexID | • Uniquely identifies the index in the IndexRecommendations table.<br>• Partial NUPI for the table. |
| ColumnName | The name of the column in the index.<br>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 statements in maintaining the recommended indexes stored in IndexRecommendations.

IndexMaintenance contains one row for each SQL statement-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:

```
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:

| Attribute | Description |
|---|---|
| WorkloadID | • Uniquely identifies the workload analyzed to create the attribute data.<br>• NUPI for the table. |
| RecommendationID | Uniquely identifies the recommendation ID in the IndexRecommendations table. |
| IndexNameTag | User-specified name for the index analysis. |
| BaseTableID | The ID of the base table that is being updated. |

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>BaseTableName</td><td>The name for the base table referenced by BaseTableID.</td></tr>
<tr><td>JINumber</td><td>The sequentially assigned number for the recommended join index in the IndexRecommendations table.<br>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.<br>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.<br>Each unique index type is associated with its own IndexTypeText.<br>The valid IndexType codes and their corresponding IndexTypeText strings are as follows:
<table>
<tr><th>IndexType Code</th><th>IndexTypeText</th></tr>
<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>
</table></td></tr>
<tr><td>IndexTypeText</td><td>The textual representation of IndexType.<br>The valid IndexTypeText strings and their meanings are as follows:
<table>
<tr><th>IndexType Text</th><th>Description</th></tr>
<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>
</table></td></tr>
</table>

| Attribute | Description |
|---|---|
| MaintCosts | The estimated cost in milliseconds to update the recommended index structures. |

# 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 statement defines the IndexRecommendations table:

```
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,
 IndexMaintMode       BYTEINT DEFAULT NULL,
 JINumber             INTEGER DEFAULT NULL,
 JITableName          VARCHAR(30) CHARACTER SET UNICODE DEFAULT NULL))
```

```
PRIMARY INDEX (WorkloadID);
```

## Attribute Definitions

The following table defines the IndexRecommendations table attributes:

| Attribute | Definition |
|---|---|
| WorkloadID | • Uniquely identifies the workload.<br>• NUPI for the table. |
| UserName | Name of the user performing the index analysis. |
| TimeOfAnalysis | The timestamp when the index recommendations were analyzed.<br>You can compare this field with the modified timestamp for TableName to verify the correctness of the recommendation before applying it to the system. |
| RecommendationID | Uniquely identifies a set of index recommendations. |
| QueryID | Uniquely identifies the QueryID of the workload for which the current entry is an index recommendation. |
| IndexID | Uniquely identifies a unique secondary index recommended by the Teradata Index Wizard for a table.<br>Set null, meaning not applicable, when the value of IndexType is 5 or 6, indicating a join index. |
| IndexNameTag | Name of the index recommendation as specified in the INITIATE INDEX ANALYSIS statement (see *SQL Reference: Data Definition Statements*). |
| TableName | Name of the table for which the row defines an index recommendation. |
| DatabaseName | Name of the database containing TableName. |
| TableID | The unique internal identifier for TableName. |
| IndexType | A number that identifies the type of index recommended.<br>Each unique index type is associated with its own IndexTypeText. |

| IndexType Code | IndexTypeText |
|---|---|
| 1 | Unique Secondary Index (USI) |
| 2 | Value-Ordered Secondary Index (VOSI) |
| 3 | Hash-Ordered Secondary Index (HOSI) |
| 4 | Nonunique Secondary Index (NUSI) |
| 5 | Simple Join Index (JI) |
| 6 | Aggregate Join Index (JIAGG) |

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>IndexTypeText</td><td>The textual representation of IndexType.<br>The valid IndexTypeText strings and their meanings are as follows:<br><table><tr><th>IndexTypeText</th><th>Description</th></tr><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></table></td></tr>
<tr><td>StatisticsInfo</td><td>The statistics, if any, used to make the index recommendations.</td></tr>
<tr><td>OriginalCost</td><td>The estimated cost of the query in milliseconds before implementing the recommended indexes.</td></tr>
<tr><td>NewCost</td><td>The estimated cost of the query in milliseconds after implementing the recommended indexes.</td></tr>
<tr><td>SpaceEstimate</td><td>The estimated space in bytes the recommended index occupies when created.</td></tr>
<tr><td>TimeEstimate</td><td>The estimated time in milliseconds required to implement the index recommendation.</td></tr>
<tr><td>DropFlag</td><td>Identifies whether the specified index is to be added or dropped.<br><table><tr><th>Code</th><th>Description</th></tr><tr><td>N</td><td>The index is not recommended for dropping.</td></tr><tr><td>Y</td><td>The index is recommended for dropping.</td></tr></table></td></tr>
<tr><td>IndexDDL</td><td>The DDL text of the CREATE INDEX, DROP INDEX, CREATE JOIN INDEX, or DROP JOIN INDEX statement for the index recommended by the Teradata Index Wizard.</td></tr>
<tr><td>StatsDDL</td><td>The DDL text of the COLLECT STATISTICS (QCD form) statements used for the analysis of the index.</td></tr>
<tr><td>Remarks</td><td>Provides details on the analysis involved in making the index recommendation.</td></tr>
<tr><td>AnalysisData</td><td>Reserved for future use.</td></tr>
<tr><td>IndexesPerTable</td><td>The limit on the number of indexes on a given table as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis.</td></tr>
</table>

| Attribute | Definition |
|---|---|
| SearchSpaceSize | The maximum number of candidate indexes that are searched on a given table as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis. |
| ChangeRateThreshold | The threshold value of the column volatility as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis.<br>Any column with a change rating less than ChangeRateThreshold is available for selection as a candidate index during index analysis. |
| ColumnsPerIndex | The maximum number of columns permissible in the index as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis. |
| ColumnsPerJoinIndex | 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 *boundary_option* specification. |
| IndexMaintMode | The integer value used during analysis to control how estimated index maintenance costs are used.<br>The value is specified by the Teradata Index Wizard or using the INITIATE INDEX ANALYSIS SET *boundary_option* specification. |
| JINumber | An integer value sequence number that identifies the recommended Join Index table for a given index analysis.<br>The field is set to null for index types other than Join Index. |
| JITableName | A system-assigned name for the Join Index.<br>If JINumber is null, then so is JITableName.<br>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. |

# IndexTable

## Function

Describes all indexes on the tables specified by the query.

## Table Definition

The following CREATE TABLE statement defines IndexTable:

```
CREATE TABLE IndexTable (
 IndexNum         INTEGER NOT NULL,
 RelationKey      INTEGER NOT NULL,
 OrderBy          CHARACTER(1) NOT NULL,
 AccessInfo       CHARACTER(1) NOT NULL,
 Field1Only       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:

| Attribute | Description |
|---|---|
| IndexNum | • Unique identifier for the captured index.<br>• Partial USI for the table. |
| RelationKey | • Unique identifier for the relation in which the captured index is defined.<br>• NUPI for the table.<br>• Partial USI for the table. |

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>OrderBy</td><td>Defines whether the index has an associated ORDER BY clause.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Index has no ORDER BY clause.</td></tr><tr><td>T</td><td>Index has an ORDER BY clause.</td></tr></table></td></tr>
<tr><td>AccessInfo</td><td>Specifies if the index is a covering index, bit map, or neither.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>B</td><td>Bit map.</td></tr><tr><td>C</td><td>Covering index.</td></tr><tr><td>N</td><td>Neither bit map nor covering index.</td></tr><tr><td>P</td><td>Primary index access.</td></tr></table></td></tr>
<tr><td>Field1Only</td><td>Defines whether the index is a join index and Field1 is the only part needed.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Not a join index requiring Field1 only. This generally means one of two things:<br>• The index is not a join index.<br>• The index is a compressed join index.</td></tr><tr><td>T</td><td>Join index requiring Field1 only.<br>This means that the join index is not compressed.</td></tr></table></td></tr>
<tr><td>RangeConstraint</td><td>Flag for value-ordered indexes that have a range constraint used by the query plan.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>There is no range constraint on the index.<br>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></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>IndexFlag</td><td>Flag indicating whether the index was used in the query plan.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>False.<br>The index was not used in the query plan.</td></tr><tr><td>T</td><td>True.<br>The index was used in the query plan.<br>IndexFlag is also set to T if a subset of the partitions of a partitioned primary index is accessed because of partition elimination.</td></tr></table></td></tr>
<tr><td>IndexName</td><td>• The name of the index if it has one.<br>• Null if this is not a named index.</td></tr>
<tr><td>IndexType</td><td>Flag indicating the type of the index.<br><br><table><tr><th>Code</th><th>Description</th></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></table></td></tr>
<tr><td>UniqueFlag</td><td>Flag indicating whether the index is unique or nonunique.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Nonunique index.</td></tr><tr><td>T</td><td>Unique index.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>IndexKind</td><td>Flag indicating whether the index is permanent or simulated.<br>Indexes are simulated using the index validation function of the Teradata Index Wizard (see “Index Validation” on page 392).<br><table><tr><th>Code</th><th>Description</th></tr><tr><td>P</td><td>Permanent index.</td></tr><tr><td>S</td><td>Simulated index.</td></tr></table></td></tr>
<tr><td>NumNulls</td><td>The number of nulls in the index.</td></tr>
<tr><td>NumIntervals</td><td>The number of intervals in the index statistics.</td></tr>
<tr><td>MinValue</td><td>The minimum value of the index.<br>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.<br>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>
</table>

# JoinIndexColumns

## Function

Captures the columns that form the join index identified by JINumber during index analysis by the Teradata Index Wizard.

See "IndexColumns" on page 325 for the definition of the QCD table that captures similar information for secondary index recommendations.

## Table Definition

The following CREATE TABLE statement defines JoinIndexColumns:

```
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:

| Attribute | Description |
|---|---|
| WorkLoadID | Uniquely identifies the workload analyzed to create this join index recommendation. |
| RecommendationID | • Uniquely identifies a set of index recommendations in the IndexRecommendations table.<br>• 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.<br>• Partial NUPI for the table. |

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>JINumber</td><td>• The system-assigned sequence number for the join index in the IndexRecommendations table.<br>• Partial NUPI for the table.</td></tr>
<tr><td>ColumnName</td><td>Name of the join index column.</td></tr>
<tr><td>AliasName</td><td>The correlation name assigned to a column or aggregate function in the join index definition.</td></tr>
<tr><td>Field1Flag</td><td>Indicates whether the column is part of the column_1 (non-compressed fields) select list in the join index definition.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>False.<br>The column is not part of the column_1 select list.</td></tr><tr><td>T</td><td>True.<br>The column is part of the column_1 select list.</td></tr></table></td></tr>
<tr><td>Field2Flag</td><td>Indicates whether the column is part of the column_2 (compressed fields) select list in the join index definition.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>False.<br>The column is not part of the column_2 select list.</td></tr><tr><td>T</td><td>True.<br>The column is part of the column_2 select list.</td></tr></table></td></tr>
<tr><td>RowIDFlag</td><td>Indicates whether the value for the column is the reserved word ROWID.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>False.<br>The value for the column is not ROWID.</td></tr><tr><td>T</td><td>True.<br>The value for the column is ROWID.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>AggregateFunc</td><td>Indicates whether an aggregate function is applied to ColumnName and, if so, the type of aggregation performed.<br><br><table><tr><th>Code</th><th>Description</th></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></table></td></tr>
<tr><td>PrimaryIndexPosition</td><td>Indicates whether ColumnName is a component of the primary index for the join index.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>0</td><td>The column is not part of the primary index definition for the join index.</td></tr><tr><td>>1</td><td>The column is part of the primary index definition for the join index.<br>The value represents the position of ColumnName within the primary index definition for the join index.</td></tr></table></td></tr>
<tr><td>GroupByPosition</td><td>Indicates whether ColumnName is a component of the GROUP BY clause in the join index definition.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>0</td><td>The column is not part of the GROUP BY specification for the join index.</td></tr><tr><td>>1</td><td>The column is part of the GROUP BY specification for the join index.<br>The value represents the position of ColumnName within the GROUP BY specification for the join index.</td></tr></table></td></tr>
</table>

# Predicate

## Function

Describes any index, join, or residual conditions applied for specific AMP steps in a query.

## Table Definition

The following CREATE TABLE statement defines the Predicate table:

```
CREATE TABLE Predicate (
 PredicateID   INTEGER NOT NULL,
 StepID        INTEGER NOT NULL,
 PredicateKind 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:

| Attribute | Description |
|---|---|
| PredicateID | • Unique identifier for the predicate.<br>• UPI for the table. |
| StepID | Unique identifier for the AMP step. |
| PredicateKind | Describes the kind of predicate condition associated with this step. |

| Code | Description |
|---|---|
| A | Additional join condition. |
| G | Range constraint.<br>Used for value-ordered relations. |
| I | Condition associated with an index. |
| J | Join condition. |
| L | Condition on left relation in a join. |
| Q | Partition elimination occurs for a source condition.<br>This is a residual condition on the left or right table in a join or on a single-table retrieval. Partition elimination occurs prior to accessing the rows, so the condition applies only to rows retrieved from partitions that were not eliminated. |

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>PredicateKind<br>(continued)</td><td>
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>R</td><td>Condition on right relation in a join.</td></tr>
<tr><td>S</td><td>Source condition.<br>This is a residual condition on the left or right table in a join or on a single-table retrieval. No partition elimination occurs prior to accessing the rows.</td></tr>
</table>
</td></tr>
<tr><td>PredicateText</td><td>Full text of the predicate as it appears in the EXPLAIN report.</td></tr>
</table>

# Predicate_Field

## Function

Associates the list of fields specified in a captured predicate with the parent relation and predicate.

## Table Definition

The following CREATE TABLE statement defines the Predicate_Field table:

```
CREATE TABLE predicate_field (
 PredicateID INTEGER NOT NULL,
 RelationKey INTEGER NOT NULL,
 FieldID     INTEGER NOT NULL)
PRIMARY INDEX ( RelationKey );
```

## Attribute Definitions

The following table defines the Predicate_Field table attributes:

| Attribute | Definition |
|---|---|
| PredicateID | Uniquely identifies the predicate. |
| RelationKey | • Unique identifier for the referenced table.<br>• NUPI for the Predicate_Field table. |
| FieldID | Unique identifier for the field. |

# QryRelX

## Function

Stores overflow text for the QueryText attribute of the Query table or the TableDDL attribute of the Relation table.

There is at least one row in QryRelX for each row in Query or Relation with an Overflow flag set to T.

## Table Definition

The following CREATE TABLE statement defines the QryRelX table:

```
CREATE TABLE QryRelX(
 RowType   CHARACTER(1),
 KeyValue  INTEGER NOT NULL,
 SeqNumber SMALLINT NOT NULL,
 Text      VARCHAR(30000) CHARACTER SET UNICODE
           NOT CASESPECIFIC NOT NULL)
PRIMARY INDEX(RowType, KeyValue);
```

## Attribute Definitions

The following table defines the QryRelX table attributes:

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>RowType</td><td>Defines whether the row handles text overflow from the Query table or from the Relation table.<br><br><table><tr><th>Code</th><th>Definition</th></tr><tr><td>Q</td><td>Text is overflow from the Query table.</td></tr><tr><td>R</td><td>Text is overflow from the Relation table.</td></tr></table></td></tr>
<tr><td>KeyValue</td><td>Defines the implicit primary key for QryRelX rows.<br><br><table><tr><th>IF RowType is this value ...</th><th>THEN the value is the ...</th></tr><tr><td>Q</td><td>QueryID</td></tr><tr><td>R</td><td>RelationKey</td></tr></table></td></tr>
</table>

| Attribute | Definition |
|---|---|
| SeqNumber | Specifies whether the current row is the last row in the Overflow.<br>If the overflow QueryText or TableDDL text does not fit into a single row, it is divided into multiple rows, each with its unique SeqNumber value.<br>SeqNumber begins at 1 and is incremented by 1 for each new text row required. |
| Text | The overflow QueryText or TableDDL text.<br>The upper bound for this text is 30 000 characters per row. |

# Query

## Function

Describes information about captured queries.

## Table Definition

The following CREATE TABLE statement defines the Query table:

```
CREATE TABLE Query(
   QueryID         INTEGER NOT NULL,
   UDB_Key         INTEGER NOT NULL,
   MachName        VARCHAR(30) CHARACTER SET UNICODE
                   NOT CASESPECIFIC NOT NULL,
   NumAMPs         INTEGER NOT NULL,
   NumPEs          INTEGER NOT NULL,
   NumNodes        INTEGER NOT NULL,
   ReleaseInfo     VARCHAR(20) NOT NULL,
   VersionInfo     VARCHAR(20) NOT NULL,
   PENum           INTEGER NOT NULL,
   QueryText       VARCHAR(20000) CHARACTER SET UNICODE
                   NOT CASESPECIFIC NOT NULL,
   DateTimeStamp   DATE FORMAT 'YY/MM/DD' NOT NULL,
   QueryName       VARCHAR(30) CHARACTER SET UNICODE
                   NOT CASESPECIFIC,
   Frequency       INTEGER NOT NULL,
   StatementTypes VARCHAR(120) NOT NULL,
   DefaultDBName   VARCHAR(30) CHARACTER SET UNICODE
                   NOT CASESPECIFIC NOT NULL,
   Overflow        CHARACTER(1) CHARACTER SET LATIN,
   Complete        CHARACTER(1) CHARACTER SET LATIN,
   ValidatedPlan   CHARACTER(1),
   ImportedPlan    CHARACTER(1))
UNIQUE PRIMARY INDEX PK_QueryID ( QueryID );
```

## Attribute Definitions

The following table defines the Query table attributes:

| Attribute | Definition |
|---|---|
| QueryID | • Unique identifier for the query generated by the system when the query plan is captured.<br>• UPI for the table. |
| UDB_Key | Identifier for the user who captured the plan by submitting either a DUMP EXPLAIN or INSERT EXPLAIN statement (see *SQL Reference: Data Definition Statements* for descriptions of these statements). |
| MachName | Name of the test machine on which the query plan is captured. |

| Attribute | Definition |
|---|---|
| NumAMPs | Number of AMPs in the configuration. |
| NumPEs | Number of PEs in the configuration. |
| NumNodes | Number of nodes in the configuration. |
| ReleaseInfo | Teradata Database release number under which the query was captured.<br>The value is defined in DBC.DBCInfo. |
| VersionInfo | Teradata Database version number under which the query was captured.<br>The value is defined in DBC.DBCInfo. |
| PENum | Number of the parsing engine on which the query was processed. |
| QueryText | The SQL DML text of the captured query.<br>If the text exceeds the upper limit of 20 000 characters, then it overflows to the QryRelX table. See "Overflow" on page 347 and "QryRelX" on page 343. |
| DateTimeStamp | Timestamp that identifies when the captured query was performed.<br>Useful for distinguishing among multiple performances of the same query on the same machine under the same software version and release. |
| QueryName | The name of the query, if provided, as specified in the AS clause. |
| Frequency | The number of times the query is performed in the workload to which it is assigned.<br>The value is specified by the INSERT EXPLAIN statement used to create the row. |
| StatementTypes | A comma-separated list of 3-character statement type codes. |

| Code | Description |
|---|---|
| ABT | ABORT statement. |
| BTS | BEGIN TRANSACTION statement. |
| DEL | DELETE statement. |
| ETS | END TRANSACTION statement. |
| INS | INSERT statement. |
| NUL | Null statement. |
| OTR | Other statement.<br>This code describes any statement type that is not described by the other 9 statement type codes. |
| RET | Retrieve statement. |
| UPD | UPDATE statement. |
| URT | Update-Retrieve statement. |

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>DefaultDBName</td><td>The name of the default database at the time the query plan is captured.</td></tr>
<tr><td>Overflow</td><td>Specifies whether QueryText exceeds 20 000 characters.<br>If QueryText exceeds 20 000 characters, then all text beyond that boundary is truncated.<br><table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>QueryText <= 20 000 characters.</td></tr><tr><td>T</td><td>QueryText > 20 000 characters and has been truncated.</td></tr></table></td></tr>
<tr><td>Complete</td><td>Identifies whether Query stores the complete query text or a truncated version.<br><table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>Query text is truncated.<br>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 Reference: Data Definition Statements).<br>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.<br>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 343).</td></tr></table></td></tr>
<tr><td>ValidatedPlan</td><td>Specifies whether the plan was captured in validation or non-validation mode.<br>The value is always set to F during the capture.<br><table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>Query plan was captured in non-validation mode.</td></tr><tr><td>T</td><td>Query plan was captured in a validation mode.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>ImportedPlan</td><td>Specifies whether the query plan was captured on the current system or imported from another system.<br>The system always sets the flag to F during the capture.<br><table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>Query plan was captured on the current system.</td></tr><tr><td>T</td><td>Query plan was imported to the current system from another system.</td></tr></table></td></tr>
</table>

# QuerySteps

## Function

Each row in the table lists the attributes of any step executed by the system corresponding to the query. If the step has more than one attribute, then multiple rows are stored for the step and the row set is linked by means of its common StepID and QueryID values and the individual rows are distinguished by their row type codes.

## Table Definition

The following CREATE TABLE statement defines the QuerySteps table:

```
CREATE TABLE QuerySteps (
 StepID              INTEGER NOT NULL,
 QueryID             INTEGER NOT NULL,
 StepNum             INTEGER,
 ParallelStepNum     INTEGER DEFAULT 0,
 StepText            VARCHAR(32000) CHARACTER SET LATIN
                     NOT CASESPECIFIC,
 RowType             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,
 Cost                FLOAT,
 MaxCost             FLOAT,
 SourceRelation1     INTEGER,
 SourceRelation2     INTEGER,
 TargetRelation1     INTEGER,
 TargetRelation2     INTEGER,
 StepAttributeType   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>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StepID</td><td>• Unique identifier for the step.<br>• NUSI for the table.</td></tr>
<tr><td>QueryID</td><td>• Unique identifier for the query.<br>• NUPI for the table.</td></tr>
<tr><td>StepNum</td><td>The number of the step whose text is reported by this QuerySteps row.<br><br><table><tr><th>IF the step is ...</th><th>THEN the value is ...</th></tr><tr><td>not performed in parallel</td><td>the step number.</td></tr><tr><td>performed in parallel</td><td>the number of the main step.</td></tr></table></td></tr>
<tr><td>ParallelStepNum</td><td>The number of the parallel step whose text is reported by this QuerySteps row.<br><br><table><tr><th>IF the step is ...</th><th>THEN the value is ...</th></tr><tr><td>not performed in parallel</td><td>0.</td></tr><tr><td>performed in parallel</td><td>the number of the parallel step.</td></tr></table></td></tr>
<tr><td>StepText</td><td>Stores text describing the step.</td></tr>
<tr><td>RowType</td><td>Describes the type of detail this row characterizes.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>A</td><td>The row describes attributes for a query plan step beyond the first row for that step, which is coded with a RowType of G.<br><br>The additional attributes are described by the StepAttributeType and StepAttributeValue fields. The remaining fields in an A row type are set null.</td></tr><tr><td>G</td><td>The row describes the first row of step information for a particular step of the query plan.<br><br>All steps have one row of this type.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StepKind</td><td>Describes the kind of step characterized by this row.<br><br>
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>AB</td><td>Abort step.</td></tr>
<tr><td>BM</td><td>Bitmap step.</td></tr>
<tr><td>CE</td><td>Correlated inclusion merge join step.</td></tr>
<tr><td>CI</td><td>Correlated exclusion product join step.</td></tr>
<tr><td>CJ</td><td>Correlated inclusion 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 (hash join on the fly) 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.<br><br>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>
</table>
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StepKind<br>(continued)</td><td>
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>NJ</td><td>Nested join step.</td></tr>
<tr><td>OJ</td><td>Not exists join step.</td></tr>
<tr><td>PJ</td><td>Product join step.</td></tr>
<tr><td>RJ</td><td>Rowid join step.</td></tr>
<tr><td>SA</td><td>Sample step.</td></tr>
<tr><td>SO</td><td>Sort step.</td></tr>
<tr><td>SP</td><td>Spoil step.</td></tr>
<tr><td>SR</td><td>Single retrieve step.</td></tr>
<tr><td>ST</td><td>Statistics step.</td></tr>
<tr><td>SU</td><td>Sum retrieve step.</td></tr>
<tr><td>UP</td><td>Update step.</td></tr>
</table>
</td></tr>
<tr><td>ParallelKind</td><td>Describes whether the step can be done in parallel with its preceding step or not.
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>B</td><td>Step is a beginning parallel step.</td></tr>
<tr><td>E</td><td>Step is an ending parallel step.</td></tr>
<tr><td>P</td><td>Step is a parallel step and can be performed in parallel with the preceding step.</td></tr>
<tr><td>S</td><td>Step is a non-parallel step that is performed sequentially and cannot be performed in parallel with the preceding step.</td></tr>
</table>
</td></tr>
<tr><td>AMPUsage</td><td>Describes how AMPs are used to process the step.
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>A</td><td>Step is an all-AMPs operation.</td></tr>
<tr><td>G</td><td>Step operates on a group of AMPs.</td></tr>
<tr><td>O</td><td>Step is a one-AMP operation.</td></tr>
<tr><td>T</td><td>Step is a two-AMP operation.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>TriggerType</td><td>Describes triggering associated with this step.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>C</td><td>Cascaded triggering statement.</td></tr><tr><td>N</td><td>No triggering involved with this statement.</td></tr><tr><td>R</td><td>Triggered statement.</td></tr><tr><td>T</td><td>Triggering statement.</td></tr></table></td></tr>
<tr><td>Cost</td><td>Estimated cost of performing this step in milliseconds.<br><br>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.</td></tr>
<tr><td>MaxCost</td><td>Estimated worst case cost of performing this step.</td></tr>
<tr><td>SourceRelation1</td><td>Number of the principal relation source in this step.</td></tr>
<tr><td>SourceRelation2</td><td>Number of an additional relation source in this step.</td></tr>
<tr><td>TargetRelation1</td><td>Number of the spool file or table for which the step operation results or acts upon.</td></tr>
<tr><td>TargetRelation2</td><td>Number of an additional spool file, if one is required, to hold additional results of this step.</td></tr>
<tr><td>StepAttributeType</td><td>Indicates the attribute that is described by the row.<br><br><table><tr><th>Attribute Type</th><th>Attribute Value</th><th>Step Where Attribute Occurs</th></tr><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><tr><td>GlobalFlag</td><td><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Flag not set.<br><br>Intermediate aggregate results computed locally.</td></tr><tr><td>T</td><td>Flag set.<br><br>Intermediate aggregate results computed globally.</td></tr></table></td><td>Sum step.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StepAttributeType (continued)</td><td>
<table>
<tr><th>Attribute Type</th><th>Attribute Value</th><th>Step Where Attribute Occurs</th></tr>
<tr><td>GroupKey</td><td>Grouping field.</td><td>Sum step.</td></tr>
<tr><td>IndexNum</td><td>Index number.</td><td>• Abort<br>• BitMap<br>• Delete<br>• Single retrieve<br>Indicates whether the index was used.</td></tr>
<tr><td>JoinType</td><td>
<table>
<tr><th>Code</th><th>Description</th></tr>
<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>
</table>
</td><td>Join type in the join step.</td></tr>
<tr><td>Kind</td><td>
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>F</td><td>Fraction</td></tr>
<tr><td>I</td><td>Fixed</td></tr>
</table>
</td><td>Indicates whether the samples are specified as a fraction of rows or as an absolute number of rows in the sample step.</td></tr>
<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>
<table>
<tr><th>Code</th><th>Description</th></tr>
<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>
</table>
</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>
</table>
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>StepAttributeType (continued)</td><td>
<table>
<tr><th>Attribute Type</th><th>Attribute Value</th><th>Step Where Attribute Occurs</th></tr>
<tr><td>SMSKind</td><td>Intersect<br>Minus<br>Union</td><td>BitMap step indicating the set manipulation operation performed.</td></tr>
<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>
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>L</td><td>Load distribution optimization.</td></tr>
<tr><td>S</td><td>Single AMP optimization.</td></tr>
</table>
</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>
</table>
</td></tr>
<tr><td>StepAttributeValue</td><td>Indicates the value of the attribute described by the row.</td></tr>
<tr><td>LockType</td><td>Defines the severity of the lock specified by the query plan for this step.<br>See Chapter 8: “Locking and Transaction Processing” for information about lock severities.
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>A</td><td>ACCESS lock.</td></tr>
<tr><td>R</td><td>READ lock.</td></tr>
<tr><td>W</td><td>WRITE lock.</td></tr>
<tr><td>X</td><td>EXCLUSIVE lock.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>RowHashFlag</td><td>Indicates if the table is locked on a row hash.<br>Used to specify if the table is locked on a row hash or not. It can be Y or N.<br><table><tr><th>Code</th><th>Description</th></tr><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></table></td></tr>
<tr><td>NoWaitFlag</td><td>Indicates if the no wait option is set for the lock step.<br><table><tr><th>Code</th><th>Description</th></tr><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></table></td></tr>
<tr><td>Cardinality</td><td>The estimated number of output rows or affected rows estimated for this given step by the Optimizer.<br>This value is applicable only to steps that retrieve or modify rows.</td></tr>
<tr><td>IndexMaintCostEst</td><td>The cost estimated by the Index Wizard to maintain the indexes affected by this step.<br>This value is applicable only to steps that modify rows.</td></tr>
</table>

# Relation

## Function

Describes all table and spool files in the access plan for the captured query.

## Table Definition

The following CREATE TABLE statement defines the Relation table:

```
   CREATE TABLE Relation(
      RelationKey    INTEGER NOT NULL,
      QueryID        INTEGER NOT NULL,
      UDB_Key        INTEGER NOT NULL,
      Name           VARCHAR(30) CHARACTER SET UNICODE
                     NOT CASESPECIFIC NOT NULL,
      RelationID     INTEGER NOT NULL,
      RelationKind   CHARACTER(1) NOT NULL,
      SortInfo       CHARACTER(1),
      SortKind       CHARACTER(3),
      SortKey        VARCHAR(1024),
      GeogInfo       CHARACTER(1),
      Cached         CHARACTER(1) NOT NULL,
      SyncScan       CHARACTER(1) NOT NULL,
      Cardinality    REAL,
      Confidence     CHARACTER(1),
      MaxCardinality REAL,
      ViewName       VARCHAR(30) CHARACTER SET UNICODE
                     UPPERCASE NOT CASESPECIFIC,
      TableDDL       VARCHAR(20000) CHARACTER SET UNICODE
                     NOT CASESPECIFIC,
      TableName      VARCHAR(30) CHARACTER SET UNICODE
                     NOT CASESPECIFIC,
      PartitionInfo  CHARACTER(1) NOT NULL,
      Overflow       CHARACTER(1) CHARACTER SET LATIN,
      Complete       CHARACTER(1) CHARACTER SET LATIN,
      Version        SMALLINT)

PRIMARY INDEX (QueryID),
   UNIQUE INDEX (RelationKey);
```

## Attribute Definitions

The following table defines the Relation table attributes:

| Attribute | Definition |
|---|---|
| RelationKey | • Unique identifier for the relation within its database.<br>• USI for the Relation table. |

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>QueryID</td><td>• Unique identifier for the query generated by the system when the query plan is captured.<br>• NUPI for the Relation table.</td></tr>
<tr><td>UDB_Key</td><td>Identifier for the user or database containing the relation described by this row.</td></tr>
<tr><td>Name</td><td>Either of two things:<br>• Alias name of the table.<br>• Spool ID of a spool.</td></tr>
<tr><td>RelationID</td><td>Unique identifier for the relation within a certain database.</td></tr>
<tr><td>RelationKind</td><td>Distinguishes among derived tables, global temporary tables, hash indexes, join indexes, permanent tables, volatile tables, and spool files.<br><br>
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>D</td><td>Relation is a derived table.<br>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>
</table></td></tr>
<tr><td>SortInfo</td><td>Describes whether the relation is sorted or not.<br><br>
<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>F</td><td>The relation is not sorted.</td></tr>
<tr><td>T</td><td>The relation is sorted.</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>SortKind</td><td>The way the relation is sorted.<br>Only used when the value for SortInfo is T.<br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F1S</td><td>Field1 sort.</td></tr><tr><td>F1U</td><td>Field1 unique sort.</td></tr><tr><td>FHS</td><td>Field1 Hash sort.</td></tr><tr><td>FID</td><td>FieldID sort.</td></tr><tr><td>FHU</td><td>Field1 Hash unique sort.</td></tr><tr><td>HN1</td><td>Field1 Hash min1 sort.</td></tr><tr><td>HN2</td><td>Field1 Hash min2 sort.</td></tr><tr><td>HX1</td><td>Field1 Hash max1 sort.</td></tr><tr><td>HX2</td><td>Field1 Hash max2 sort.</td></tr><tr><td>JIS</td><td>JoinIndex sort.</td></tr><tr><td>MN1</td><td>Field1 min1 sort.</td></tr><tr><td>MN2</td><td>Field1 min2 sort.</td></tr><tr><td>MX1</td><td>Field1 max1 sort.</td></tr><tr><td>MX2</td><td>Field1 max2 sort.</td></tr><tr><td>RF1</td><td>Rowhash field1 sort.</td></tr><tr><td>RHR</td><td>RowHashRow sort.</td></tr><tr><td>RHS</td><td>Rowhash sort.</td></tr><tr><td>UF1</td><td>Unique field1 sort.</td></tr><tr><td>UNK</td><td>Unknown sort kind.</td></tr><tr><td>URS</td><td>Unique rowID sort.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>SortKey</td><td>A list of sort information strings composed of the concatenated database, table, and column names that make up the sort key. Only the first 1 024 characters are captured: any remaining characters are truncated.<br>The format for the SortKey list is the following:<br><pre>SortKey1, SortKey2, ..., SortKeyn</pre>There must be a SPACE character following each COMMA character in the list.<br>The format for the individual SortKey strings is one of the following:<br>• <code>database_name.table_name.column_name</code><br>• <code>spool_number.column_name</code><br>Spool numbers are used in place of database.table names whenever the table information is not available.</td></tr>
<tr><td>GeogInfo</td><td>Describes the configuration geography of the relation.<br><table><tr><th>Code</th><th>Description</th></tr><tr><td>D</td><td>Relation is duplicated on all AMPs.</td></tr><tr><td>L</td><td>Relation is built locally.</td></tr><tr><td>H</td><td>Relation is hash-distributed.</td></tr></table></td></tr>
<tr><td>Cached</td><td>Describes whether the relation is cached or not.<br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Relation is not cached.</td></tr><tr><td>T</td><td>Relation is cached.</td></tr></table></td></tr>
<tr><td>SyncScan</td><td>Describes whether a relation is eligible for synchronized scanning or not.<br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Relation is not eligible for synchronized scanning.</td></tr><tr><td>T</td><td>Relation is eligible for synchronized scanning.</td></tr></table></td></tr>
<tr><td>Cardinality</td><td>Estimated cardinality of the relation.</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>Confidence</td><td>Describes the confidence level for the estimated cardinality.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>H</td><td>High confidence.</td></tr><tr><td>I</td><td>Index join confidence.</td></tr><tr><td>L</td><td>Low confidence.</td></tr><tr><td>N</td><td>No confidence.</td></tr></table></td></tr>
<tr><td>MaxCardinality</td><td>Estimated maximum cardinality of the relation.</td></tr>
<tr><td>ViewName</td><td>Name of a view, if any, used by the query to access the relation.</td></tr>
<tr><td>TableDDL</td><td>The SQL DDL text for the captured relation.<br>If the text exceeds the upper limit of 20 000 characters, then it overflows to the QryRelX table. See “Overflow” on page 361 and “QryRelX” on page 343.</td></tr>
<tr><td>TableName</td><td>The non-aliased name of the relation.<br>Compare with “Name” on page 358.</td></tr>
<tr><td>PartitionInfo</td><td>Identifies whether a table or spool has a partitioned primary index.<br><br><table><tr><th>Code</th><th>Description</th></tr><tr><td>F</td><td>Relation does not have a partitioned primary index.</td></tr><tr><td>T</td><td>Relation has a partitioned primary index.</td></tr></table></td></tr>
<tr><td>Overflow</td><td>Specifies whether there is overflow query text stored in QryRelX or not.<br>The default value is F.<br><br><table><tr><th>Flag</th><th>Description</th></tr><tr><td>F</td><td>There is no overflow.</td></tr><tr><td>T</td><td>Overflow text is stored in a QryRelX table.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>Complete</td><td>Identifies whether Relation stores the complete relation DDL or a truncated version.<br><br><table><tr><th>Flag</th><th>Definition</th></tr><tr><td>F</td><td>Table DDL is truncated.<br><br>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 Reference: Data Definition Statements).<br><br>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.<br><br>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 343).</td></tr></table></td></tr>
<tr><td>Version</td><td>Stores the version number of the table at the time the plan was captured.<br><br>Used to ensure that any changes to the schema information captured for analysis are handled correctly.</td></tr>
</table>

# SeqNumber

## Function

Generates sequential alternate keys to be used by other QCD tables.

Each column in the table (except QCDVersion, DemographicsID, and StatisticsID) is initialized to 1 and then incremented after each use.

## Table Definition

The following CREATE TABLE statement defines the SeqNumber table:

```
CREATE TABLE SeqNumber (
 PIndex           INTEGER NOT NULL,
 WorkLoadID       INTEGER NOT NULL,
 MachConfigID     INTEGER NOT NULL,
 QueryID          INTEGER NOT NULL,
 UDB_KEY          INTEGER NOT NULL,
 StepID           INTEGER NOT NULL,
 RelationKey      INTEGER NOT NULL,
 PredicateID      INTEGER NOT NULL,
 RecommendationID INTEGER NOT NULL,
 QCDVersion       VARCHAR(30) CHARACTER SET LATIN
                  NOT CASESPECIFIC,
 DemographicsID   INTEGER NOT NULL,
 StatisticsID     INTEGER NOT NULL
)
PRIMARY INDEX (PIndex)
UNIQUE INDEX (RelationKey);
```

## Table Initialization Statement

SeqNumber is initialized by the following INSERT statement:

```
INSERT INTO SeqNumber VALUES(1,1,1,1,1,1,1,1,1,'QCF03.00.00',2,2);
```

## Attribute Definitions

The following table defines the SeqNumber table attributes:

| Attribute | Definition |
|---|---|
| PIndex | An artificial column used as the NUPI for this table. |
| WorkLoadID | Unique identifier for the workload. |
| MachConfigID | Unique identifier for the configuration of a particular hardware configuration stored in QCD. |
| QueryID | Unique identifier for the query. |

| Attribute | Definition |
|---|---|
| UDB_KEY | Unique identifier on MachConfigID for the user or database that performed the query. |
| StepID | Unique identifier for a particular AMP step in the query. |
| RelationKey | Unique identifier for a table, derived table, or spool file used in the query. |
| PredicateID | Unique identifier for the predicate used in the query. |
| RecommendationID | Unique identifier for a particular set of index recommendations. |
| QCDVersion | Describes the version of QCD currently running. |
| DemographicsID | Identifies whether the demographics were captured on the current system or imported from another system.<br>Initially set to 2. |
| StatisticsID | Identifies whether the statistics were captured by a COLLECT STATISTICS (QCD Form) or INSERT EXPLAIN statement.<br>Initially set to 2. |

# StatsRecs

## Function

Each row set contains a COLLECT STATISTICS statement and related information for collecting the recommended statistics generated by a DUMP EXPLAIN or INSERT EXPLAIN statement specified with a CHECK STATISTICS clause.

## Table Definition

The following CREATE TABLE statement defines the StatsRecs table:

```
CREATE SET TABLE StatsRecs (
 QueryID      INTEGER NOT NULL,
 StatsID      INTEGER NOT NULL,
 DatabaseName VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC
              NOT NULL,
 TableName    VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC
              NOT NULL,
 FieldName    VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC
              NOT NULL,
 Level        INTEGER NOT NULL,
 StatsDDL     VARCHAR(2500) CHARACTER SET UNICODE NOT CASESPECIFIC)
PRIMARY INDEX (QueryID);
```

## Attribute Definitions

The following table defines the StatsRecs table attributes:

| Attribute | Description |
|---|---|
| QueryID | • Unique identifier for the query to which the recommendation applies.<br>• The NUPI for the StatsRecs table. |
| StatsID | Uniquely identifies a statistics collection recommendation for QueryID.<br>There are multiple StatsID values corresponding to a single QueryID for multicolumn statistics recommendations.<br>In other words, a set of rows having the same value in StatsId represents all the columns for a given recommendation. |
| DatabaseName | Name of the containing database for TableName. |
| TableName | Name of the table in which FieldName is defined. |
| FieldName | Name of the column in the statistics recommendation. |
| FieldID | Unique identifier for FieldName within TableName. |

<table>
<tr><th>Attribute</th><th>Description</th></tr>
<tr><td>Level</td><td>A representation of the conviction the Optimizer has in the usefulness of the statistics recommendations it has generated for this query-column set combination.<br><br>The level is determined by a number of factors, including the following:<br><br>• The number of columns in the recommendation.<br>• Whether the recommendations are for collecting single column statistics or multicolumn statistics.<br><br>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.<br><br>You can also aggregate levels to rank different recommendations.<br><br>
<table>
<tr><th>Level</th><th>Description</th></tr>
<tr><td>1</td><td>The primary recommendation.<br><br>This recommendation is more likely to help the Optimizer to generate the least costly plan possible than any others.<br><br>Recommendations to collect single-column and multicolumn statistics with all the fields are considered to be primary.</td></tr>
<tr><td>2</td><td>An optional or alternative recommendation for the multicolumn statistics.<br><br>Optional recommendations provide an alternative to collecting statistics on multiple non-index 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.<br><br>Optional recommendations are provided based on information collected from usable indexes and usable fields from the table described by the TableName column.</td></tr>
</table>
</td></tr>
<tr><td>StatsDDL</td><td>The DDL text of the COLLECT STATISTICS statement used to populate this row set of the table.<br><br>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 field for the other rows is null.</td></tr>
</table>

# TableStatistics

## Function

Captures the statistics on a data sample for all the columns that are possible index candidates in the specified query.

TableStatistics also captures the detail statistics if you perform the SQL COLLECT STATISTICS (QCD Form) statement.

## Table Definition

The following CREATE TABLE statement defines the TableStatistics table:

```
CREATE TABLE TableStatistics (
 MachineName    VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 TableName      VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 DatabaseName   VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 IndexName      VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC,
 ColumnName     VARCHAR(30) CHARACTER SET UNICODE
                UPPERCASE NOT CASESPECIFIC NOT NULL,
 CollectedTime  TIMESTAMP(6) NOT NULL,
 SamplePercent  FLOAT NOT NULL,
 IndexType      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:

| Attribute | Definition |
|---|---|
| MachineName | The name of the system to which the table belongs. |
| TableName | Name of the table for which the row defines statistics. |
| DatabaseName | Name of the containing database or user for TableName. |

<table>
<tr><th>Attribute</th><th>Definition</th></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.<br><br><table><tr><th>Code</th><th>Description</th></tr><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></table></td></tr>
<tr><td>IndexID</td><td>The unique identifier for the index to which the statistics represented by this row pertain.<br><br><table><tr><th>IF IndexType has this value …</th><th>THEN IndexID is …</th></tr><tr><td>N</td><td>null.</td></tr><tr><td>P</td><td>the ID value generated by scanning the existing index set for the table and incrementing the largest value.</td></tr><tr><td>Y</td><td>the ID value assigned by the system.</td></tr></table></td></tr>
<tr><td>StatisticsInfo</td><td>Contains the column or index statistics.</td></tr>
<tr><td>DataType</td><td>Indicates the data type of the column. The value is the same that is returned in the datatype field in the PrepInfo CLIv2 parcel.</td></tr>
<tr><td>DataLength</td><td>The maximum length of the column in bytes.</td></tr>
<tr><td>Attributes</td><td>Set null and reserved for future use.</td></tr>
<tr><td>ModifiedTime</td><td>If the statistics have been modified, indicates the timestamp value when the last modification occurred.</td></tr>
</table>

| Attribute | Definition |
|---|---|
| ModifiedStats | Contains user-modified column or index statistics.<br>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.<br>You create the statistics in ModifiedStats using the "what if" features of the Teradata Index Wizard utility. |
| QueryID | The unique ID of the query.<br>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.<br>Statistics are recorded in the row corresponding to FieldPosition 1 only. The value for other field positions are set null. |
| StatisticsID | Set to 1 if the demographics are captured by a COLLECT STATISTICS (QCD Form) or INSERT EXPLAIN statement.<br>StatisticsID has different values if the demographics are imported rather than captured directly. |

# User_Database

## Function

Describes User and Database identifiers to capture the identity of the user who submitted the query, the names of databases used in the query plan, and so on.

## Table Definition

The following CREATE TABLE statement defines the User_Database table:

```
CREATE TABLE User_Database(
 UDB_Key     INTEGER NOT NULL,
 UDB_ID      INTEGER NOT NULL,
 MachineName VARCHAR(30) CHARACTER SET UNICODE
             NOT CASESPECIFIC NOT NULL,
 UDB_Name    VARCHAR(30) CHARACTER SET UNICODE
             NOT CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX PK_UDB_KEY ( UDB_Key );
```

## Attribute Definitions

The following table defines the User_Database table attributes:

| Attribute | Definition |
|---|---|
| UDB_Key | • Unique identifier for the user or database.<br>• UPI for the table. |
| UDB_ID | Unique identifier for the user or database on the system on which the captured query was performed. |
| MachineName | Name of the production system on which the database identified by UDB_ID exists. |
| UDB_Name | Name of the user or database identified by UDB_ID. |

# UserRemarks

## Function

Stores user comments about captured plans written using either the TSET or Visual EXPLAIN client utilities.

## Table Definition

The following CREATE TABLE statement defines the UserRemarks table:

```
CREATE TABLE UserRemarks(
 QueryID     INTEGER,
 StepID      INTEGER,
 WorkloadID  INTEGER,
 RowType     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:

| Attribute | Definition |
|---|---|
| QueryID | The ID of the query about which remarks are being saved.<br>Partial NUPI for the table. |
| StepID | The ID of the step about which remarks are being saved. |
| WorkloadID | The ID of the workload to which the query belongs. |
| RowType | Specifies which utility collected the remarks and whether they were saved or generated. |
| SeqNumber | A system-generated number (1 for the first row, 2 for second, and so on) to indicate the sequential order of remarks. |

| Code | Description |
|---|---|
| T | Comments generated automatically by the TSET utility during import. |
| V | Comments saved by the Visual EXPLAIN utility. |

| Attribute | Definition |
|---|---|
| UpdTime | The timestamp when the remark was last updated. |
| UserName | The UserName of the user who last updated the row.<br>Partial NUPI for the table. |
| Remarks | The remarks of the step or query. |

# ViewTable

## Function

Captures the DDL for any views used in a query.

## Table Definition

The following CREATE TABLE statement defines ViewTable:

```
CREATE SET TABLE ViewTable(
 QueryID   INTEGER NOT NULL,
 ViewName  VARCHAR(30) CHARACTER SET UNICODE
           UPPERCASE NOT CASESPECIFIC NOT NULL,
 DBName    VARCHAR(30) CHARACTER SET UNICODE
           UPPERCASE NOT CASESPECIFIC NOT NULL,
 ViewText  VARCHAR(30000) CHARACTER SET UNICODE
           NOT CASESPECIFIC NOT NULL,
 SeqNumber BYTEINT NOT NULL,
 Complete  CHARACTER(1))
PRIMARY INDEX QueryID_ViewName_DBName (QueryID,ViewName,DBName);
```

## Attribute Definitions

The following table defines the ViewTable attributes:

| Attribute | Definition |
|---|---|
| QueryID | • Unique identifier for the query.<br>• Partial NUPI for the table. |
| ViewName | • Name of the view.<br>• Partial NUPI for the table. |
| DBName | • Name of the database.<br>• Partial NUPI for the table. |

<table>
<tr><th>Attribute</th><th>Definition</th></tr>
<tr><td>ViewText</td><td>Stores the DDL view text.<br><br><table><tr><th>IF ViewText is ...</th><th>THEN SeqNumber is set to ...</th></tr><tr><td><= 30 000 characters</td><td>1</td></tr><tr><td>> 30 000 characters</td><td>1 + (1-n), where n is the identifier for the fragment of ViewText stored in the row following the first fragment</td></tr></table><br>When ViewText > 30 000 characters, only the first 30 000 are stored and the remaining text is stored in overflow rows within ViewTable. The overflow rows are identified by their SeqNumber value.</td></tr>
<tr><td>SeqNumber</td><td>The sequence of ViewText stored in this row.</td></tr>
<tr><td>Complete</td><td>Identifies whether ViewTable stores the complete view DDL or a truncated version.<br><br><table><tr><th>Flag</th><th>Definition</th></tr><tr><td>F</td><td>ViewText DDL is truncated.<br><br>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.<br><br>See SQL Reference: Data Definition Statements.</td></tr><tr><td>T</td><td>Full ViewText DDL is stored.<br><br>Overflow ViewText DDL is stored in additional rows in ViewTable. Individual overflow rows are identified by their respective SeqNumber.</td></tr></table></td></tr>
</table>

# Workload

## Function

Workload table has one entry for each workload defined in the database.

## Table Definition

The following CREATE TABLE statement defines the Workload table:

```
CREATE TABLE Workload(
 WorkloadID       INTEGER NOT NULL,
 WorkloadName     VARCHAR(30) CHARACTER SET UNICODE
                  UPPERCASE NOT CASESPECIFIC NOT NULL,
 CreatedTimeStamp TIMESTAMP(6) NOT NULL,
 LastModified     TIMESTAMP(6) NOT NULL
 CreatorName      VARCHAR(30) CHARACTER SET UNICODE
                  NOT CASESPECIFIC)
UNIQUE PRIMARY INDEX(WorkLoadName),
UNIQUE INDEX(WorkLoadID);
```

## Attribute Definitions

The following table defines the Workload table attributes:

| Attribute | Definition |
|---|---|
| WorkloadID | • Internally generated unique (within the database) identifier for the workload.<br>• USI for the table. |
| WorkloadName | • User-specified unique workload name.<br>• UPI for the table. |
| CreatedTimeStamp | Timestamp when the workload was created. |
| LastModified | Timestamp when the workload was last modified. |
| CreatorName | Name of the user that created the workload. |

# WorkloadQueries

## Function

Captures the queries that belong to the workload named by WorkloadID.

## Table Definition

The following CREATE TABLE statement defines the WorkloadQueries table:

```
CREATE TABLE WorkloadQueries(
 WorkloadID           INTEGER NOT NULL,
 QueryID              INTEGER NOT NULL,
 Frequency            INTEGER NOT NULL)
PRIMARY INDEX(WorkLoadID)
UNIQUE INDEX(WorkloadID, QueryID);
```

## Attribute Definitions

The following table defines the WorkloadQueries table attributes:

| Attribute | Definition |
|---|---|
| WorkloadID | • The workload ID that identifies the data in the row.<br>• NUPI for the table.<br>• Partial USI for the table. |
| QueryID | • Unique qualifier for a query. Used to map it to the workload.<br>• Partial USI for the table. |
| Frequency | Indicates the number of times the query is typically performed in the workload. |

# WorkloadStatus

## Function

Maintains the status of workloads in the QCD.

This table is maintained by the Teradata Index Wizard.

## Table Definition

The following CREATE TABLE statement defines the WorkloadStatus table:

```
CREATE TABLE WorkloadStatus (
 WorkloadID          INTEGER NOT NULL,
 RecommendationID    INTEGER,
 IndexNameTag        VARCHAR(30) CHARACTER SET UNICODE
                     NOT CASESPECIFIC,
 ValidatedSystem     VARCHAR(30) CHARACTER SET UNICODE
                     NOT CASESPECIFIC,
 ValidatedTimeStamp TIMESTAMP(6),
 ValidatedQCD        VARCHAR(30) CHARACTER SET UNICODE
                     NOT CASESPECIFIC,
 RecosAppliedSystem VARCHAR(30) CHARACTER SET UNICODE
                     NOT CASESPECIFIC,
 Remarks             VARCHAR(20000) CHARACTER SET UNICODE
                     NOT CASESPECIFIC)
UNIQUE PRIMARY INDEX (WorkloadID ,IndexNameTag);
```

## Attribute Definitions

The following table defines the WorkloadStatus table attributes:

| Attribute | Definition |
|---|---|
| WorkloadID | • Unique identifier for the workload.<br>• Partial UPI for the table. |
| RecommendationID | Unique identifier for the set of indexes recommended for this workload after an index analysis. |
| IndexNameTag | • Name of the index recommendation as specified in the INITIATE INDEX ANALYSIS statement (see *SQL Reference: Data Definition Statements*).<br>• Partial UPI for the table. |
| ValidatedSystem | Name of the system on which index recommendations are last validated by the Teradata Index Wizard. |
| ValidatedTimeStamp | Timestamp of the index validation. |

| Attribute | Definition |
|---|---|
| ValidatedQCD | Name of the QCD on which the validation is performed by the Teradata Index Wizard. |
| RecosAppliedSystem | Name of the system on which index recommendations are last created by the Teradata Index Wizard. |
| Remarks | Free form textual remarks about the workload. |

# CHAPTER 6 Database Foundations for the Teradata Index Wizard

This chapter describes the server-based architecture for the Teradata Index Wizard utility.

The Teradata Index Wizard helps you to reengineer existing databases by providing a method to propose optimal sets of secondary and single-table join indexes[1] to support particular SQL query workloads. This process permits you to analyze the potential performance enhancing effects of various join and secondary indexes on statistical representations of live data from your data warehouse.

In most cases, the data was probably not available at the time you performed the physical design for the database, so the effects of the join and secondary indexes you defined to support your query workloads were guessed at rather than tested empirically.

Loosely speaking, the Teradata Index Wizard tool permits you to revisit the Activity Transaction Modeling (ATM) phase of the transition from logical design to physical design (see *Database Design* for more information on ATM).

Information in the present chapter includes the following topics:



Detailed information about related tools and procedures is found in the following sources:

- Chapter 5: "Query Capture Facility"
- Chapter 7: "Target Level Emulation"
- "COLLECT DEMOGRAPHICS" in *SQL Reference: Data Definition Statements*.
- "COLLECT STATISTICS (QCD Form)" in *SQL Reference: Data Definition Statements*.
- "INITIATE INDEX ANALYSIS" in *SQL Reference: Data Definition Statements*.
- "DUMP EXPLAIN" in *SQL Reference: Data Definition Statements*.
- "INSERT EXPLAIN" in *SQL Reference: Data Definition Statements*.
- "RESTART INDEX ANALYSIS" in *SQL Reference: Data Definition Statements*.
- *Teradata Index Wizard User Guide*.
- Online help for Teradata Index Wizard utility.

1. The Teradata Index Wizard does not make recommendations for multitable join indexes or hash indexes.

- *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 join and secondary index assignment. The analysis can produce recommendations to delete, as well as to add, indexes to the existing physical design.

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 Selection

The following table outlines the general procedure followed in performing index analysis using the Teradata Index Wizard:

| Step | Activity Name | Activity Description |
|---|---|---|
| 1 | Identify workload. | Identify a set of SQL statements that constitute a workload.<br>The following repositories are potential sources for identifying workload components:<br>• Database Query Log (see *SQL Reference: Data Definition Statements*).<br>• Query Capture Database (see Chapter 5: "Query Capture Facility").<br>See "Workload Identification" on page 383 for details. |
| 2 | Define workload. | Perform the AddWorkload macro to define a new workload in a query capture database.<br>See "Workload Definition" on page 385 for details. |
| 3 | Analyze indexes. | Perform an index analysis on the defined workload to produce a set of index recommendations.<br>See "Index Analysis" on page 388 for details. |
| 4 | Validate indexes. | Validate the recommended indexes on your production system.<br>This is an optional step in the process.<br>See "Index Validation" on page 392 and *SQL Reference: Data Definition Statements* for details. |
| 5 | Create indexes. | Apply the index recommendations to your production system.<br>This is an optional step in the process.<br>See "Index Application" on page 394 for details. |

## Constraints on the Output of the Teradata Index Wizard

The following constraints and restrictions apply to the use of the Index Wizard:

- By default, any query search conditions that default to full table scans cannot benefit from indexes; therefore, the Teradata Index Wizard does not generate index recommendations for such queries.
- If you generate index recommendations on a test system without emulating the production environment, then the index recommendations might not produce an optimal result when applied to the production system.

  You should always use Target Level Emulation (see Chapter 7: "Target Level Emulation") to emulate production environments when performing index analyses on a test system.
- The Teradata Index Wizard produces recommendations for secondary and single-table join indexes only.

1 A workload is identified. See "Workload Identification" on page 383.

2 Query plans that characterize the workload are collected in the QCD. This is referred to as defining the workload. See "Workload Definition" on page 385.

3 Index analysis and validation are performed on the QCD data. See "Index Analysis" on page 388 and "Index Validation" on page 392.

   a The Teradata Index Wizard begins the analysis by invoking the SQL INITIATE INDEX ANALYSIS statement (see "INITIATE INDEX ANALYSIS" in the chapter "SQL Data Manipulation Language Statement Syntax" in *SQL Reference: Data Manipulation Statements*).

   b INITIATE INDEX ANALYSIS selects candidate indexes for the submitted workload and submits each of them with the workload to the Optimizer.

| IF the optimizer generates a query plan that … | THEN the Teradata Index Wizard … |
|---|---|
| does not use the candidate index | drops that index from consideration as a recommendation. |
| uses the candidate index | keeps that index for recommendation. |

   See "How Index Analyses Are Done" on page 388 for details of this process.

   c End of sub-process.

4 End of process.

## What Not To Include In A Workload

Workload analysis is based on Optimizer costing of the various SQL statements defined for a workload. Because the only DML statement the Optimizer costs is SELECT, inclusion of other SQL DML statements such as DELETE, INSERT, MERGE, and UPDATE has no effect on the index recommendations the Index Wizard makes. While you can add non-costed DML statements to a workload definition, they do not contribute in any way to the index analysis.

# Workload Identification

## Introduction

The objective of workload identification is to isolate the set of queries, called a workload, you want to analyze for the possibility of enhancing their performance by redefining the secondary indexes defined on the tables they access.

There are several methods for identifying workloads, including these:

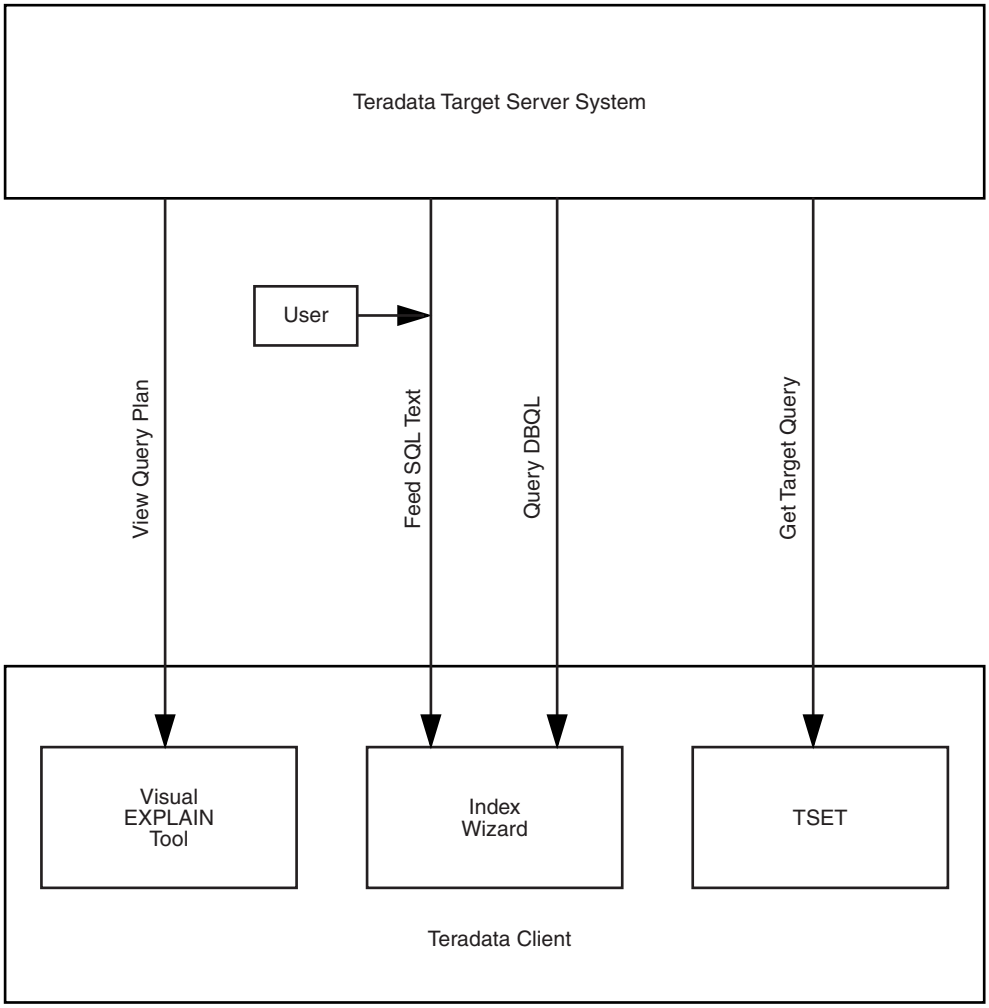
- Ad hoc qualitative assessment of the performance of some queries. This is the "something is not right here" approach.
- Sift through the following data stores in search of candidate queries for performance enhancement:
  - The Database Query Log using the Teradata Index Wizard.
  - The Query Capture Database using tools like Visual EXPLAIN or Teradata System Emulation Tool (TSET).

After determining the set of queries to be analyzed, you can identify the workload using the Teradata Index Wizard tool.

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:




GG02A006

| Key | |
|---|---|
| **Input Option** | **Description** |
| View query plan | You can invoke the Index Wizard from a query plan viewed by means of the Visual EXPLAIN tool.<br>The query used for index analysis is the active plan in Visual EXPLAIN at the time the Index Wizard is invoked. |
| Feed SQL text | You can directly type the text for the queries that are to make up a workload or you can load them from the server. |
| Query DBQL | You can fetch logged Database Query Log SQL text for index analysis by means of the Index Wizard. |
| Get target query | You can export a set of queries for analysis using TSET. |

# Workload Definition

## Introduction

The objective of workload definition is to capture the set of query plans associated with the queries identified in the workload identification phase and define them to the Teradata Index Wizard.

Workloads are defined to the Teradata Index Wizard using the query plans for the queries identified as a workload. These query plans must first be captured in the QCD using the INSERT EXPLAIN WITH STATISTICS statement (see *SQL Reference: Data Definition Statements*).

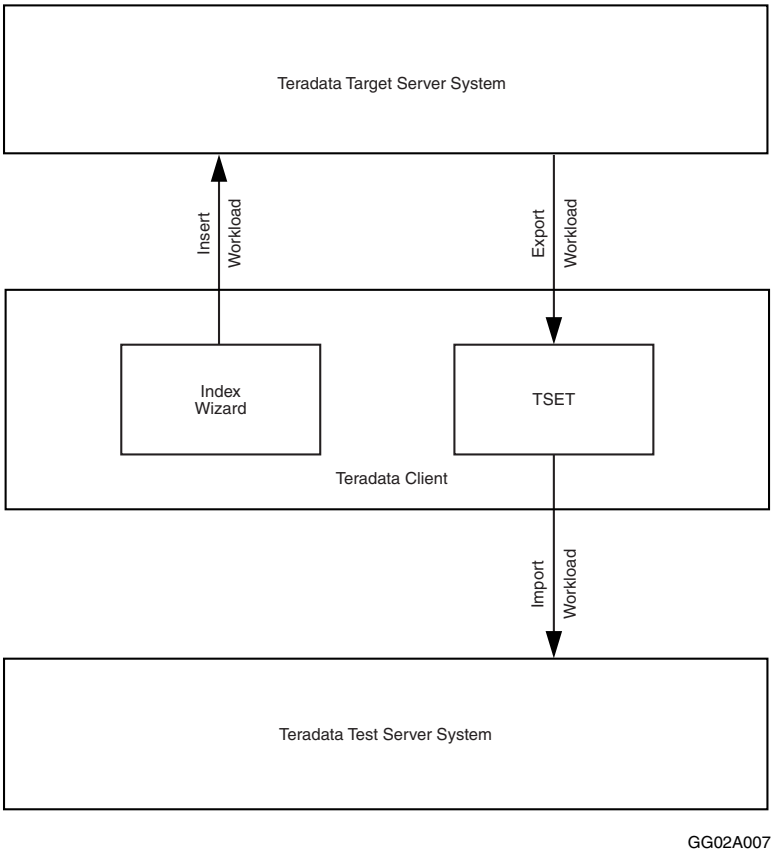
See *Teradata Index Wizard User Guide* for details about this process.

## Workload Definition Macros

The Teradata Database supplies a QCD with a set of macros to help you define workloads, modify those workloads, and delete them when they are no longer useful. These macros are normally invoked using the Teradata Index Wizard utility.

## Workload Definition Process

The process of defining SQL workloads for subsequent index analysis is indicated by the following graphic:




| Key | |
|---|---|
| **Action** | **Description** |
| Insert workload | The workload is defined in the specified QCD on the production system. |
| Export workload[1] | Once it has been created, the workload can be exported from the production system to the test system using TSET. |
| Import workload | In the second stage of this process, the workload is imported to the test system from the production system using either TSET or the Teradata Index Wizard. |

1. 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 7: “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:

| ATM Activity | Workload Definition Activity |
|---|---|
| Identify and define attribute domains and constraints for physical columns. | Define workload on tables and databases whose definitions exist in the data dictionary of a production environment.<br>Because they already exist, they need not be identified. |
| Identify and model database applications. | Not applicable. |
| Model application processing activities to include their transactions and run frequencies. | Submit the SQL requests as transactions and run frequencies using the INSERT EXPLAIN statement.<br>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.<br>The Index Wizard client interface helps you to derive the run frequency information as the total count of the entries for the same queries logged in DBQL. |
| Model transactions to identify the tables used and the columns required for value or join access and estimating cardinalities | The workload analysis includes this activity.<br>For the SQL requests submitted as workload, the system assimilates the information and stores it in a Query Capture Database. |
| Summarize value and join access information across all transactions. | The information stored in QCD as part of workload analysis is used to summarize the information. |
| Compile a preliminary set of data demographics by estimating table cardinalities and value distributions and assigning change ratings. | The workload analysis includes this activity.<br>For the tables referenced in the SQL requests submitted as workload, the system assimilates the demographics information and stores it in the QCD. The processing includes a COLLECT STATISTICS (QCD Form) statement performed to obtain the information on columns identified as index candidates. |

# Index Analysis

## Introduction

The objective of Index Analysis is to recommend a set of CREATE INDEX or DROP INDEX statements (or both) on the different tables referenced in the workload that provides the optimal improvement in the response time of that workload.

See *Teradata Index Wizard User Guide* for details about this process.

## How Index 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 workload proposed index list are set null.

2 The workload candidate search space is built as follows:

| Stage | Process |
|---|---|
| 1 | For each SQL statement in the workload: |

| Stage | Process |
|---|---|
| 1 | Regenerate the query plan.<br>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.<br>To perform this task, the Index Search Engine analyzes the statement index candidate list and removes the least promising indexes, producing the resultant statement proposed index list. |
| 3 | Add the statement proposed index list to the workload candidate search space. |
| 4 | Determine if any table constraints have been violated. |

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1<br>(cont.)</td><td>
<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>4<br>(cont.)</td><td>
<table>
<tr><th>IF a table constraint violation is ...</th><th>THEN ...</th></tr>
<tr><td>found</td><td>1 Invoke the Index Search Engine to filter candidate indexes from the workload candidate search space.<br>2 Build a new workload proposed index list.<br>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.<br>Go to Stage 2.1.4.</td></tr>
<tr><td>not found</td><td>Go to Stage 2.2.</td></tr>
</table>
</td></tr>
<tr><td>5</td><td>Set workload candidate search space to workload proposed index list.</td></tr>
</table>
</td></tr>
<tr><td>2</td><td>
<table>
<tr><th>IF a checkpoint is ...</th><th>AND the specified checkpoint frequency ...</th><th>THEN ...</th></tr>
<tr><td>specified</td><td>is hit</td><td>write the workload proposed index list to the AnalysisLog table.</td></tr>
<tr><td>not specified</td><td></td><td>Go to Stage 3.0.</td></tr>
</table>
</td></tr>
</table>

3 Filter the workload proposed index list using the best indexes analyzed so far to produce the index recommendations.

4 Delete any remaining AnalysisLog entries.

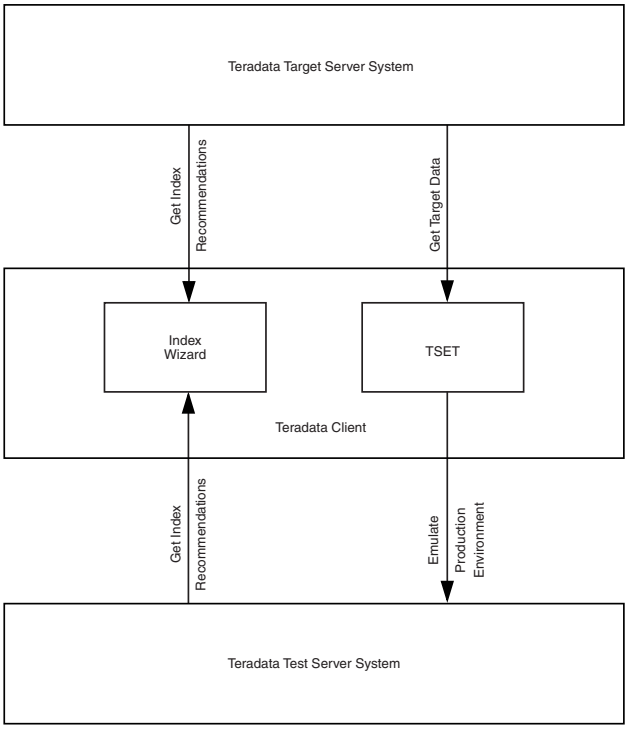
You can specify various index analysis parameters to control the choice of candidate indexes using the various boundary options of the INITIATE INDEX ANALYSIS statement (see *SQL Reference: Data Definition Statements* for details) or the Teradata Index Wizard (see *Teradata Index Wizard User Guide*). 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 just recommended indexes: they can also include recommendations to drop existing indexes. Index recommendations never include volatile tables that are accessed by the SQL statements making up the workloads.

## Index Analysis Process

The process of analyzing statistical data in order to develop a set of recommended indexes is indicated by the following graphic.

Note that if the analysis is performed on a production system, the two Get Index Recommendations steps collapse into a single step.




| Key | |
|---|---|
| **Activity** | **Description** |
| Get target data | Use TSET (see *Teradata System Emulation Tool User Guide*) to collect production system data such as environmental cost parameters and random AMP statistical samples. |
| Emulate the production environment | Use the Teradata Index Wizard client interface (see *Teradata Index Wizard User Guide*) to set up the production system data at session level on a test system so it can emulate the production environment. |
| Get index recommendations | Perform the index analysis on the test system.<br>Use the INITIATE INDEX ANALYSIS statement (see *SQL Reference: Data Definition Statements*) or Teradata Index Wizard to submit the SQL statement set to be analyzed.<br>INITIATE INDEX ANALYSIS collects the index recommendations generated and saves them in the specified QCD.<br>Note that index recommendations also include relevant CREATE INDEX and DROP INDEX statements plus the associated COLLECT STATISTICS statements for any recommended indexes. |

| Key | |
|---|---|
| **Activity** | **Description** |
| Get index recommendations | When this procedure is performed on a production system, this step is identical to the previous step. |

# Index Validation

## Introduction

You can validate the indexes recommended by the Index Wizard by comparing query response latencies with the same workload against the existing index set against response latencies for the same query set and workload using the recommended index set. This is an optional, but recommended, step in the database query analysis process.

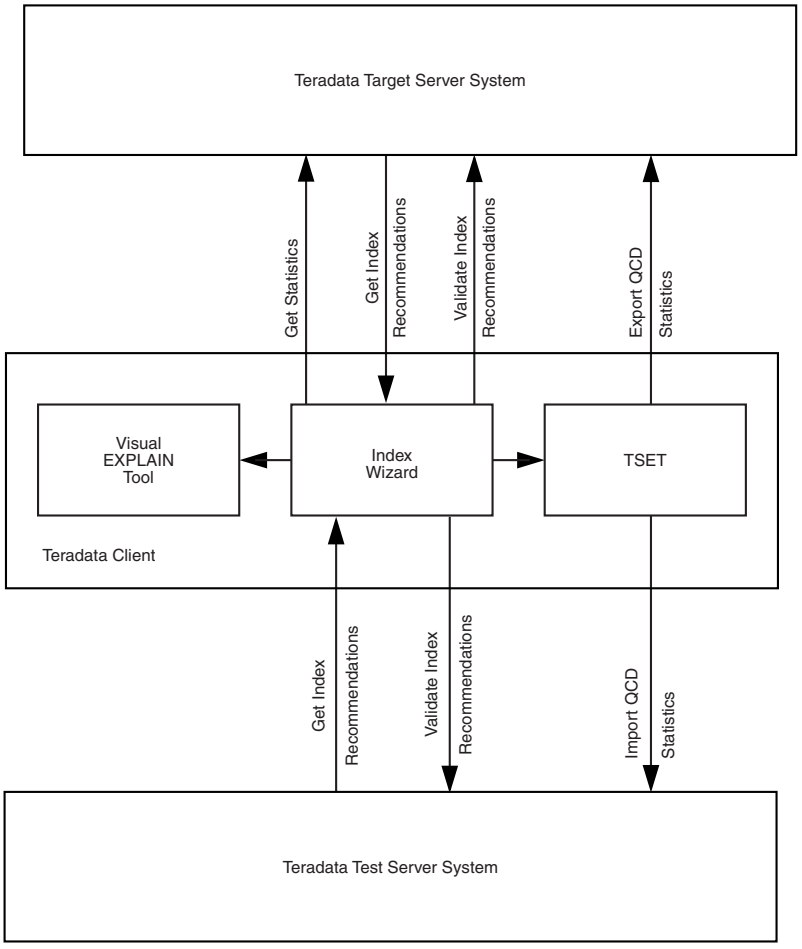
You can perform index validation either on a production system or on a test system, as indicated by the graphic.

See *SQL Reference: Data Definition Statements* for additional information about validating indexes recommended by the Teradata Index Wizard.

See *Teradata Index Wizard User Guide* for details about this process.

## Index Validation Process

The process of validating recommended indexes is indicated by the following graphic:



<table>
<tr><th colspan="2">Key</th></tr>
<tr><th>Activity</th><th>Description</th></tr>
<tr><td>Get index recommendations</td><td>The Index Wizard utility retrieves the index recommendations from the QCD.</td></tr>
<tr><td>Get statistics</td><td>The SQL COLLECT STATISTICS (QCD Form) statement gathers sampled statistics from the production system for use in the validation process.<br><br>See SQL Reference: Data Definition Statements for more information.</td></tr>
<tr><td>• Validate recommendations<br>• Import/export QCD statistics</td><td>The actions required to complete these activities depend on whether the analysis is performed on a production system or a test system.<br><br>
<table>
<tr><th>FOR this type of system ...</th><th>The following actions must be taken ...</th></tr>
<tr><td>production</td><td>Rerun the entire workload after the previous activity in Index Validation has been performed.<br><br>The recommendations are made available to the Optimizer for generating the query plan.</td></tr>
<tr><td>test</td><td>in the following order:<br><br>1 Export the statistics from the previous stage of Index Validation from the production system using TSET.<br>2 Import the exported statistics from the production system to the test system using TSET.<br>3 Rerun the entire workload.<br>The recommendations are made available to the test system Optimizer for generating the query plan.</td></tr>
</table>
</td></tr>
<tr><td>Compare the query plans with and without implementing the recommended indexes</td><td>Use the client Visual EXPLAIN tool (see Teradata Visual Explain User Guide) to compare the query plans with and without incorporating the index recommendations to evaluate the extent of query processing enhancement produced by the recommendations.</td></tr>
</table>

# Index Application

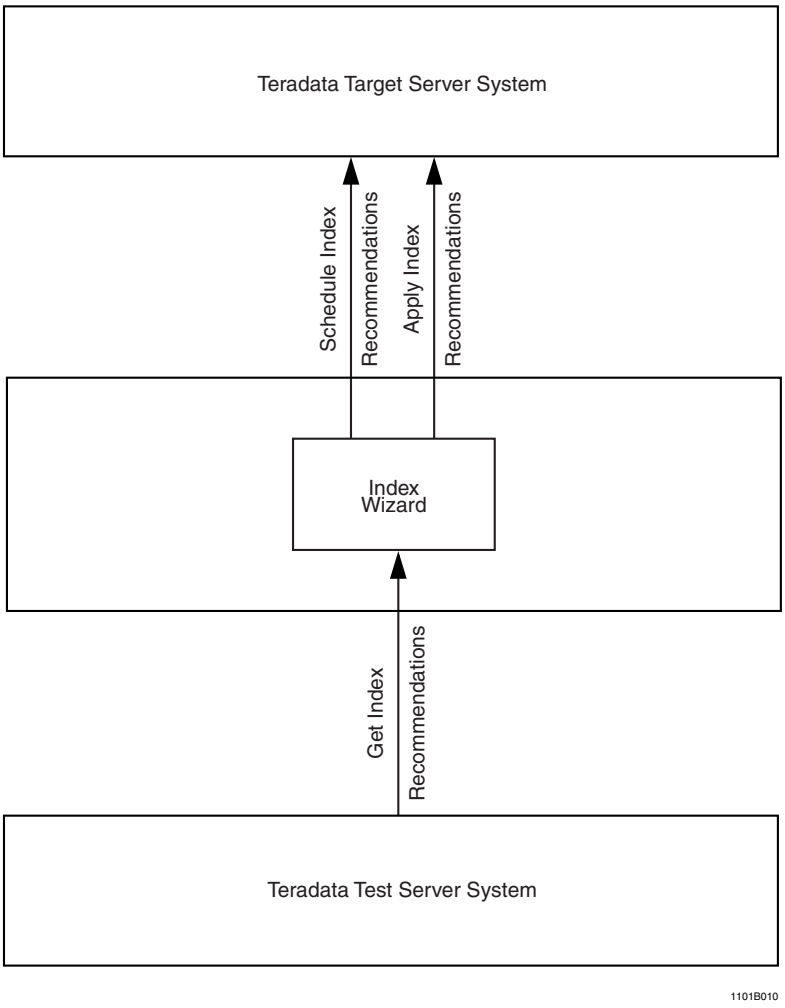
## Introduction

At this stage of the database query analysis, you apply the indexes recommended by the Index Analysis stage to the production database.

See *Teradata Index Wizard User Guide* for details about this process.

## Index Creation Process

The process of applying the recommended indexes to the production system is indicated by the following graphic:



| Key | |
|---|---|
| **Activity** | **Description** |
| Get index recommendations | Use the Teradata Index Wizard tool (see *Teradata Index Wizard User Guide*) to retrieve index recommendations from the QCD. |
| Schedule application of index recommendations | Use the Index Wizard tool to schedule the application of the index recommendations. |
| Index recommendations | Perform the generated DDL statements to create the recommended index set on the production system.<br>You can do this directly using the Teradata Index Wizard tool or you can schedule it for later using the Teradata Query Scheduler. |

# CHAPTER 7 Target Level Emulation

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 DBSControl 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 106.

For additional information about random AMP sampling, see “Random AMP Sampling” on page 76.

1. See the footnote to “What Is a Query Optimizer?” on page 35 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 DBSControl 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 DBSControl 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.[2] 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 5 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 DBSControl 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 Reference: Data Definition Statements* for definitions of what these terms mean in the context of Target Level Emulation).

  When you first set cost parameters, they are read dynamically from SystemFE.Opt_Cost_Table and copied to a memory-resident data structure. The Optimizer then initializes its cost parameters from this memory-resident structure on the fly. The cost parameters do *not* persist across system restarts and reboots.

  The advantage of dynamic target level emulation is that multiple users in multiple sessions can simultaneously emulate environmental cost parameters from multiple target systems on the same test system.

Random AMP sample-based target level emulation permits you to generate random AMP statistical samples on a production system and then export the captured data to a test system for detailed query analysis.

## Related 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 5: "Query Capture Facility."
- Visual Explain Tool.

  See *Teradata Visual Explain User Guide*.
- Teradata System Emulation Tool (TSET).

  See *Teradata System Emulation Tool User Guide*.
- Teradata Index Wizard.

  See *Teradata Index Wizard User Guide* and Chapter 6: "Database Foundations for the Teradata Index Wizard."
- CHECK STATISTICS option of the INSERT EXPLAIN and DUMP EXPLAIN statements.[3]

  See the documentation for these statements in *SQL Reference: Data Definition Statements*.
- Teradata Statistics Wizard.

  See *Teradata Statistics Wizard User Guide*.
- Statistics Collection tool (Teradata Manager).

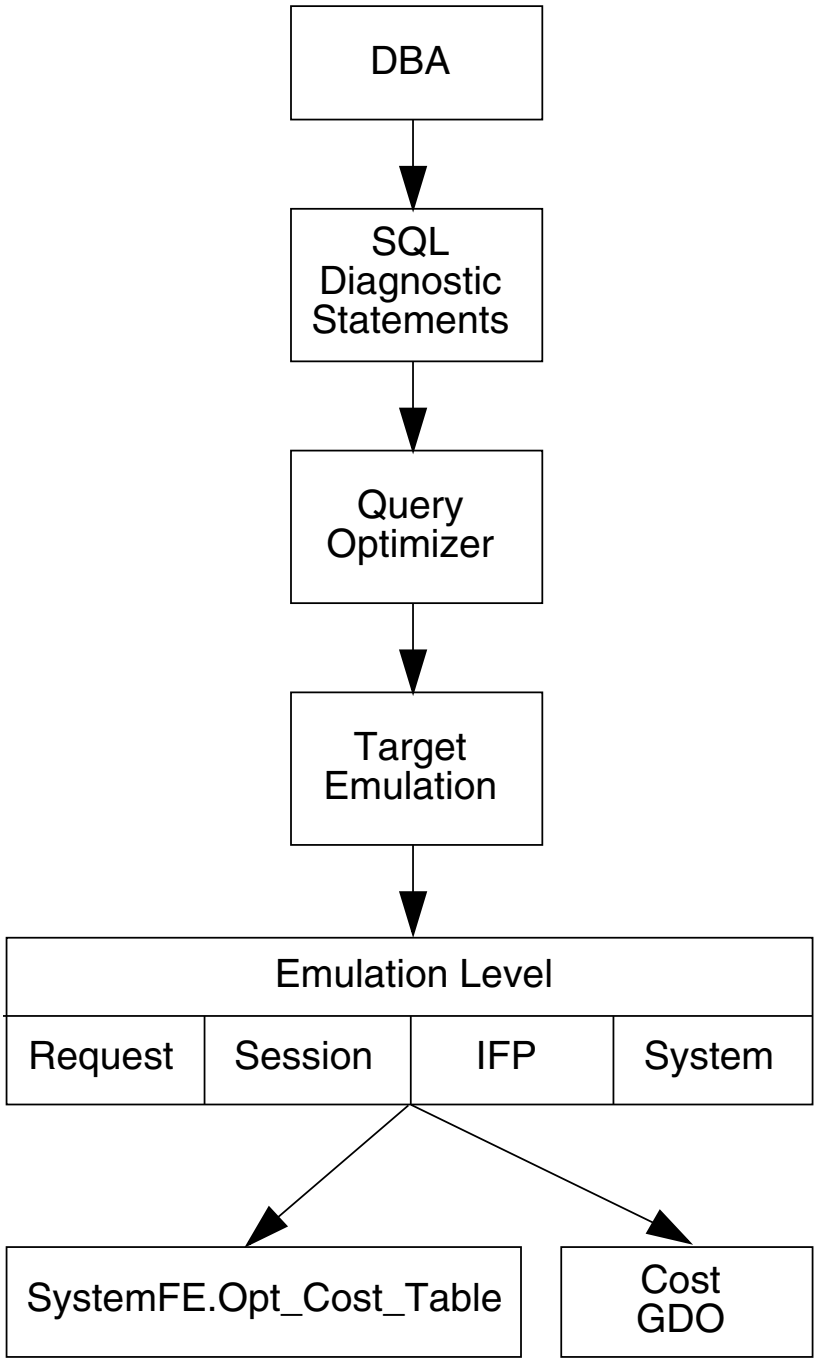
  See *Teradata Manager User Guide*.

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

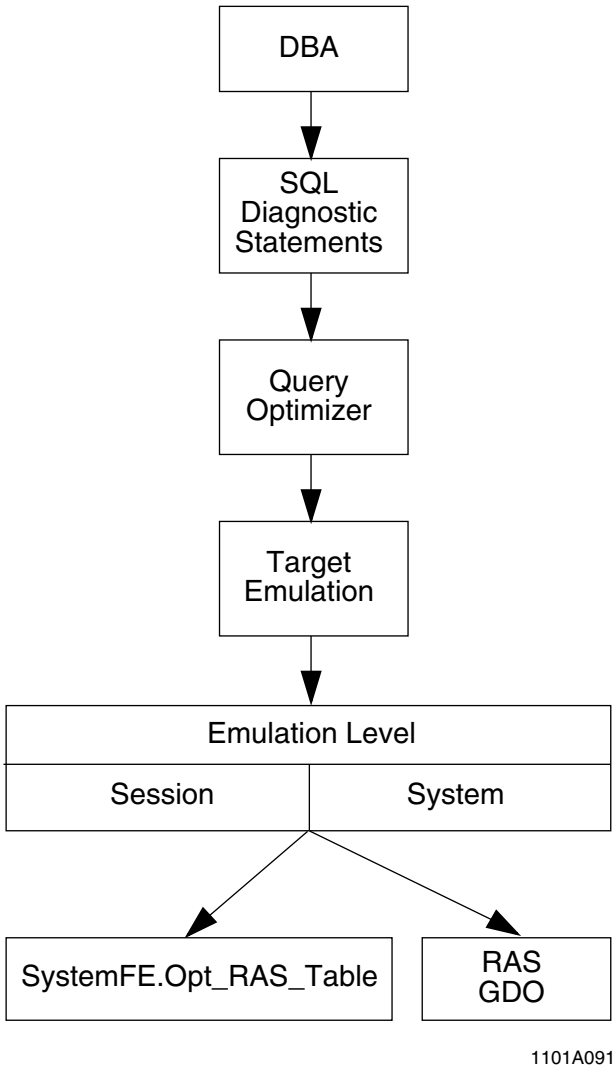
The principal benefit of target level emulation in the end user environment is that it ensures that queries are thoroughly debugged and optimized in a safe, but equivalent, emulated environment prior to introducing them into the production environment.

The following graphic indicates the work flow for environmental cost data capture and test:




FF07D342

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.

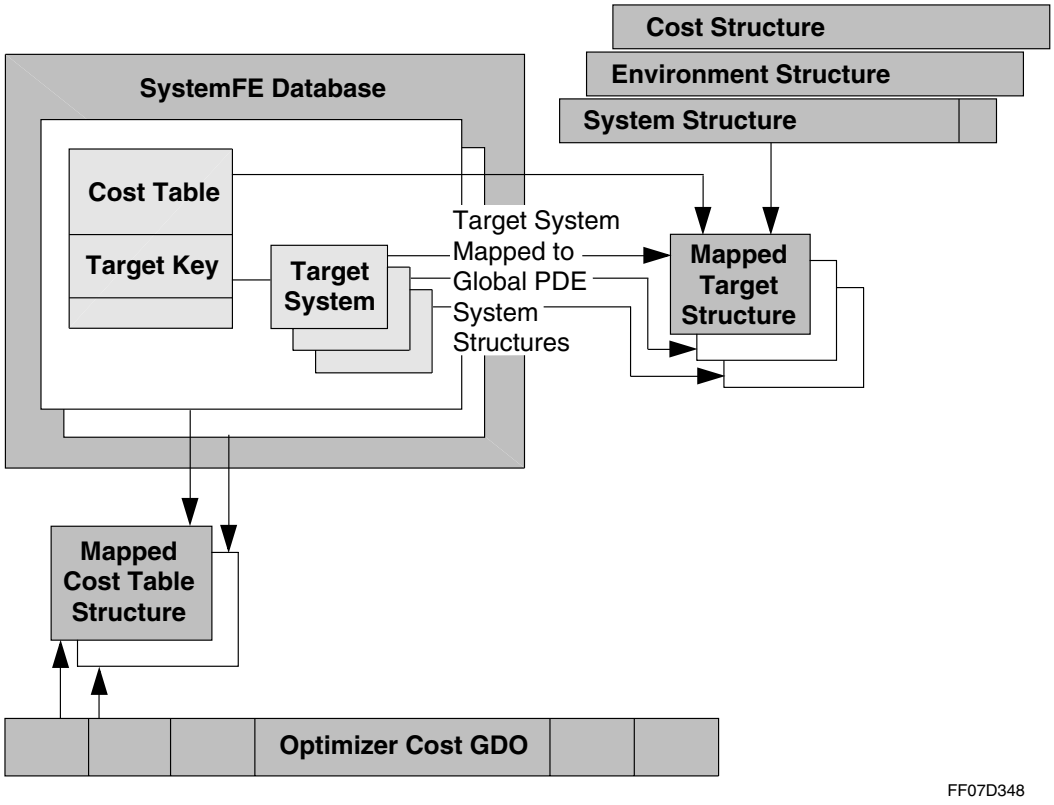
## Procedure To Enable Target Level Emulation

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 DBSControl parameters table SystemFE.Opt_DBSCtl_Table.
2. Use the DBSControl utility to enable Target Level Emulation on your test system by setting DBSControl general field 22, Target Level Emulation, to TRUE.
3. End of procedure.

# Mapping Target System Files to a Test System Optimizer Table and GDO

The following graphic indicates the flow of how target system optimizer environmental cost parameters are mapped to test system structures that, in turn, cause the test system query optimizer to generate query plans that are based on the environmental cost parameters and data demographics of the emulated production system.




The flow for random AMP statistical samples is essentially identical after making the necessary changes to the node labels in the diagram.

# CHAPTER 8 Locking and Transaction Processing

This chapter discusses how the Teradata Database server 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*.

Topics include:

# Database Transactions

## Definition: Transaction

All processing in the 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 the 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 418). 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 409). 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 *un*do 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:

| Response Code Type | Session Mode | Action Taken |
|---|---|---|
| Error | ANSI | Rolls back the error-generating request only.<br>Does *not* release locks placed on behalf of the rolled back request. |
| | Teradata | Not applicable[1] |
| Failure | ANSI | Rolls back the entire transaction that contains the error-generating request. |
| | Teradata | |

1. 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 408) 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:

| Term | Definition |
|---|---|
| Atomicity | A transaction either occurs or it does not. No matter how many component SQL operations are specified within the boundaries of a transaction, they must all complete successfully and commit or they must all fail and rollback. There are no partial transactions. |
| Consistency | A transaction transforms one consistent database state into another. Intermediate inconsistencies in the database are not permitted.<br>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 498 for a more complete statement of this position and its unenforceability. |
| Isolation | The operations of any transaction are concealed from all other transactions until that transaction commits.<br>Isolation is a synonym for serializable (see "Serializability" on page 418). |
| Durability | Once a commit has been made, the new consistent state of the database survives even if the underlying system crashes.<br>Durability is a synonym for persistent. |

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 419 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 406).

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 406) 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[2] and recoverable (see "Serializability" on page 418). 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 418), 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 $Tx_1$ writes before transaction $Tx_2$ reads, then $Tx_1$ must be serialized before $Tx_2$. 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.

## Writing Transaction Histories Symbolically

It is often handy to be able to write a transaction history in shorthand notation. Suppose you have the following sequence of operations occurring with two concurrently running transactions:

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.

Assume the following sequence of actions for the concurrently running transactions $Tx_1$ and $Tx_2$:

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:

| Syntax element … | Specifies … |
|---|---|
| H: | that the following set of symbols represent a transaction history |
| r | a read operation |
| w | a write operation |
| (*x*) | data item *x* |
| (*y*) | data item *y* |
| $c_n$ | commit transaction *n* |
| $r_n$ | roll back transaction *n* |

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.[3]
- A macro.[4]

The restrictions that apply to the items in this list are described when the 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 416).
- 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: "Statement Parsing" for details) are all performed at the level of the request:

- SQL statement syntax checking (see "Syntaxer" on page 7).
- SQL statement syntax annotation (see "Resolver" on page 10).
- Access rights checking (see "Security Checking" on page 12).
- Request optimization (see "Optimizer" on page 13 and Chapter 2: "Query 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 416). Locks are never upgraded or released *within* a request.

For example, consider the following multistatement request:

```
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:

```
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[6]: 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:

| Submission Category | Minimum Number of Requests or SQL Statements Per Transaction | | |
|---|---|---|---|
| | Teradata Mode Implicit Transaction | Teradata Mode Explicit Transaction | ANSI Mode Transaction |
| Requests | 1[1] | 1[2] | 1[4] |
| SQL Statements | 1 | 3[3] | 2[5] |

1. Each individual SQL statement and each multistatement request is treated as a single request. Note that ODBC-compliant applications do not support multistatement requests.
2. 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.
3. The minimum statements are: BEGIN TRANSACTION, an SQL action statement, and END TRANSACTION.
4. 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.
5. 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.

## Two-Phase Locking

The solution to all these 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:

| Phase Number | Action Taken By Transaction Scheduler |
|---|---|
| 1 | Take locks. |
| 2 | Release locks. |

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 Teradata Database 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 the 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 412).
  - The system encounters either of the following SQL statements:[8]
    - 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:[9]
    - ROLLBACK
    - ABORT
  - The system aborts the transaction for reasons external to the transaction itself.

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.

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 423).
- The uncommitted dependency problem (see "The Uncommitted Dependencies (Dirty Read) Phenomenon" on page 424).
- The inconsistent analysis problem (see "The Inconsistent Analysis Phenomenon" on page 426).
- The dirty read problem (see "The Unrepeatable Read Phenomenon" on page 428).

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 413).
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 413).

- The deadlock problem (see "Deadlock" on page 447 and "Pseudo-Table Locks" on page 445).
- Increased system overhead to administer locking.
- Decreased concurrency.

  The impact of the lower concurrency that locks introduce is reduced greatly in a system like the 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). The Teradata Database ensures serializability as long as the current isolation level for the session is SERIALIZABLE (see "The ACID Properties of Transactions" on page 408 and the SET SESSION CHARACTERISTICS statement in *SQL Reference: Data Definition Statements* for details).

For example, suppose table A is a checking accounts table and table B is a savings accounts table. Suppose one transaction, $Tx_1$, 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, $Tx_2$, 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:

| State | Checking Account Amount | Savings Account Amount |
|---|---|---|
| 1 | $900.00 | $100.00 |
| 2 | $500.00 | $100.00 |
| 3 | $500.00 | $500.00 |

- State 1 depicts the initial state of the accounts.
- State 2 depicts an intermediate condition.

  $Tx_1$ 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.

  $Tx_1$ has deposited the $400.00 withdrawn from the checking account into the savings account.

Without two-phase locking, $Tx_2$ 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 424).

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. $Tx_1$ runs.
2. $Tx_2$ runs.

   In this scenario, $Tx_2$ only sees state 3, so the customer passes the credit check.

1. $Tx_2$ runs.
2. $Tx_1$ runs.

   In this scenario, $Tx_2$ 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 416) 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 408, "Levels of Isolation" on page 419, and the SET SESSION CHARACTERISTICS statement in *SQL Reference: Data Definition Statements*).

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."[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.

10. International Standard ISO/IEC 9075-2, Part 2: Foundation (SQL/Foundation), 2003, page 116.

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[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.[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

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:

| Isolation Level | Dirty Read | Nonrepeatable Read | Phantom Read |
|---|---|---|---|
| Read Uncommitted | Possible | Possible | Possible |
| Read Committed | Not possible | Possible | Possible |
| Repeatable Read | Not possible | Not possible | Possible |
| Serializable | Not possible | Not possible | Not possible |

The Teradata Database does not support the isolation levels READ COMMITTED and REPEATABLE READ. READ UNCOMMITTED is implemented using the ACCESS level lock (see "ACCESS" on page 433).

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.

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).

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.[13]

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).

## 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.[14] The mechanism for this is to downgrade the default lock for read-only operations from a severity of READ to a severity of ACCESS.

13. This new form of isolation is commonly referred to as Snapshot Isolation.

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 in accurate result sets. In this case, the results probably would not provide even a reasonable impression of the table data.

The Teradata Database provides methods for allowing the possibility of dirty reads at two different levels: the individual request and the session.

| TO set the default read-only locking severity for this level ... | USE this method ... |
|---|---|
| individual request | LOCKING statement modifier.<br>See *SQL Reference: Data Manipulation Statements* for details of the syntax and usage of this statement modifier. |
| session | SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement.<br>See *SQL Reference: Data Definition Statements* for details of the syntax and usage of this statement. |

# 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 419), 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.

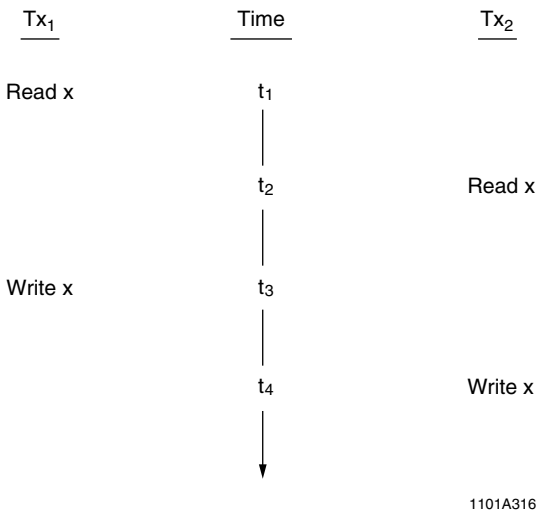
In transaction history notation, this partial[15] history is written as follows:

$$H: r_1(x)\ r_2(x) w_1(x) w_2(x)$$

$Tx_2$ 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.

In the following graphic, the first transaction is called $Tx_1$ and the second is called $Tx_2$.

15. Partial because the history does not include either a commit or rollback operation.




The stages in the transaction history are as follows:

1. At time $t_1$, $Tx_1$ retrieves **row$_1$**.

   The value of the field in question is **value$_0$**.

2. At time $t_2$, $Tx_2$ also retrieves **row$_1$**.

   The value of the field at this time is still **value$_0$**.

3. At time $t_3$, $Tx_1$ updates the field value in **row$_1$**.

   The value of the field is now **value$_1$**.

4. At time $t_4$, $Tx_2$ updates the same field value in **row$_1$**.

   The value of the field is now **value$_2$**.

   The problem with this result is that the update was based on the initial field value of **value$_0$** rather than the correct, updated value, which is **value$_1$**. $Tx_2$ did not see **value$_1$** before writing **value$_2$** 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,[16] and the second transaction then rolls back. As a result, the uncommitted update that occurred at time t1 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

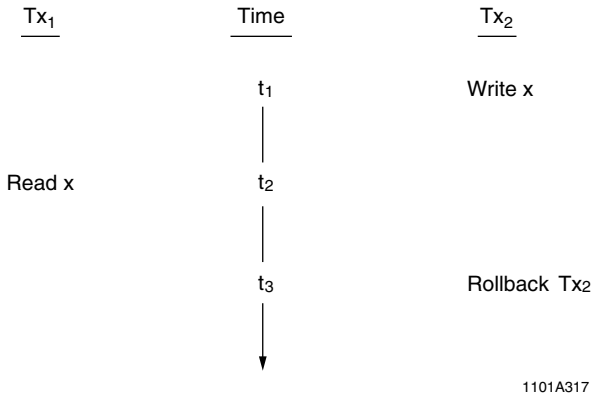
16. This action is referred to as an uncommitted update.

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 $Tx_2$.




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.

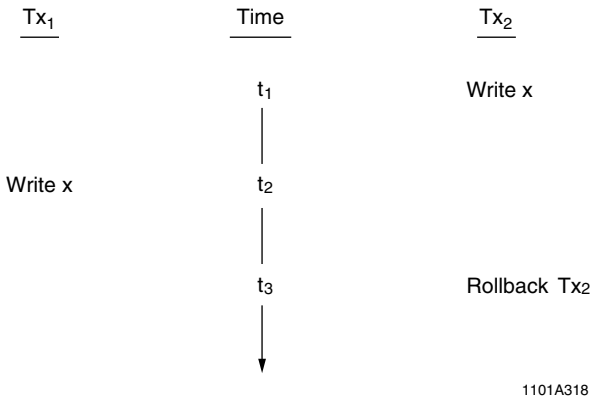
The second example combines an uncommitted dependency with a lost update.

The uncommitted dependency occurs when $Tx_1$ updates **value$_1$** to **value$_2$** after $Tx_2$ had updated **value$_0$** to **value$_1$**, but without committing. $Tx_1$ becomes dependent on $Tx_2$ committing, but because $Tx_2$ instead rolls back at time $t_3$, setting the dependent update value back to $value_0$, the update operation made by $Tx_1$ 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:$w_2(x)w_1(x)r_2$

In the following graphic, the first transaction is named $Tx_1$ and the second is named $Tx_2$.




The stages in the transaction history are as follows:

1. At time $t_1$, $Tx_2$ updates a field in **row$_1$** from **value$_0$** to **value$_1$**.
2. At time $t_2$, $Tx_1$ updates the same field in **row$_1$** from **value$_1$** to **value $_2$**.
3. At time $t_3$, $Tx_2$ rolls back, setting the field value back to **value$_0$**.

## The Inconsistent Analysis Phenomenon

In the following scenario, one transaction is reporting the quantity of a particular part $P_{101}$ 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 $t_0$ are as follows:

| Warehouse 1 | Warehouse 2 | Warehouse 3 |
|---|---|---|
| Qty_$P_{101}$ | Qty_$P_{101}$ | Qty_$P_{101}$ |
| 30 | 750 | 1000 |

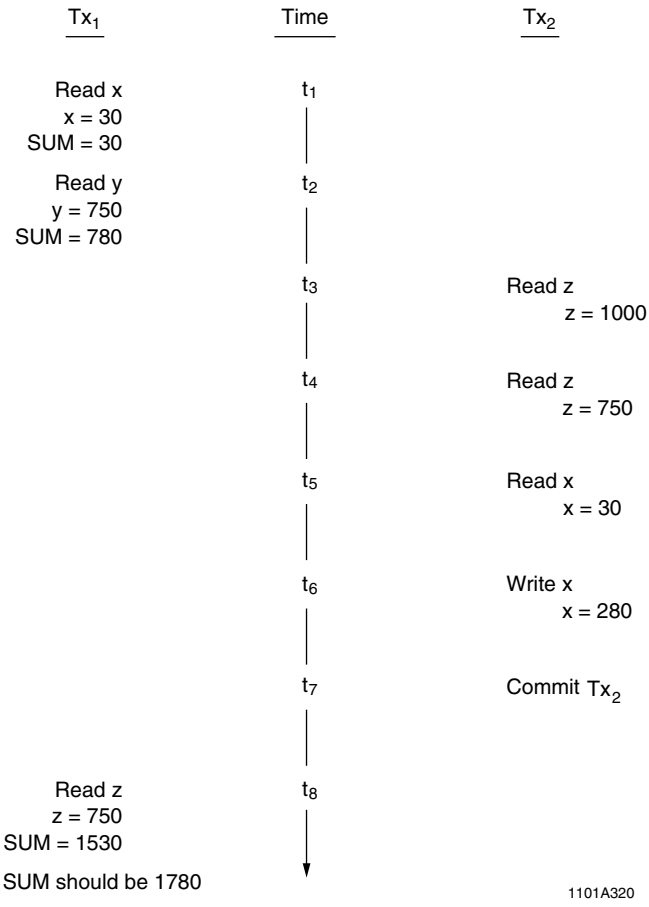
Assume that no parts are shipped or deleted during the course of the example transactions.

In transaction history notation, this partial[17] history is written as follows:

$$H{:}r_1(x - 40)r_1(y - 50)r_2(z - 30)w_2(z - 20)r_2(x - 40)w_2(x - 50)c_2r_1(z - 20)$$

17. Partial because the history does not specify either a commit or a rollback of Transaction 1.

In the following graphic, the first transaction is named $Tx_1$ and the second is named $Tx_2$.




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.
7. At time $t_7$, $Tx_2$ commits.

   According to the database, the quantities of $P_{101}$ in each warehouse are 280, 750, and 750 for warehouse 1, warehouse 2, and warehouse 3, respectively.
8. At time $t_8$, $Tx_1$ retrieves **row$_{WH3}$**, which now has the $P_{101}$ 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

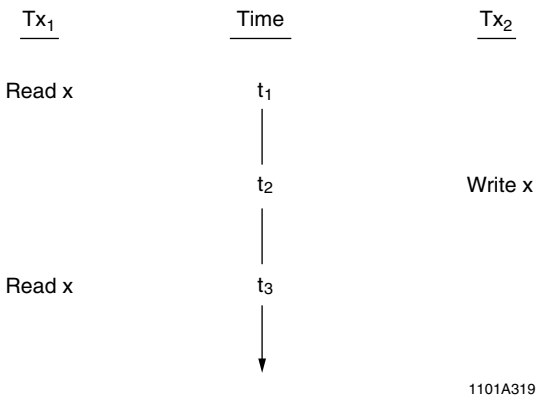
Depending on the application, the unrepeatable read phenomenon might not be a concern. In fact, the Teradata ACCESS-level locking severity is specifically designed to permit unrepeatable reads.

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 Teradata Database 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 the 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 modifier (see *SQL Reference: Data Manipulation Statements*).

## Locking Levels

The Lock Manager implicitly locks an object at the following levels:

| This lock level ... | Locks ... |
|---|---|
| **Most Restrictive** | |
| Database | all rows of all tables in the database and their associated secondary index subtables. |
| Table | all rows in the base table and in any secondary index and fallback subtables associated with it. |
| View | all underlying tables accessed by the view. |
| Row Hash | the primary copy of rows sharing the same row hash value.<br>A row hash lock permits other users to access other data in the table and is the least restrictive type of lock.<br>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.<br>• A row hash lock is applied whenever a table is accessed using a *unique primary index (UPI)*.<br>• For an update that uses a *unique secondary index (USI)*, the appropriate row of the secondary index subtable is also locked.<br>• It is not necessary to lock the fallback copy of the row, nor any associated row, of a *nonunique secondary index* (NUSI) subtable. |
| **Least Restrictive** | |

## Locking Considerations

The 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. When determining whether to grant a lock, the Lock Manager takes into consideration both the requested locking severity and the object to be locked.

For example, a READ lock requested at the table level cannot be granted if a WRITE or EXCLUSIVE lock already exists on any of the following database objects:

- The database that owns the table
- The table itself
- Any rows in the table

A WRITE lock requested at the row hash level cannot be granted if a READ, WRITE, or EXCLUSIVE lock already exists on any of the following database objects:

- The owner database for the table
- The parent table for the row
- The row hash itself

In each case, the request is queued until the conflicting lock is released.

It is possible to exhaust Lock Manager resources. Any transaction that requests a lock when Lock Manager resources are exhausted aborts. In such cases, row hash locking for DDL statements can be disabled.

You can review all the active locks and determine which other user locks are blocking your transactions if you have the performance monitor MONITOR SESSION privilege and an application that uses the Performance Monitor/Application Programming Interface.

## 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)

# 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[19], 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.

19. 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 Lock Levels and Severities

## Introduction

Teradata Database locks have two orthogonal dimensions: level and severity.[20]

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,[21] 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 432 and "Lock Manager" on page 429 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 433 and "Default Lock Assignments and Lock Upgradeability" on page 437 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[22]
- 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 429) 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.

20. 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.

21. Note that the 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.

22. See "Pseudo-Table Locks" on page 445 for a description of a special category of table-level locking.

## Locking Severity and Relative Restrictiveness

The available lock severities, from most restrictive to least restrictive, are as follows:

| Lock Severity | Description |
|---|---|
| **Most Restrictive** | |
| EXCLUSIVE | Placed only on a database or table when the object is undergoing structural changes (for example, a column is being created or dropped) or when a database is being restored by a host utility.<br>An EXCLUSIVE lock restricts access to the object by any other user.<br>You can also place this lock explicitly using the LOCKING modifier (see *SQL Reference: Data Manipulation Statements*). |
| WRITE | Placed in response to an INSERT, UPDATE, or DELETE request.<br>A WRITE lock restricts access by other users (except for readers who are not concerned with data consistency and choose to override the automatically applied lock by specifying a less restrictive ACCESS lock).<br>You can also place this lock explicitly using the LOCKING modifier (see *SQL Reference: Data Manipulation Statements*). |
| READ | Placed in response to a SELECT request.<br>A READ lock restricts access by users who require EXCLUSIVE or WRITE locks.<br>You can also place this lock explicitly using the LOCKING modifier (see *SQL Reference: Data Manipulation Statements*). |
| CHECKSUM | Placed in response to a user-defined LOCKING FOR CHECKSUM modifier (see *SQL Reference: Data Manipulation Statements*) when using cursors in embedded SQL.<br>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.<br>See also "Cursor Locking Modes" on page 465, *SQL Reference: Data Manipulation Statements*, and *SQL Reference: Stored Procedures and Embedded SQL*. |
| ACCESS | Placed in response to a user-defined LOCKING FOR ACCESS modifier (see *SQL Reference: Data Manipulation Statements*) or by setting the session default isolation level to READ UNCOMMITTED using the SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement (see *SQL Reference: Data Definition Statements*).<br>Permits the user to access for READ an object that may be already locked for READ or WRITE. An ACCESS lock does not restrict access by another user except when an EXCLUSIVE lock is required.<br>A user requesting access disregards all data consistency issues. Because ACCESS and WRITE locks are compatible, the data might be undergoing updates while the user who requested the access is reading it. Therefore, any query that requests an ACCESS lock might return incorrect or inconsistent results |
| **Least Restrictive** | |

A locking scheme that provides multiple levels of locking severity, such as the scheme implemented by the 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.

## 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 the Teradata Database.

Read locks always conflict with Write locks, while Write locks always conflict with other Write locks.

The Teradata Lock Manager uses three types of Read lock:

- ACCESS (see "ACCESS" on page 433).
- CHECKSUM (see "CHECKSUM" on page 433).
- READ (see "READ" on page 433).

The Teradata Lock Manager also uses two types of Write lock:

- WRITE (see "WRITE" on page 433).
- EXCLUSIVE (see "EXCLUSIVE" on page 433).

The following table summarizes the compatibilities and incompatibilities among the various locking severities used by the Teradata Lock Manager:

| This locking severity ... | Is compatible with this locking severity ... | But is not compatible with this locking severity ... |
|---|---|---|
| • ACCESS<br>• CHECKSUM | • ACCESS<br>• CHECKSUM<br>• READ<br>• WRITE | EXCLUSIVE |
| READ | • ACCESS<br>• CHECKSUM<br>• READ | • WRITE<br>• EXCLUSIVE |
| WRITE | • ACCESS<br>• CHECKSUM | • READ<br>• WRITE<br>• EXCLUSIVE |
| EXCLUSIVE | None. | • ACCESS<br>• CHECKSUM<br>• READ<br>• WRITE<br>• EXCLUSIVE |

The following notation is used to describe the locking severity compatibilities:

| Notation | Definition |
|---|---|
| *Q* | A locking queue associated with a database object. |
| *L* | A list of locks currently held. |
| `lock(<object>,<lock requested>)` | A database object-locking severity requested pair. |

Each database object has an associated locking queue *Q* and list of currently held locks *L*. All transactions perform a locking operation before they access any database objects.

| IF lock(<object>,<lock requested>) is ... | THEN ... |
|---|---|
| queued for any lock in *L* | the transaction is placed in *Q* and waits there as long as `lock(<object>,<lock requested>)` is queued.<br>A request in this state is said to be blocked (see "Blocked Requests" on page 442). |
| granted | `lock(<object>,<lock requested>)` is added to *L* with `<lock requested>` and the transaction resumes processing. |

After the transaction finishes with an object by either committing or rolling back, its lock is removed from *L*.

The following table summarizes the action taken when a requested locking severity competes with an existing locking severity:

| Severity of Requested Lock | Severity of Held Lock | | | | |
|---|---|---|---|---|---|
| | None | ACCESS | READ | WRITE | EXCLUSIVE |
| ACCESS or CHECKSUM | Lock Granted | Lock Granted | Lock Granted | Lock Granted | Request Queued* |
| READ | Lock Granted | Lock Granted | Lock Granted | Request Queued* | Request Queued* |
| WRITE | Lock Granted | Lock Granted | Request Queued* | Request Queued* | Request Queued* |
| EXCLUSIVE | Lock Granted | Request Queued* | Request Queued* | Request Queued* | Request Queued* |

* If you specify NOWAIT, the transaction aborts instead of queueing.

A queued request is in an I/O wait state and is said to be blocked (see “Blocked Requests” on page 442).

A WRITE or EXCLUSIVE lock on a database, table, or view restricts all requests from accessing data within the domain of that object except the one holding the lock.

Because a lock on an entire database can restrict access to a large quantity of data, the Lock Manager ensures that default database locks are applied at the lowest possible level and severity required to secure the integrity of the database while simultaneously maximizing concurrency.

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.

## NOWAIT

When you specify the NOWAIT option for the SQL LOCKING phrase, 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 phrase, see *SQL Reference: Data Manipulation Statements*.

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 459.

# Default Lock Assignments and Lock Upgradeability

## Introduction

The Lock Manager assigns locking levels and severities to SQL requests by default.[23] 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:

| SQL Statement | Access Type | | Default Lock Type Assigned |
|---|---|---|---|
| | UPI or USI | NUSI or FTS | |
| SELECT | Row hash | Table | READ[1]<br>SHARE |
| SELECT AND CONSUME | Row hash | Row hash | WRITE[2] |
| INSERT | Row hash | Not applicable | WRITE |
| UPDATE | Row hash | Table | WRITE |
| MERGE | Row hash | • Not applicable for INSERT<br>• Table for UPDATE | WRITE |
| DELETE | Row hash | Table | WRITE |
| CREATE TABLE<br>DROP TABLE<br>ALTER TABLE | Not applicable | Table | EXCLUSIVE |
| CREATE DATABASE<br>DROP DATABASE<br>MODIFY DATABASE | Not applicable | Database | EXCLUSIVE |

1. 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 459 for further information.
2. 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.

23. The Optimizer determines what locking levels and severities are necessary to service a request, but it does not actually assign locks.

## Changing Lock Assignments

Depending on the assigned lock and the individual SQL request, you can change some default lock assignments using the LOCKING modifier (see *SQL Reference: Data Manipulation Statements* 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:

| A change from this default assignment lock ... | TO this user-specified lock ... | IS ... |
|---|---|---|
| ACCESS[1] | ACCESS | redundant, but valid. |
| READ[2]<br>SHARE | READ<br>SHARE | |
| WRITE | WRITE | |
| EXCLUSIVE | EXCLUSIVE | |
| ACCESS | READ<br>SHARE | a valid upgrade. |
| | WRITE | |
| | EXCLUSIVE | |
| READ<br>SHARE | WRITE | |
| | EXCLUSIVE | |
| WRITE | EXCLUSIVE | |
| READ | ACCESS | a valid downgrade. |

1. Even though a CHECKSUM lock is essentially identical to an ACCESS lock, you can only specify CHECKSUM locks explicitly using the LOCKING modifier; you cannot upgrade them to a higher severity lock. This is because the Teradata Database never specifies CHECKSUM as a default lock severity.
2. READ and SHARE are synonyms.

The following table combines the information from the two previous tables to indicate associations between individual SQL DML statements and lock upgrades and downgrades at the row hash, view, and table database object levels:

| This LOCKING modifier severity specification … | Is available for this SQL DML statement … |
|---|---|
| EXCLUSIVE | DELETE<br>INSERT<br>MERGE<br>SELECT<br>UPDATE |
| WRITE | SELECT<br>SELECT AND CONSUME |
| READ | SELECT |
| ACCESS | SELECT |

The reason you can only specify the LOCKING FOR EXCLUSIVE modifier for the DELETE, INSERT, MERGE, and UPDATE statements is that the default lock severity for these statements is WRITE. You cannot downgrade a WRITE lock 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.

## 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 QuerySession).

## Releasing Locks

Except for locks placed during Host Utility (HUT) processing, users have no control over when locks are released. You must explicitly release any locks set by HUT processing (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 424).
- 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 Reference: Data Manipulation Statements* for information about how to do this).
- If the CREATE INDEX or ALTER TABLE … FALLBACK LOCKING modifier specifies WRITE (or if there is no LOCKING modifier specified), then specify an ACCESS lock in your SELECT statement to permit concurrency (see "LOCKING" in *SQL Reference: Data Manipulation Statements* 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 Reference: Data Manipulation Statements* 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 434) is considered to be blocked (see "Compatibility Among Locking Severities" on page 434).[24] 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.

## 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.[25]

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 (Franaszek and Robinson, 1985; Haerder, 1984; Mohan, 1992 - see Thomasian (1998) for a review).

## 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 the head, at which time they are granted the locks they are waiting for and execute.

24. 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 431.

25. 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.

The only requests the Lock Manager never enqueues are those explicitly specified with a LOCKING modifier and the NOWAIT option (see *SQL Reference: Data Manipulation Statements* for information about the LOCKING 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.[26] A deadlock is a lock contention situation that cannot be solved without external intervention of some kind (see "Deadlock" on page 447). For the 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 Statement Transactions

When several requests that compete for the same table are submitted as separate, single statement transactions, the Lock Manager resolves their locking requirements as illustrated by the following process:

1. Job_1 requires a READ lock on table_a.

   table_a is free, so the lock is granted and job_1 begins.
2. While job_1 is still running, job_2 requires a WRITE lock.

   This conflicts with the active READ lock, so the WRITE lock is denied and job_2 is queued.
3. Job_3 requires an ACCESS lock.

   ACCESS locks are compatible with both READ and WRITE locks (if job_1 completes, releasing the READ lock, then job_2 can begin whether or not job_3 still holds the ACCESS lock), so the ACCESS lock is granted and job_3 is allowed to run concurrently with job_1.
4. Job_4 requires a READ lock.

   This conflicts with the queued WRITE lock, so job_4 is queued behind job_2.
5. Job_5 requires an EXCLUSIVE lock.

   An EXCLUSIVE lock conflicts with all other locks, so job_5 is queued behind job_4.
6. Job_6 requires an ACCESS lock.

   This conflicts with the queued EXCLUSIVE lock, so job_6 is queued behind job_5.
7. End of process.

26. 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.

## 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:

| | Process | | | |
|---|---|---|---|---|
| Stage | This transaction number … | Has this request … | That requires this lock … | With this result … |
| 1 | 1 | `SELECT … FROM table_a` | READ | the READ lock on table_a is granted and the SELECT request begins processing. |
| | | `INSERT INTO table_a …` | WRITE | the WRITE lock on table_a is not granted and the INSERT into table_a request is blocked pending completion of the SELECT request. |
| 2 | 2 | `INSERT INTO table_a …` | WRITE | transaction 2 is queued because its request for a WRITE lock on table_a is not compatible with the READ lock already in place on table_a. |
| 3 | 1 | `SELECT … FROM table_a` | None.<br>The READ lock is released. | processing of the SELECT request from Transaction 1 completes. |
| 4 | | `INSERT INTO table_a …` | WRITE | the Lock Manager upgrades the lock on table_a for the Transaction 1 INSERT request. |
| 5 | 2 | `INSERT INTO table_a …` | WRITE | transaction 2 remains queued and inactive, waiting for its WRITE lock request on table_a to be granted. |

# Pseudo-Table Locks

A pseudo-table can be thought of 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 447).

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.[27]

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 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.

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.

27. 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, a 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. See any of the texts listed in "References" on page 494 for a detailed explanation of intention locks.

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.

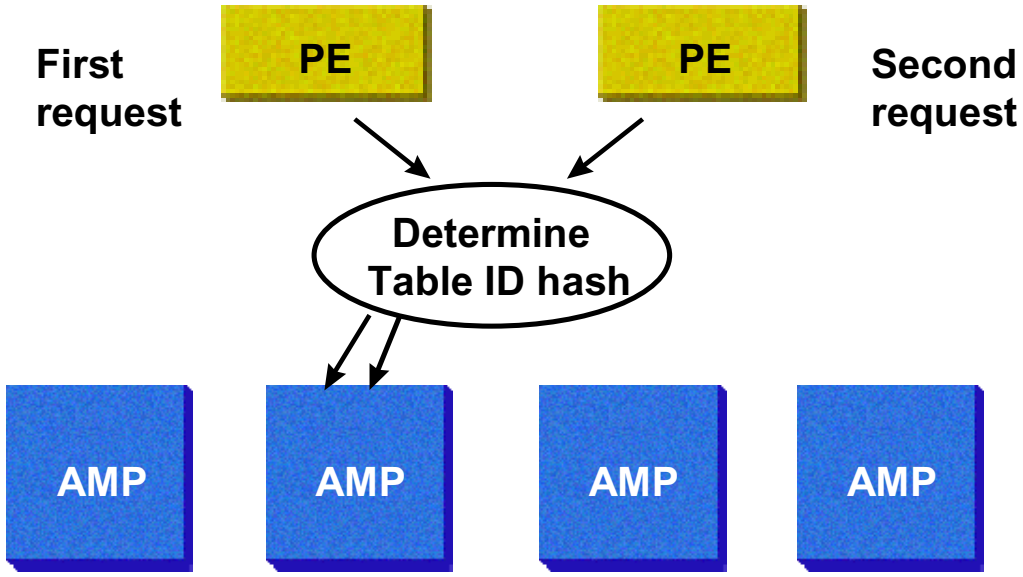
The Teradata Database handles all-AMP lock requests as follows:

1 The PE that processes an all-AMPs request determines the table ID hash for an AMP to manage the all-AMP lock request.

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:



# Deadlock

A problem that can occur with two-phase locking (see "Two-Phase Locking" on page 416) is the situation in which two requests lock database objects 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 object 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[28]

The 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 DBSControl 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) transaction of the two in the deadlock.

Be aware that client software handles deadlocks differently than the Teradata Database. For example, the BTEQ command .RETRY retries a *request*, not a transaction (see *Basic Teradata Query Reference* for details). If this occurs, the outcome of any updates made by statements in the transaction prior to the failed request are lost, as is the information that a transaction is in progress. The system does not retry any SQL statements in the transaction prior to the failed statement, but it *does* retry the failed statement and all subsequent statements.

## 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.

28. 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.

# 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 modifier in your SQL statements.

## How You Can Use The LOCKING Modifier To Help Prevent Deadlocks

The LOCKING 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 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 modifier:

- The LOCKING modifier is not ANSI SQL-2003-compliant.
- The LOCKING 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 modifier can be specified in the same request.
- The CREATE VIEW and REPLACE VIEW statements allow the LOCKING modifier to be specified as part of the view definition.
- You can use the LOCKING 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[29].

29. 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):

| User | SQL Text |
|---|---|
| A | BEGIN TRANSACTION;<br>SELECT y<br>FROM table_a<br>WHERE pi = 1;<br>UPDATE table_a<br>SET y = 0<br>WHERE pi = 1;<br>END TRANSACTION; |
| B | BEGIN TRANSACTION;<br>SELECT z<br>FROM table_a<br>WHERE pi = 1;<br>UPDATE table_a<br>SET z = 0<br>WHERE pi = 1;<br>END TRANSACTION; |

In this case, UserA and UserB are allowed to access the rows sharing the same row hash value simultaneously for READ during SELECT processing. When the UserA UPDATE statement requires a WRITE lock on the row hash value, the upgrade request is queued, waiting for the UserB READ lock to be released.

However, the UserB READ lock cannot be released because the UserB UPDATE statement also requires a WRITE lock on the row hash value, and that request is queued waiting for the UserA Read lock to be released.

You can avoid such deadlocks by preceding the transaction with a 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:

| User | SQL Text |
|---|---|
| A | BEGIN TRANSACTION;<br>  LOCKING ROW FOR WRITE<br>  SELECT y<br>  FROM table_a<br>  WHERE USI = 1;<br>  UPDATE table_a<br>  SET y = 0<br>  WHERE USI = 1;<br>END TRANSACTION; |
| B | BEGIN TRANSACTION;<br>  LOCKING ROW WRITE<br>  SELECT z<br>  FROM table_a<br>  WHERE USI = 1;<br>  UPDATE tableA<br>  SET z = 0<br>  WHERE USI = 1;<br>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 modifier with the NOWAIT option.

NOWAIT specifies that the entire transaction, (including one in ANSI mode), is to be aborted if the lock manager cannot place the necessary lock on the target object immediately upon receipt of a statement.

This situation is treated as a fatal error. The user is informed that the transaction was aborted, and any processing performed up to the point at which NOWAIT took effect is rolled back.

## The LOCKING Modifier in View Definitions

The LOCKING 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 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 QualifyAccnt is defined as a secondary index, the following request:

```
LOCKING TABLE AccntRec FOR ACCESS
SELECT AccntNo, QualifyAccnt
FROM AccntRec
WHERE QualifyAccnt = 1587;
```

could return the following result:

```
AccntNo  QualifyAccnt
-------  ------------
   1761          4214
```

In this case, the value 1587 was found in the secondary index subtable, and the corresponding data row was selected and returned. However, the data for account 1761 had been changed by the other user while this selection was in progress.

Returns such as this are possible even if the data is changed or deleted only momentarily by a transaction that is subsequently aborted.

This type of inconsistency can occur even if the data is changed only momentarily by a transaction that is later backed out.

# Example: Transaction Without Deadlock

## Introduction

The following example demonstrates how to optimize concurrency using the guidelines from "Releasing Locks" on page 440.

## 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:

| Transaction Number | SQL Text |
|---|---|
| 1 | `LOCKING table_1 FOR READ`<br>`ALTER TABLE table_1, FALLBACK;` |
| 2 | `SELECT *`<br>`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:

| Transaction Number | Request Number | SQL Text |
|---|---|---|
| 1 | 1 | `LOCKING table_1 FOR READ`<br>`CREATE INDEX (column_3, column_4) ON table_1;` |
| 2 | 2 | `BEGIN TRANSACTION;` |
| | 3 | `SELECT *`<br>`FROM table_1;` |
| | 4 | `UPDATE table_1`<br>`SET column_1 = <value-1>`<br>`WHERE column_1 = <value-2>;` |
| | 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.

```
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 modifier from the first transaction.

```
CREATE INDEX (column_3, column_4) ON table_1;
```

# Example: Two Serial Transactions

## Introduction

The following example shows two transactions, with the first transaction starting before the second transaction.

## Table Definition

Consider the following table definition:

```
CREATE TABLE table_1
 (column_1 INTEGER,
  column_2 INTEGER,
  column_3 INTEGER,
  column_4 INTEGER)
PRIMARY INDEX (column_1);
```

## Problem Transactions

The following two transactions are running concurrently, with the first having started prior to the second:

| Transaction Number | SQL Text |
|---|---|
| 1 | `LOCKING table_1 FOR READ`<br>`CREATE INDEX (column_3, column_4) ON table_1;` |
| 2 | `SELECT *`<br>`FROM table_1`<br>`WHERE column_3 = 124`<br>`AND column_4 = 93;` |

Each transaction places a table-level READ lock on table_1. The transactions obtain access to table_1 and run concurrently.

Note that the SELECT statement does not recognize the index being created by the CREATE INDEX statement.

## Problem Resolution

To eliminate concurrency, change the coding of the first transaction to make its explicit lock EXCLUSIVE.

| Transaction Number | SQL Text |
|---|---|
| 1 | `LOCKING table_1 FOR EXCLUSIVE`<br>`CREATE INDEX (column_3, column_4) ON table_1;` |
| 2 | `SELECT *`<br>`FROM table_1`<br>`WHERE column_3 = 124`<br>`AND column_4 = 93;` |

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 tables are affected by this locking downgrade on a blocked READ lock request:

- DBC.AccLogRuleTbl
- DBC.ConstraintNames
- DBC.Indexes
- DBC.TableConstraints
- DBC.TextTbl
- DBC.Triggers
- 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 Statements and Locks

When it processes DML statements, 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 a result of DML statement request are listed in the following table:

| DML Request | Updated Fields | Selection Criteria | Object Locked | Locking Severity |
|---|---|---|---|---|
| SELECT | Not applicable | UPI or USI[3] | Row hash | READ |
| | | NUPI[4] | Set of rows | READ |
| | | Any other[3] | Table | READ[5] |
| SELECT AND CONSUME | Not applicable | Any | Row hash | WRITE |
| DELETE[1] | Not applicable | UPI or USI | Row hash | WRITE |
| | | NUPI | Set of rows | WRITE |
| | | Any other | Table | WRITE |
| INSERT … SELECT | Not applicable | Select table | | |
| | | UPI or USI | Row hash | READ |
| | | NUPI | Set of rows | READ |
| | | Any other | Table | READ |
| | | | Insert table | WRITE |
| INSERT … [VALUES][1] | Not applicable | Not applicable | Primary row | WRITE |
| UPDATE[1] | Neither UPI nor USI | UPI or USI | Row hash | WRITE |
| | Neither NUPI nor USI | NUPI | Set of rows | WRITE |
| | | Any other | Table | WRITE |
| | USI | USI | Table | WRITE |
| MERGE (Update)[2] | Neither UPI nor USI | UPI or USI | Row hash | WRITE |
| | Neither NUPI nor USI | NUPI | Set of rows | WRITE |
| | | Any other | Table | WRITE |
| | USI | USI | Table | WRITE |
| MERGE (Insert) [1] | Not applicable | Not applicable | Primary row | WRITE |

1. 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.
2. 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.
3. If a join or hash index includes the specified columns, then the entire index is READ-locked and the base table is not accessed.
4. 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.
5. When the SELECT is a tactical query, the system downgrades the lock request from READ to ACCESS if the operation would otherwise be blocked.

# Locking Issues With Consume Mode SELECT Queries on a Queue Table

Several locking issues apply to consume mode SELECT[30] 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 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:

  **a** The system grants the WRITE lock on the first row hash in the queue.

  **b** Transaction processing resumes.

  **c** Rows are consumed until the queue is empty.

  **d** 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;
```

30. 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 412):

```
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.
- The Teradata Database uses the Teradata Workload Manager software to manage all deferred requests (transactions in a delayed state) against a queue table.[31] 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.

31. 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 the 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 modifier. The LOCKING modifier is supported for updatable cursors, and the CHECKSUM locking severity is designed especially for use with LOCKING.

## Example

This example uses 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 the Teradata Database to UPDATE or DELETE the data row when the application specifies such an action against WHERE CURRENT OF *cursor_name*.

| IF the locking modifier is … | THEN the system … |
|---|---|
| ACCESS | does *not* check to ensure that the data row to be updated or deleted has not been modified since the response data was generated for the spool table.<br>The target data row could be a completely new row if some other application has deleted the original source row and inserted a new row in its place. |
| CHECKSUM | inserts a checksum into each row of the spool file.<br>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.<br>Note that the CHECKSUM option does *not* guarantee that all conflicts will be detected. There is a small, but finite, possibility that a row created by update might have an identical checksum to the original, unmodified row.<br>The CHECKSUM option uses ACCESS severity locks. CHECKSUM locking differs in that it also provides the checksums of the spool file rows. |

# 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:

| FOR this client software … | USE these commands or options … | TO switch to this session mode … |
|---|---|---|
| BTEQ | `.[SET] SESSION TRANSACTION ANSI` | ANSI |
| | `.[SET] SESSION TRANSACTION BTET` | Teradata |
| | Log off prior to entering this command.<br>The command does not take effect until the next logon.<br>See *Basic Teradata Query Reference* for more detail on using BTEQ commands. | |
| Preprocessor2 | `TRANSACT(ANSI)` | ANSI |
| | `TRANSACT(BTET)`<br>`TRANSACT(2PC)`<br>`TRANSACT(COMMIT)` | Teradata |
| | See *Teradata Preprocessor2 for Embedded SQL Programmer Guide* for more detail on setting Preprocessor options. | |

| FOR this client software ... | USE these commands or options ... | TO switch to this session mode ... |
|---|---|---|
| CLIv2 | `set tx_semantics = 'A'` | ANSI |
| | `set tx_semantics = 'T'` | Teradata |
| | `set tx_semantics = 'D'` | Server default |
| | See the following manuals for more detail on setting the tx_semantics field:<br>• *Teradata Call-Level Interface Version 2 Reference for Channel-Attached Systems*<br>• *Teradata Call-Level Interface Version 2 Reference for Network-Attached Systems* | |

See the relevant product documentation for client software based on the ODBC API to determine how to switch session modes using that software. Also see *SQL Reference: Fundamentals* for information about changing 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:

| These SQL statements ... | Are not valid in this session mode ... |
|---|---|
| `BEGIN TRANSACTION`<br>`BT` | ANSI |
| `END TRANSACTION`<br>`ET` | |
| `COMMIT [WORK]` | Teradata |

## 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 Warehouse* for a brief description of 2PC) is supported for Teradata session mode, but not for ANSI session mode sessions.

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:
  a. No transaction is currently active.
  b. 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 409), 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 (Row hash and Table) and are not released until a transaction is committed.
- 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 412).
- 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 412).

- Error responses (see *SQL Reference: 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 408) 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, the 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 Reference: Functions and Operators*).
- 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 Warehouse* and "2PC" on page 521) 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 478).

The set contains the following case studies:



## ANSI Mode Successful Transaction

The following transaction is an example of a successful ANSI mode transaction:

| | |
|---|---|
| `INSERT INTO employee`<br>`SELECT *`<br>`FROM employee;`<br><br>`*** Insert completed. 26 rows added.` | This single statement/single request begins an ANSI mode explicit transaction.<br>WRITE locks are held on employee. |
| `UPDATE employee`<br>`SET department_number = 400`<br>`WHERE department_number = 401`<br>`;DELETE FROM employee`<br>`WHERE department_number = 401;`<br><br>`*** Update completed. 7 rows changes.`<br><br>`*** Delete completed. No rows removed.` | This multiresponse request is composed of two statements, but is a single request.<br>WRITE locks are still held on employee. |
| `SELECT *`<br>`FROM employee`<br>`WHERE department_number = 401;`<br><br>`*** Query completed. No rows found.` | This single statement/single request is a check to ensure that all employees in department_number 401 were deleted in the previous request.<br>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. |
| `COMMIT;`<br><br>`*** COMMIT done.`<br>`*** Total elapsed time was 1 second.` | This statement terminates the explicit transaction by committing all changes.<br>All locks are released and the Transient Journal is dropped from the dictionary.<br>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`<br>`SELECT *`<br>`FROM customer_service.employee;`<br><br>`*** Insert completed. 26 rows added.` | This INSERT … SELECT statement begins an ANSI mode explicit transaction. |
| `SELECT *`<br>`FRM employee;`<br><br>`*** Error 3706 Syntax error; SELECT * must have a FROM clause.` | Syntaxer problems are errors, not failures.<br>The inserts from the previous statement remain, and all locks continue to be in force. |
| `SELECT *`<br>`FROM employee;`<br><br>`*** Query completed. 26 rows found. 9 columns returned.` | Correcting the SELECT syntax produces a successful statement that indicates the inserted rows from the first statement remain in place.<br>The transaction is still not committed. |
| `SELECT *`<br>`FROM employee`<br>`WHERE emp_num = 1010;`<br><br>`*** Error 5628 Column emp_num not found in Employee.` | Resolver problems are neither always errors, nor always failures. In this case, the problem evokes an Error response, not a Failure response.<br>The inserts from the first statement remain, and all locks continue to be in force. |
| `SELECT *`<br>`FROM employee;`<br><br>`*** Query completed. 26 rows found. 9 columns returned.` | Repeating the selection of all columns from the table proves that the inserted rows from the first statement remain.<br>The transaction is still not committed. |

| | |
|---|---|
| `INSERT INTO employee`<br>`SELECT *`<br>`FROM customer_service.employee;`<br><br>`*** Insert completed. 26 rows added.` | This INSERT … SELECT statement begins an ANSI mode explicit transaction. |
| `CREATE TABLE tbl_1 (`<br>`  col_1,`<br>`  col_2 INTEGER);`<br><br>`*** Error 3739 The user must give a data type for .` | Syntaxer problems are errors, not failures.<br>The inserts from the previous statement remain, and all locks continue to be in force. |
| `SELECT *`<br>`FROM employee;`<br><br>`*** Query completed. 26 rows found. 9 columns returned.` | Selecting all columns from the table proves that the inserted rows from the first statement remain.<br>The transaction is still not committed. |

| | |
|---|---|
| CREATE TABLE tbl_1 (<br>  col_1 INTEGER,<br>  col_2 INTEGER);<br><br>*** Failure 3802 Table 'tbl_1' already exists. | This resolver problem evokes a Failure response, which terminates the transaction.<br>The transaction rolls back and the rows inserted into employee are deleted. |
| SELECT *<br>FROM employee;<br><br>*** Query completed. No rows found. | Selecting all columns from the table proves that the previously inserted rows have all been deleted. |

## ANSI Mode Transactions and DDL: DDL Request Placement Within the Transaction

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.

| | |
|---|---|
| CREATE TABLE table_3 (<br>  col_1 INTEGER);<br><br>*** Table has been created. | This single-statement DDL request begins an explicit ANSI mode transaction.<br>No further requests are valid within the boundaries of the current transaction. |
| SHOW TABLE table_3;<br><br>*** Error 3722 Only a COMMIT WORK or null statement is legal after a DDL Statement. | The SHOW TABLE request evokes an Error response because the previous CREATE TABLE request cannot be followed by any other requests within the boundaries of the current transaction.<br>The transaction does not roll back. |
| COMMIT;<br><br>*** COMMIT done. | A COMMIT request commits the transaction, which is now complete. |
| SHOW TABLE table_3;<br><br>CREATE MULTISET TABLE DB.table_3 ,NO FALLBACK,<br>  NO BEFORE JOURNAL,<br>  NO AFTER JOURNAL (<br>  col_1 INTEGER)<br>  PRIMARY INDEX ( col_1 ); | A new explicit transaction begins with this SHOW TABLE request, which reports the DDL used to create the table named table_3. |

## 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 the 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 412).
- Explicit transaction boundaries are specified using BEGIN TRANSACTION (BT) and END TRANSACTION (ET) statements (see *SQL Reference: Data Manipulation Statements* 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 Reference: 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 409) 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.
- 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 412).
- Error responses (see *SQL Reference: 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, the 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 412).
- 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 Warehouse* or "2PC" on page 521 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 472).

The set contains the following case studies:



## Teradata Mode Failure (ADD 3-23)

| | |
|---|---|
| BTEQ -- Enter your DBC/SQL request or BTEQ command:<br>BEGIN TRANSACTION;<br><br>*** Begin transaction accepted. | Beginning of an explicit transaction. |
| BTEQ -- Enter your DBC/SQL request or BTEQ command:<br>INSERT INTO employee<br>SELECT *<br>FROM customer_service.employee;<br><br>*** Insert completed. 26 rows added. | Single statement.<br>Single request.<br>WRITE locks held. |
| BTEQ -- Enter your DBC/SQL request or BTEQ command:<br>SELECT *<br>FRM employee<br>WHERE empnum = 401;<br><br>*** Failure 3706 Syntax error; SELECT * must have a FROM clause. | Syntax error is a failure.<br>Transaction rolled back.<br>All previous requests in transaction are also rolled back.<br>All locks released. |
| BTEQ -- Enter your DBC/SQL request or BTEQ command:<br>SELECT *<br>FROM employee;<br><br>*** Query completed. No rows found. | Single statement.<br>Single request.<br>Implicit transaction. |
| BTEQ -- Enter your DBC/SQL request or BTEQ command:<br>END TRANSACTION;<br><br>*** Failure 3510 Too many END TRANSACTION statements. | Request to end the transaction causes a failure response because the transaction begun with the BEGIN TRANSACTION statement had already rolled back. |

## Teradata Mode Requests (ADD 3-25 - 3-27)

| | |
|---|---|
| `DELETE FROM table_1`<br>`WHERE PI_col=2;`<br><br>`INSERT INTO table_1`<br>`VALUES (2,3,4);`<br><br>`UPDATE table_1`<br>`SET col_3=4;` | These three statements are separate requests. They are also implicit transactions. The implications of this are:<br>• The requests are performed serially in the order they are received.<br>• Their locks are applied and released separately.<br>• The success or failure of each has no effect on the success or failure of the others. |
| `BEGIN TRANSACTION;`<br><br>`DELETE FROM table_1`<br>`WHERE PI_col=2;`<br><br>`INSERT INTO table_1`<br>`VALUES (2,3,4);`<br><br>`UPDATE table_1`<br>`SET col_3=4;`<br><br>`END TRANSACTION;` | These five statements are separate requests within a single explicit transaction. The implications of this are:<br>• The requests are performed serially in the order they are specified in the transaction.<br>• 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.<br>• The success or failure of each has a direct effect on the success or failure of the others. |
| `DELETE FROM table_1`<br>`WHERE PI_col=2`<br>`;INSERT INTO table_1`<br>`VALUES (2,3,4)`<br>`;UPDATE table_1`<br>`SET col_3=4;` | These three statements form a single multistatement request. They are also implicitly a single transaction. The implications of this are:<br>• The most restrictive lock held by the transaction, a table-level WRITE lock, is applied to table_1.<br>• The work done by the request is atomic: either it is all committed or it is all rolled back. |
| `CREATE MACRO mac_1 AS (`<br>`DELETE FROM table_1`<br>`WHERE PI_col=2;`<br>`INSERT INTO table_1`<br>`VALUES (2,3,4);`<br>`UPDATE table_1`<br>`SET col_3=4;`<br>`);` | 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. |
| `EXEC mac_1`<br><br>`DELETE FROM table_1`<br>`WHERE PI_col=2`<br>`;INSERT INTO table_1`<br>`VALUES (2,3,4)`<br>`;UPDATE table_1`<br>`SET col_3=4;` | The result of executing the macro is atomic in exactly the same way its equivalent multistatement request is atomic. |
| `EXPLAIN EXEC mac_1`<br><br><br>`EXPLAIN DELETE FROM`<br>`table_1`<br>`WHERE PI_col=2`<br>`;INSERT INTO table_1`<br>`VALUES (2,3,4)`<br>`;UPDATE table_1`<br>`SET col_3=4;` | The EXPLAIN reports generated for these two requests are identical. |

## 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 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 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 statement. 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:

| Rule | ANSI Session Mode | Teradata Session Mode |
|---|---|---|
| Transaction declaration | Always implicit. | Either implicit or explicit. |
| Transaction initiation and termination | Transactions initiate when no transaction is active and an SQL statement is performed.<br>Opened by one of the following:<br>• First SQL statement performed in a session.<br>• First statement performed after the previous transaction terminates.<br>In both cases, the transaction terminates with either a COMMIT [WORK] or ROLLBACK [WORK] (or ABORT) statement. | Opened by one of the following:<br>• Each request performed in a session.<br>• BEGIN TRANSACTION statement.<br>If statements are grouped into explicit transactions in this way, they must be terminated by a matching END TRANSACTION statement.<br>Transaction rollbacks are specified by an ABORT or ROLLBACK statement.<br>The COMMIT statement is not valid. |
| Cursors | Always positioned. | Never positioned. |
| Error behavior | Errors roll back only the request that causes them, *not* the entire transaction.<br>Locks placed by erroneous statements are not released and the Transient Journal is not deleted from the dictionary. | Errors roll back the entire transaction.<br>All locks are released and the Transient Journal is first applied, then deleted from the dictionary. |
| Statement failure behavior | Not applicable.[1] | Statement failures roll back the entire transaction.<br>All locks are released and the Transient Journal is first applied, then deleted from the dictionary. |
| Default attribute for character comparisons | CASESPECIFIC | NOT CASESPECIFIC |
| Control of character truncation errors | Records errors for truncation problems with trailing non-blank characters. | Truncation problems do not cause errors. |

| Rule | ANSI Session Mode | Teradata Session Mode |
|---|---|---|
| Default table type semantics | MULTISET | SET |
| Default TRIM function behavior | Trims both leading and trailing pad characters as if TRIM (BOTH FROM *expression*) were explicitly specified. | Trims only trailing pad characters as if TRIM(TRAILING FROM *expression*) were explicitly specified. |
| DDL statement placement | Must be last statement in transaction. | Must be last statement in transaction. |
| Statements performed through a logon startup string | Not applicable. | Must follow standard rules for:<br>• Transaction definition<br>• Transaction termination<br>• DDL statement placement |

1. 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[32] asynchronous ABORT causes full transaction rollback in both session modes. This occurs by means of a CLIv2 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.

## 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.

32. The “application” in this case is a component of the database management system, whether client-based or server-based.

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

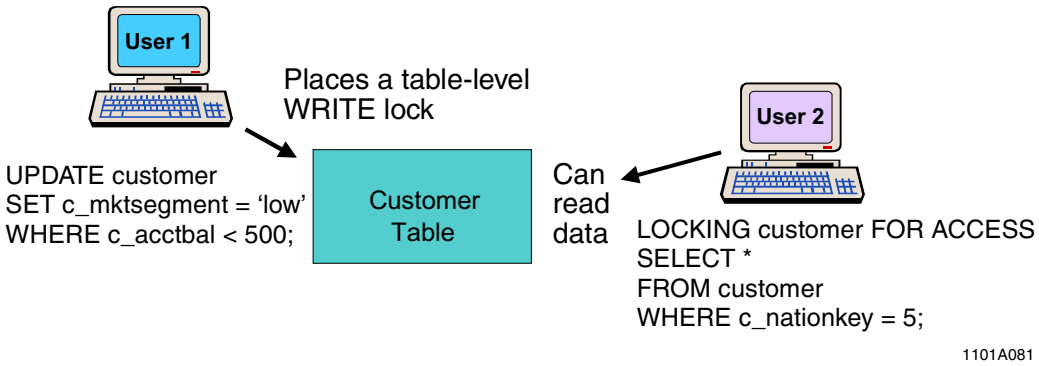
Group AMP operations use a series of row hash-level locks, one for each of the rows touched by the query. See *Database Design* for more information about group AMP operations.

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;
```

```
Explanation
------------------------------------------------------------------------
  1) First, we lock CAB.customer for access.
  2) Next, we do a single-AMP RETRIEVE step from CAB.customer by
     way of the unique primary index "CAB.customer.C_CUSTKEY =
     93522" with no residual conditions.  The estimated time for this
     step is 0.03 seconds.
  3) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

For a single-AMP operation such as reading a single row using a primary index value, LOCKING ROW FOR ACCESS is always a better locking modifier to use than locking the entire table. You can see in the EXPLAIN report that a row-level ACCESS lock is applied and that only one AMP is used to process the query.

The following EXPLAIN text illustrates a row-level ACCESS lock:

```
EXPLAIN
LOCKING ROW FOR ACCESS
SELECT c_name, c_acctbal
FROM customer
WHERE c_custkey = 93522;
```

```
Explanation
------------------------------------------------------------------------
  1) First, we do a single-AMP RETRIEVE step from CAB.customer by
     way of the unique primary index "CAB.customer.C_CUSTKEY =
     93522" with no residual conditions locking row for access.  The
     estimated time for this step is 0.03 seconds.
```

## Automatic Lock Escalation

Even if you explicitly specify a row hash-level ACCESS lock, the Optimizer automatically converts it to a table-level lock if the query plan requires an all-AMPs operation and there is only one table referenced in the query. Always check the EXPLAIN report to verify that row hash or, when appropriate, table-level ACCESS locks are issued for tactical queries.

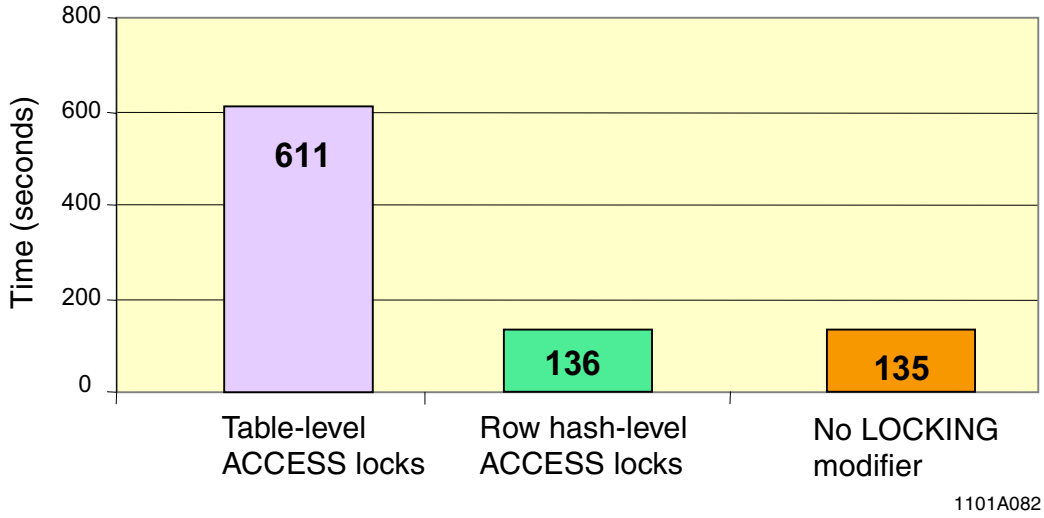
The following EXPLAIN report for an all-AMPs query shows that the Optimizer applies a table-level ACCESS lock even though the LOCKING modifier explicitly requests a row hash-level ACCESS lock.

```
EXPLAIN
LOCKING ROW FOR ACCESS
SELECT c_name, c_acctbal
FROM customer
WHERE c_nationkey = 15;
```

```
Explanation
------------------------------------------------------------------------
1) First, we lock CAB.customer for access.
2) Next, we do an all-AMPs RETRIEVE step from CAB.customer by way
   of an all-rows scan with a condition of ("CAB.customer.C_NATIONKEY = 15")
   into Spool 1, which is built locally on the AMPs. The input table will not be cached
   in memory, but it is eligible for synchronized scanning.  The size of Spool 1
   is estimated with high confidence to be 300,092 rows.  The estimated time for
   this step is 2 minutes and 8 seconds.
3) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
```

Even if more than one table is specified in the query, if a LOCKING ROW FOR ACCESS modifier has been specified, the Optimizer applies a table-level ACCESS lock for all tables undergoing all-AMP access in that query.

The following graph illustrates the cost of using table-level ACCESS locks compared to row hash-level ACCESS locks when the query itself only performs single-AMP operations:




The elapsed time represents the total time to perform 100 000 single-row SELECT statements using 20 sessions. When table-level ACCESS locks were specified in the query, total execution time for this workload increased by a factor of 5. The response times for the variables labeled Row hash-level ACCESS locks and No locking modifier are almost identical because when no locking modifier was specified, the Optimizer applied a row hash-level READ lock in the background. This lock has the same overhead as the row-level ACCESS lock.

Some query tools make it difficult to specify an ACCESS lock modifier. In spite of this, you can enforce explicit ACCESS locking by querying views and placing the appropriate LOCKING modifier in their view definitions.

## Row 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.

```
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:

```
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 = 832094" locking
     row of CAB.ordertbl for access and row of CAB.lineitem for
     access.  CAB.ordertbl and CAB.lineitem are joined using a merg
     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:

```
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 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 492. 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:

```
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

| Topic | Reference |
|---|---|
| General transaction processing references | Philip Bernstein and Eric Newcomer, *Principles of Transaction Processing*. San Francisco: Morgan Kaufmann Publishers, 1997.<br><br>A more introductory text than the Gray-Reuter and Weikum-Vossen books. |
| | Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley Publishing Company, 1987.<br><br>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 from two different sources as a PDF file:<br>• As a free download at the following URL: http://research.microsoft.com/pubs/ccontrol/<br>• 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.<br>You can write or call the ACM offices at the following address and phone number:<br>Association for Computing Machinery<br>1515 Broadway, New York, New York 10036<br>USA<br>1-800-342-6626 (USA and Canada) or +212-626-0500 (Global)<br>You can also purchase the item online at http://www.acm.org or http://www.sigmod.org.<br>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*. |
| | Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann Publishers, 1993.<br><br>Together with the Weikum-Vossen volume, the ultimate reference for all things related to database transaction processing. |
| | Christos H. Papadimitriou, *The Theory of Database Concurrency Control*. Rockville, MD: Computer Science Press, 1986.<br><br>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.<br>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. |

| Topic | Reference |
|---|---|
| General transaction processing references (continued) | Gerhard Weikum and Gottfried Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco: Morgan Kaufmann Publishers, 2001.<br><br>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.<br><br>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. |
| Origins of the concept of transaction management in relational database management systems | Charles T. Davies, Jr., "Recovery Semantics for a DB/DC System," *Proceedings 1973 ACM National Conference*, pp. 135-141, 1973. |
| | Lawrence A. Bjork, "Recovery Scenario for a DB/DC System," *Proceedings 1973 ACM National Conference, pp. 142-146*, 1973.<br><br>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. |
| | Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, 13(2):223-242, 1981.<br><br>Published earlier as "The Recovery Manager of a Data Management System," *IBM Research Report RJ 2623*, IBM Research Laboratories, San Jose, California, June, 1979.<br><br>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.<br><br>The design of the transaction management system was based on the earlier work of Bjork (1973) and Davies (1973). |

| Topic | Reference |
|---|---|
| Two-Phase Locking | K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traeger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, 19(11):624-633, 1976.<br>Published earlier as "On the Notions of Consistency and Predicate Locks in a Data Base System," *IBM Research Report RJ 1487*, IBM Research Laboratories, San Jose, California, December, 1974.<br>Among its many innovations, this paper introduced the critical concepts of two-phase locking and lock granularity. |
| | J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traeger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base." In: G.M. Nijssen (ed.) *IFIP Working Conference on Modeling of Data Base Management Systems*. Amsterdam, North-Holland, 1976 pp. 365-394.<br>Published earlier as J.N. Gray, R.A. Lorie, and G.R. Putzolu, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," *IBM Research Report RJ 1654*, IBM Research Laboratories, San Jose, California, September, 1975.<br>Further develops the concept of a hierarchy of locking granularities as an alternative to the predicate locks introduced by Eswaran et al. (1976). |
| | Irving L. Traiger, James N. Gray, Cesare A. Galtieri, and Bruce G. Lindsay, "Transactions and Consistency in Distributed Database Systems," *IBM Research Report RJ 2555*, IBM Research Laboratories, San Jose, California, June, 1979.<br>Introduces strict 2PL, in which no locks are dropped until a transaction either commits or rolls back. |
| ACID | Theo Haerder and Andreas Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, 15(4):287-317, 1983.<br>Haerder (sometimes spelled Härder) and Reuter review the principles of transaction management in relational database management systems, introducing the concept of ACID (see "The ACID Properties of Transactions" on page 408) in the process. |
| Two-Phase Commit Protocol | Butler W. Lampson and Howard E. Sturgis, "Crash Recovery in a Distributed Data Storage System," *Technical Report*, Xerox Palo Alto Research Center, Palo Alto, California, 1976.<br>Develops the concept of two-phase commit (2PC).<br>Unfortunately, this work was never published in the refereed literature, though it can be obtained from the Publications section of Lampson's personal web site at http://research.microsoft.com/Lampson/21-CrashRecovery/WebPage.html.<br>Some of this work was eventually published as Butler W. Lampson and Howard E. Sturgis, "Atomic Transactions," in B. Lampson, M. Paul, and H. Siegert (eds.) *Distributed Systems-Architecture and Implementation: Lecture Notes in Computer Science 105*, Berlin: Springer-Verlag, pp. 246-265, 1981.<br>Note that the 2PC protocol has nothing to do with two-phase locking (2PL) despite their similar sounding names. |

| Topic | Reference |
|---|---|
| Replication | Philip A. Bernstein and Nathan Goodman, "The Failure and Recovery Problem for Replicated Databases," *Proceedings of 2nd ACM Symposium on Principles of Distributed Computing*, pp. 114-122, 1983.<br><br>Philip A. Bernstein and Nathan Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, 12(2):185-221, 1981.<br><br>Maurice P. Herlihy and Jeannette M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.<br><br>Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha, "The Dangers of Replication and a Solution," *Proceedings of ACM SIGMOD 1996*, pp. 173-182, 1996. |
| Optimistic Replication | Paul R. Johnson and Robert H. Thomas, "The Maintenance of Duplicate Databases," *Network Working Group RFC # 677*, 1975. Available at http://www.faqs.org/rfcs/rfc677.html.<br><br>A classic paper on the subject. |
| | Yasushi Saito and Marc Shapiro, "Optimistic Replication," *ACM Computing Surveys*, 37(1):42-81, 2005.<br><br>A survey paper on optimistic replication. |
| Livelock | Jeffrey D. Ullman, *Principles of Database Systems* (2nd ed.). Rockville, Maryland: Computer Science Press, 1982.<br><br>This is perhaps the only place you will ever see livelock mentioned.<br><br>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. |

| Topic | Reference |
|---|---|
| The Date critique of the use of consistency in the ACID initialism | C.J. Date, *An Introduction to Database Systems* (8th ed.). Boston, MA: Addison-Wesley, 2004.<br><br>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.<br><br>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.<br><br>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).<br><br>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." |
| | Chris Date, "On 'ACID' and Multiple Assignment," September 29, 2003. Available at the following URL: http://www.dbdebunk.com/page/page/806828.htm.<br><br>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 413), a construct that is unique to the Teradata Database.<br><br>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. |

| Topic | Reference |
|---|---|
| Transaction Isolation Levels | J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traeger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base." In: G.M. Nijssen (ed.) *IFIP Working Conference on Modeling of Data Base Management Systems*. Amsterdam, North-Holland, 1976 pp. 365-394.<br>Published earlier as "On the Notions of Consistency and Predicate Locks in a Data Base System," *IBM Research Report RJ 1654*, IBM Research Laboratories, San Jose, California, September, 1975.<br>This paper introduced the concept of transaction isolation levels under the name "degrees of consistency." |
| | Jim Gray, "A Transaction Model," in: *Source Lecture Notes In Computer Science, Vol. 85: Proceedings of the 7th Colloquium on Automata, Languages and Programming*. Berlin: Springer-Verlag, 1980.<br>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. |
| | Atul Adya, Barbara Liskov, and Patrick O'Neil, "Generalized Isolation Level Definitions," *Proceedings of 16th International Conference on Data Engineering*, pp. 67-78, 2000.<br>Presents an update to Berenson et al., (1995) with a serious attempt to provide a complete, flexible, implementation-independent definition of all possible isolation levels. |
| | Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha, "Making Snapshot Isolation Serializable," *ACM Transactions on Database Systems*, 30(2):492-528, 2005.<br>Analyzes static dependencies between concurrently running applications to determine the conditions under which so-called "snapshot isolation" is also serializable.<br>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 500 for an important reference regarding this line of research. |
| | Alan Fekete, "Allocating Isolation Levels to Transactions," *Proceedings of PODS 2005*, 206-215, 2005.<br>A theoretical discussion of isolation levels. |

| Topic | Reference |
|---|---|
| The critique of the ANSI SQL standard transaction isolation levels | Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil, "A Critique of ANSI SQL Isolation Levels," *Proceedings of SIGMOD 1995*, 1-10, 1995.<br><br>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.<br><br>The paper is also the initial exposition of the snapshot isolation level first defined by O'Neil and O'Neil. |
| | C.J. Date with Hugh Darwen, *A Guide to the SQL Standard* (4th ed.). Boston, MA: Addison-Wesley, 1997.<br><br>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)."<br><br>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. |
| | Patrick O'Neil, *Course Notes for CS735, Database Internals*. Department of Computer Science, University of Massachusetts, Boston, 2004.<br><br>A PDF file of these notes is available at http://www.cs.umb.edu/cs734/.<br><br>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. |

| Topic | Reference |
|---|---|
| The critique of the ANSI SQL standard transaction isolation levels (continued) | Patrick O'Neil and Elizabeth O'Neil, *Project Summary for Isolation Testing*. NSF Project Proposal, University of Massachusetts, Boston, 15pp., 1999.<br>Available from the Publications section of O'Neil's personal web site at http://www.cs.umb.edu/~poneil/publist.html.<br>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.<br>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.<br>"But absolutely no support is given the DBA as to how to perform such analysis."<br>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.<br>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." |

| Topic | Reference |
|---|---|
| Concurrency and performance | R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal on Applied Mathematics*, 14(6):1390-1411, 1966.<br><br>The paper that first established the basic concepts underlying all ensuing research on concurrency control. |
| | Y.C. Tay, Nathan Goodman, and Rajan Suri, "Locking Performance in Centralized Databases," *ACM Transactions on Database Systems*, 10(4):415-462, 1985.<br><br>Shows that blocking due to lock conflicts is the factor that imposes the upper bound on transaction throughput.<br><br>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. |
| | Y.C. Tay, *Locking Performance in Centralized Databases*. Boston, MA: Academic Press, Inc., 1987.<br><br>The title says it all.<br><br>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.<br><br>The material is highly quantitative, but clearly reported and summarized. |
| | Alexander Thomasian, "Concurrency Control: Methods, Performance, and Analysis," *ACM Computing Surveys*, 30(1):70-119, 1998.<br><br>Reviews a plethora of quantitative data about concurrency control. The coverage is not restricted to relational database management applications. |
| | Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price, "The Convoy Phenomenon," *ACM Operating Systems Review*, 13(2):20-28, 1979.<br><br>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." |

| Topic | Reference |
|---|---|
| Optimistic Concurrency Control | H.T. Kung and John T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, 6(2):213-226, 1981.<br><br>The original paper to propose OCC as an alternative to locking mechanisms for transaction management. |
| | Peter Franaszek and John T. Robinson, "Limitations of Concurrency in Transaction Processing," *ACM Transactions on Database Systems*, 10(1):1-28, 1985.<br><br>A downward revision of the general applicability of OCC for transaction management from the IBM laboratory that had originally proposed the concept. |
| | Theo Haerder, "Observations on Optimistic Concurrency Control Schemes," *Information Systems*, 9(2):111-120, 1984.<br><br>The first important paper to point out many of the problems with OCC in a realistic production environment. |
| | C. Mohan, "Less Optimism About Optimistic Concurrency Control," *Proceedings of 2nd International Workshop on Research Issues on Data Engineering 1992*, 199-204, 1992.<br><br>Extends the observations of Haerder (1984) and introduces several additional concerns with OCC.<br><br>For anyone wondering what Mohan's first name is, the riddle is solved by this explanation at his personal IBM web site:<br><br>"**The first name puzzle:** 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." |

| Topic | Reference |
|---|---|
| Write Ahead Logging | C. Mohan and Frank Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," *Proceedings of SIGMOD 1992*, 371-380, 1992.<br><br>Published earlier as C. Mohan and Frank Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," *IBM Research Report RJ6846*, IBM Almaden Research Center, San Jose, California, June, 1989.<br><br>A comprehensive review of index management (the characters IM in ARIES/IM represent Index Management) by transaction processing systems that introduces WAL to the mix.<br><br>The internal IBM technical report is available from the Publications page of Mohan's personal web site at http://www.almaden.ibm.com/u/mohan/RJ6846.pdf. |
| | C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions on Database Systems*, 17(1):94-162, 1992.<br><br>The first widely available presentation of the IBM ARIES concept of WAL. The initialism ARIES represents Algorithm for Recovery and Isolation Exploiting Semantics. |
| | K. Rothermel and C. Mohan, "ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions," *Proceedings of 15th International Conference on Very Large Data Bases*, pp. 337-346, 1989.<br><br>Another early presentation of the ARIES concept, this time devoted to nested transactions (the characters NT in ARIES/NT represent Nested Transactions). |

# APPENDIX A Notation Conventions

This appendix describes the notation conventions used in this book.

Throughout this book, three conventions are used to describe the SQL syntax and code:

- Syntax diagrams, used to describe SQL syntax form, including options. See "Syntax Diagram Conventions" on page 506.
- 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 510.

Symbols from the predicate calculus are also used occasionally to describe logical operations. See "Predicate Calculus Notation Used in This Book" on page 512.

# Syntax Diagram Conventions

## Notation Conventions

The following table defines the notation used in this section:

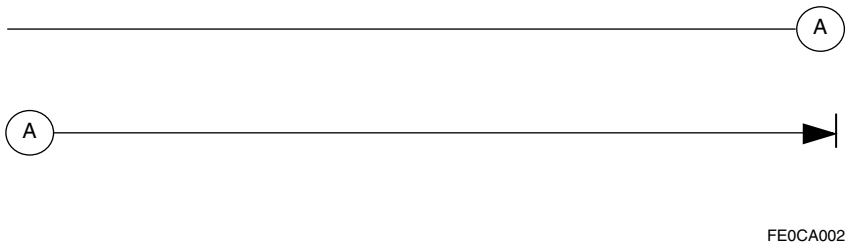
<table>
<tr><th>Item</th><th>Definition / Comments</th></tr>
<tr><td>Letter</td><td>An uppercase or lowercase alphabetic character ranging from A through Z.</td></tr>
<tr><td>Number</td><td>A digit ranging from 0 through 9.<br>Do not use commas when entering a number with more than three digits.</td></tr>
<tr><td>Word</td><td>Variables and reserved words.
<table>
<tr><th>IF a word is shown in ...</th><th>THEN it represents ...</th></tr>
<tr><td>UPPERCASE LETTERS</td><td>a keyword.<br>Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase.<br>If a keyword is shown in uppercase, you may enter it in uppercase or mixed case.</td></tr>
<tr><td>lowercase letters</td><td>a keyword that you must enter in lowercase, such as a UNIX command.</td></tr>
<tr><td>lowercase italic letters</td><td>a variable such as a column or table name.<br>You must substitute a proper value.</td></tr>
<tr><td>lowercase bold letters</td><td>a variable that is defined immediately following the diagram that contains it.</td></tr>
<tr><td><u>UNDERLINED LETTERS</u></td><td>the default value.<br>This applies both to uppercase and to lowercase words.</td></tr>
</table></td></tr>
<tr><td>Spaces</td><td>Use one space between items, such as keywords or variables.</td></tr>
<tr><td>Punctuation</td><td>Enter all punctuation exactly as it appears in the diagram.</td></tr>
</table>

## Paths

The main path along the syntax diagram begins at the left, and proceeds, left to right, to the vertical bar, which marks the end of the diagram. Paths that do not have an arrow or a vertical bar only show portions of the syntax.

The only part of a path that reads from right to left is a loop.

Paths that are too long for one line use continuation links. Continuation links are small circles with letters indicating the beginning and end of a link:




When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue.

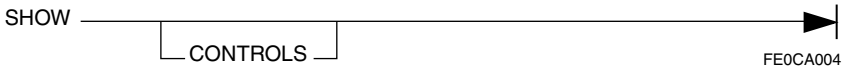
## Required Items

Required items appear on the main path:


If you can choose from more than one item, the choices appear vertically, in a stack. The first item appears on the main path:


## Optional Items

Optional items appear below the main path:




If choosing one of the items is optional, all the choices appear below the main path:


You can choose one of the options, or you can disregard all of the options.

## Abbreviations

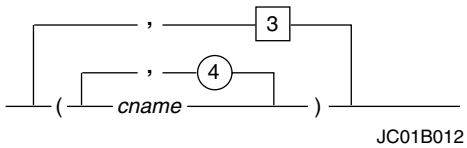
If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.


In the above syntax, the following formats are valid:

- SHOW CONTROLS
- SHOW CONTROL

## Loops

A loop is an entry or a group of entries that you can repeat one or more times. Syntax diagrams show loops as a return path above the main path, over the item or items that you can repeat.



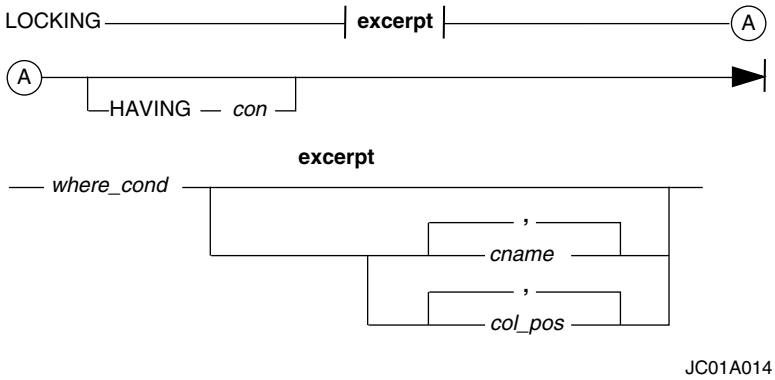

The following rules apply to loops:

| IF ... | THEN the ... |
|---|---|
| there is a maximum number of entries allowed | number appears in a circle on the return path.<br>In the example, you may enter cname a maximum of 4 times. |
| there is a minimum number of entries required | number appears in a square on the return path.<br>In the example, you must enter at least 3 groups of column names. |
| a separator character is required between entries | character appears on the return path.<br>If the diagram does not show a separator character, use one blank space.<br>In the example, the separator character is a comma. |
| a delimiter character is required around entries | beginning and end characters appear outside the return path.<br>Generally, a space is not needed between delimiter characters and entries.<br>In the example, the delimiter characters are the left and right parentheses. |

## Excerpts

Sometimes a piece of a syntax phrase is too large to fit into the diagram. Such a phrase is indicated by a break in the path, marked by | terminators on either side of the break. A name for the excerpted piece appears between the break marks in boldface type.

The named phrase appears immediately after the complete diagram, as illustrated by the following example.



# Character Shorthand Notation Used In This Book

## Introduction

This book uses the UNICODE naming convention for characters. For example, the lowercase character ‘a’ is more formally specified as either LATIN SMALL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the book, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings.

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in *International Character Set Support*.

## Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

| Symbol | Encoding | Meaning |
|---|---|---|
| a–z<br>A–Z<br>0–9 | Any | Any single byte Latin letter or digit. |
| a–z<br>A–Z<br>0–9 | Unicode compatibility zone | Any fullwidth Latin letter or digit. |
| < | KanjiEBCDIC | Shift Out [SO] (0x0E).<br>Used to indicate transition from single to multibyte character in KanjiEBCDIC. |
| > | KanjiEBCDIC | Shift In [SI] (0x0F).<br>Used to indicate transition from multibyte to single byte KanjiEBCDIC. |
| **T** | Any | Any multibyte character.<br>Its encoding depends on the current character set.<br>For KanjiEUC, code set 3 characters are sometimes shown preceded by “$ss_3$”. |

| Symbol | Encoding | Meaning |
|---|---|---|
| **I** | Any | Any single byte Hankaku Katakana character.<br>In KanjiEUC, it must be preceded by "$ss_2$", forming an individual multibyte character. |
| Δ | Any | Represents the graphic pad character. |
| Δ | Any | Represents either a single or multibyte pad character, depending on context. |
| $ss_2$ | KanjiEUC | Represents the EUC code set 2 introducer (0x8E). |
| $ss_3$ | KanjiEUC | Represents the EUC code set 3 introducer (0x8F). |

For example, string "TEST", where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set

LMN<**TEST**>QRS

is represented as:

```
D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2
```

## Pad Characters

The following table lists the pad characters for the various character data types.

| Character Data Type | Pad Character Name | Pad Character Value |
|---|---|---|
| LATIN | SPACE | 0x20 |
| UNICODE | SPACE | U+0020 |
| GRAPHIC | IDEOGRAPHIC SPACE | U+3000 |
| KANJISJIS | ASCII SPACE | 0x20 |
| KANJI1 | ASCII SPACE | 0x20 |

# Predicate Calculus Notation Used in This Book

Relational databases are based on the theory of relations as developed in set theory. Predicate calculus is often the most unambiguous way to express certain relational concepts.

Occasionally this book uses the following predicate calculus notation to explain concept:

| This symbol ... | Represents this phrase ... |
|---|---|
| iff | If and only if |
| ∀ | For all |
| ∃ | There exists |
| ∅ | Empty set |

# APPENDIX B References

This appendix is a compilation of the individual reference sections for the topics of query processing and transaction processing in Chapter 2: "Query Optimization" and Chapter 8: "Locking and Transaction Processing," respectively.

The two topics are presented in separate reference lists as follows:


- "Query Processing References" on page 514
- "Transaction Processing References" on page 517


There is a very slight overlap between the contents of the two lists.

# Query Processing References

M.M. Astrahan, M. Schkolnick, and W. Kim, “Performance of the System R Access Path Selection Mechanism,” *Information Processing 80*, IFIP, North-Holland Publishing Co., 1980, pp. 487-491.

M.W. Blasgen and K.P. Eswaran, “Storage and Access in Relational Data Bases,” *IBM Systems Journal*, 16(4):363-377, 1977.

Béla Bollobás, *Modern Graph Theory*, Berlin: Springer-Verlag, 1998.

J.A. Bondy and U.S.R. Murty, *Graph Theory With Applications*, Amsterdam: North-Holland, 1976.
Available as a free PDF file from the following URL: http://www.ecp6.jussieu.fr/pageperso/bondy/books/gtwa/pdf/GTWA.pdf.

Stephen Brobst and Dominic Sagar, “The New, Fully Loaded, Optimizer,” *DB2 Magazine*, 4(3), 1999.
http://www.db2mag.com/db_area/archives/1999/q3/brobst.shtml.

Stephen Brobst and Bob Vecchione, “Starburst Grows Bright,” *Database Programming and Design*, 11(2), 1998.

Philip Y. Chang, “Parallel Processing and Data Driven Implementation of a Relational Data Base System,” *Proceedings of 1976 ACM National Conference*, 314-318, 1976.

Gary Chartrand, *Introductory Graph Theory*, Mineola, NY: Dover Publishing Co., 1984. Republication of Boston, MA: Prindle, Weber, and Schmidt, Inc., *Graphs as Mathematical Models*, 1977.

Surajit Chaudhuri, "An Overview of Query Optimization in Relational Systems," *Proceedings of PODS 1998*, 34-43, 1998.

Richard L. Cole, Mark J. Anderson, and Robert J. Bestgen, “Query Processing in the IBM Application System 400,” *IEEE Data Engineering Bulletin*, 16(4):18-27, 1993.

C.J. Date, *An Introduction to Database Systems* (8th ed.), Reading, MA: Addison-Wesley, 2004.

Reinhard Diestel, *Graph Theory* (2nd ed.), Berlin: Springer-Verlag, 2000.
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.
Note that the PDF version of the book cannot be printed.

Shimon Even, *Graph Algorithms*, Rockville, MD: Computer Science Press, 1979.

Peter Gassner, Guy Lohman, and K. Bernhard Schiefer, “Query Optimization in the IBM DB2 Family,” *IEEE Data Engineering Bulletin*, 16(4):4-17, 1993.

Philip B. Gibbons and Yossi Matias, "Synopsis Data Structures for Massive Data Sets," in: *External Memory Systems: DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, 1998.

Goetz Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2):73-170, 1993.

P.A.V. Hall, “Optimization of Single Expressions in a Relational Data Base System,” *IBM Journal of Research and Development*, 20 (3):244-257, 1976.

Waqar Hasan, *Optimization of SQL Queries for Parallel Machines*, Ph.D. dissertation, Department of Computer Science, Stanford University, 1996.
Available as a PostScript file at the following URL: http://www-db.stanford.edu/pub/hasan/1995/thesis.ps.

Waqar Hasan, *Optimization of SQL Queries for Parallel Machines*, Berlin: Springer-Verlag, 1996.

Toshihide Ibaraki and Tiko Kameda, “On the Optimal Nesting Order for Computing N-Relational Joins,” *ACM Transactions on Database Systems*, 9(3):482-502, 1984.

Yannis E. Ioannidis, “The History of Histograms (abridged),” *Proceedings of VLDB Conference*, 2003.

Yannis E. Ioannidis and Stavros Christodoulakis, “On the Propagation of Errors in the Size of Join Results,” *Proceedings of SIGMOD 1991*, 268-277, 1991.

Yannis E. Ioannidis and Stavros Christodoulakis, "Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results," *ACM Transactions on Database Systems*, 18(4):709-448, 1993.

Matthias Jarke and Jürgen Koch, "Query Optimization in Database Systems," *ACM Computing Surveys*, 16(2):111-152, 1984.

Donald Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd. ed.), Reading, MA: Addison-Wesley, 1997.

Edmund Landau, *Foundations of Analysis* (3rd ed.), London: Chelsea Publishing Co., 2001. Translation by F. Steinhardt of the third German edition *Grundlagen der Analysis*, Berlin: Heldermann Verlag, 1970.

David Maier, *The Theory of Relational Databases*, Rockville, MD: Computer Science Press, 1983.

Michael V. Mannino, Paicheng Chu, and Thomas Sager, "Statistical Profile Estimation in Database Systems," *ACM Computing Surveys*, 20(3):191-221, 1988.

Priti Mishra and Margaret H. Eich, “Join Processing in Relational Databases,” *ACM Computing Surveys*, 24(1):63-113, 1992.

Kiyoshi Ono and Guy M. Lohman, “Measuring the Complexity of Join Enumeration in Query Optimization,” *Proceedings of 16th International Conference on Very Large Data Bases*, 314-325, 1990.

Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *Proceedings of SIGMOD 1996*, 294-305, 1996.

P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of SIGMOD 1979*, 23-34, 1979.

Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T.Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan, "Cost-Based Optimization for Magic: Algebra and Implementation," *Proceedings of SIGMOD 1996*, 435-446, 1996.

Michael Sipser, *Introduction to the Theory of Computation* (2nd ed.), Boston, MA: Thomson Course Technology, 2005.

John Miles Smith and Philip Yen-Tang Chang, "Optimizing the Performance of a Relational Algebra Database Interface," *Communications of the ACM*, 18(10):568-579, 1975.

SQL Summit, *Query Optimization: Articles, Papers, Podcasts, Webcasts*. A collection of links to several papers and presentations on several basic aspects of query optimization. http://www.sqlsummit.com/Optimization.htm.

Michael Stonebraker, "Retrospection on a Database System," *ACM Transactions on Database Systems*, 5(2):225-240, 1980.

Michael Stonebraker, Eugene Wong, and Peter Kreps, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems*, 1(3):189-222, 1976.

S.J.P. Todd, "The Peterlee Relational Test Vehicle—a system overview," *IBM Systems Journal*, 15(4):285-308, 1976.

Richard J. Trudeau, *Introduction to Graph Theory*, Mineola, NY: Dover Publishing Co., 1994. Corrected and enlarged edition of Kent, OH: The Kent State University Press, *Dots and Lines*, 1976.

Jeffrey D. Ullman, *Principles of Database Systems* (2nd ed.), Rockville, MD: Computer Science Press, 1982.

Eugene Wong and Karel Youssefi, "Decomposition—A Strategy for Query Processing," *ACM Transactions on Database Systems*, 1(3):223-241, 1976.

Karel Youssefi and Eugene Wong, "Query Processing In a Relational Database Management System," *Proceedings of 5th International Conference on Very Large Data Bases*, 409-417, 1979.

Clement T. Yu and Weiyi Meng, *Principles of Database Query Processing for Advanced Applications*, San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1998.

# Transaction Processing References

Atul Adya, Barbara Liskov, and Patrick O'Neil, "Generalized Isolation Level Definitions," *Proceedings of 16th International Conference on Data Engineering*, pp. 67-78, 2000.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil, "A Critique of ANSI SQL Isolation Levels," *Proceedings of ACM SIGMOD 1995*, 1-10, 1995.

Philip A. Bernstein and Nathan Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, 12(2):185-221, 1981.

Philip A. Bernstein and Nathan Goodman, "The Failure and Recovery Problem for Replicated Databases," *Proceedings of 2nd ACM Symposium on Principles of Distributed Computing*, pp. 114-122, 1983.

Philip Bernstein and Eric Newcomer, *Principles of Transaction Processing*. San Francisco: Morgan Kaufmann Publishers, 1997.

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley Publishing Company, 1987.

Lawrence A. Bjork, "Recovery Scenario for a DB/DC System," *Proceedings 1973 ACM National Conference, pp. 142-146*, 1973.

Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price, "The Convoy Phenomenon," *ACM Operating Systems Review*, 13(2):20-28, 1979.

C.J. Date, *An Introduction to Database Systems* (8th ed.). Boston, MA: Addison-Wesley, 2004.

Chris Date, "On 'ACID' and Multiple Assignment," September 29, 2003. Available at the following URL: http://www.dbdebunk.com/page/page/806828.htm.

C.J. Date with Hugh Darwen, *A Guide to the SQL Standard* (4th ed.). Boston, MA: Addison-Wesley, 1997.

Charles T. Davies, Jr., "Recovery Semantics for a DB/DC System," *Proceedings 1973 ACM National Conference*, pp. 135-141, 1973.

K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traeger, "On the Notions of Consistency and Predicate Locks in a Database System," *IBM Research Report RJ 1487*, IBM Research Laboratories, San Jose, California, December, 1974.

K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traeger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, 19(11):624-633, 1976.

Alan Fekete, "Allocating Isolation Levels to Transactions," *Proceedings of PODS 2005*, 206-215, 2005.

Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha, "Making Snapshot Isolation Serializable," *ACM Transactions on Database Systems*, 30(2):492-528, 2005.

Peter Franaszek and John T. Robinson, "Limitations of Concurrency in Transaction Processing," *ACM Transactions on Database Systems*, 10(1):1-28, 1985.

Jim Gray, “A Transaction Model,” in: *Source Lecture Notes In Computer Science, Vol. 85: Proceedings of the 7th Colloquium on Automata, Languages and Programming*. Berlin: Springer-Verlag, 1980.

Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann Publishers, 1993.

J.N. Gray, R.A. Lorie, and G.R. Putzolu, “Granularity of Locks and Degrees of Consistency in a Shared Data Base,” *IBM Research Report RJ 1654*, IBM Research Laboratories, San Jose, California, September, 1975.

J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traeger, “Granularity of Locks and Degrees of Consistency in a Shared Data Base.” In: G.M. Nijssen (ed.) *IFIP Working Conference on Modeling of Data Base Management Systems*. Amsterdam, North-Holland, 1976 pp. 365-394.

Jim Gray, Pat Helland, Patrick O’Neil, and Dennis Shasha, “The Dangers of Replication and a Solution,” *Proceedings of ACM SIGMOD 1996*, pp. 173-182, 1996.

Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger, “The Recovery Manager of a Data Management System,” *IBM Research Report RJ 2623*, IBM Research Laboratories, San Jose, California, June, 1979.

Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, 13(2):223-242, 1981.

Theo Haerder, “Observations on Optimistic Concurrency Control Schemes,” *Information Systems*, 9(2):111-120, 1984.

Theo Haerder and Andreas Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, 15(4):287-317, 1983.

Maurice P. Herlihy and Jeannette M. Wing, “Linearizability: A Correctness Condition for Concurrent Objects,” *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

Paul R. Johnson and Robert H. Thomas, “The Maintenance of Duplicate Databases,” *Network Working Group RFC # 677*, 1975. Available at http://www.faqs.org/rfcs/rfc677.html.

R.M. Karp and R.E. Miller, “Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing,” *SIAM Journal on Applied Mathematics*, 14(6):1390-1411, 1966.

H.T. Kung and John T. Robinson, “On Optimistic Methods for Concurrency Control,” *ACM Transactions on Database Systems*, 6(2):213-226, 1981.

Butler W. Lampson and Howard E. Sturgis, “Crash Recovery in a Distributed Data Storage System,” *Technical Report*, Xerox Palo Alto Research Center, Palo Alto, California, 1976.

Jim Melton (ed.), *International Standard ISO/IEC 9075-2 (5th ed.), Part 2: Foundation (SQL/Foundation)*. Geneva, Switzerland: ISO/IEC/ANSI, 2003.

C. Mohan, “Less Optimism About Optimistic Concurrency Control,” *Proceedings of 2nd International Workshop on Research Issues on Data Engineering 1992*, 199-204, 1992.

C. Mohan and Frank Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," *IBM Research Report RJ6846*, IBM Almaden Research Center, San Jose, California, June, 1989.

C. Mohan and Frank Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," *Proceedings of SIGMOD 1992*, 371-380, 1992.

C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions on Database Systems*, 17(1):94-162, 1992.

Patrick O'Neil, *Course Notes for CS735, Database Internals*. Department of Computer Science, University of Massachusetts, Boston, 2004.

Patrick O'Neil and Elizabeth O'Neil, *Project Summary for Isolation Testing*. NSF Project Proposal, University of Massachusetts, Boston, 15pp., 1999.

Christos H. Papadimitriou, *The Theory of Database Concurrency Control*. Rockville, MD: Computer Science Press, 1986.

K. Rothermel and C. Mohan, "ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions," *Proceedings of 15th International Conference on Very Large Data Bases*, pp. 337-346, 1989.

Yasushi Saito and Marc Shapiro, "Optimistic Replication," *ACM Computing Surveys*, 37(1):42-81, 2005.

Y.C. Tay, *Locking Performance in Centralized Databases*. Boston, MA: Academic Press, Inc., 1987.

Y.C. Tay, Nathan Goodman, and Rajan Suri, "Locking Performance in Centralized Databases," *ACM Transactions on Database Systems*, 10(4):415-462, 1985.

Alexander Thomasian, "Concurrency Control: Methods, Performance, and Analysis," *ACM Computing Surveys*, 30(1):70-119, 1998.

Irving L. Traiger, Janes N. Gray, Cesare A. Galtieri, and Bruce G. Lindsay, "Transactions and Consistency in Distributed Database Systems," *IBM Research Report RJ 2555*, IBM Research Laboratories, San Jose, California, June, 1979.

Jeffrey D. Ullman, *Principles of Database Systems* (2nd ed.). Rockville, Maryland: Computer Science Press, 1982.

Gerhard Weikum and Gottfried Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco: Morgan Kaufmann Publishers, 2001.

# Glossary

**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 roll back.

**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.

**ACID** Atomicity, Consistency, Isolation, Durability

**ACM** Association for Computing Machinery (http://www.acm.org)

The leading US-based professional society for computer professionals.

**AMP** Access Module Processor vproc
The set of software services that controls the file system and data management components of a Teradata Database.

**ANSI** American National Standards Institute (http://www.ansi.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://webstore.ansi.org/ansidocstore/find.asp?

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 (of a relation)" on page 523.

**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 standards: ISO-14962-1997 and ANSI-X3.4-1986(R1997).

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” on page 523.

**AWP** AMP Worker Task

**Black tree** Synonym for “SynTree” on page 528.

**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” on page 522.

**BNF** Backus-Naur Form or Backus Normal Form

A metalanguage used to specify computer languages.

**BTEQ** Basic Teradata Query facility.
A Teradata request and script processing facility based on the CLIv2 API.

**BYNET** Banyan Network - High speed interconnect

**cardinality** The 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.

**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 the Teradata Database and receiving their results.

**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” on page 522.

**Cover** A condition in which all the column data requested by a query can be obtained by index-only access.

**cs0, cs1, cs2, cs3** The four code sets (codeset 0, 1, 2, and 3) used in EUC encoding.

cs0 always contains an ISO-646 character set.

All of the other sets must have the most-significant bit set to 1, and they can use any number of bytes to encode the characters. In addition, all characters within a set must have:

- The same number of bytes to encode all characters
- The same column display width (number of columns on a fixed-width terminal)

Each character in cs2 is preceded by the control character ss2 (single-shift 2, 0x8E). Code sets that conform to EUC do not use the ss2 control character other than to identify the third set.

Each character in cs3 is preceded by the control character ss3 (single-shift 3, 0x8F). Code sets that conform to EUC do not use the ss3 control character other than to identify the fourth set.

The EUC for Japanese consists of single-byte and multibyte characters (2 and 3 bytes). The encoding conforms to ISO-2022 and is based on JIS and EUC definitions as follows:

| Code Set | Encoding | Character Set |
|---|---|---|
| cs0 | 0xxxxxxx | ASCII |
| cs1 | 1xxxxxxx 1xxxxxxx | JIS X0208-1990 |
| cs2 | 0x8E 1xxxxxxx | JIS X0201-1976 |
| cs3 | 0x8F 1xxxxxxx 1xxxxxxx | JIS X0212-1990 |

**degree (of a relation)** The number of columns in a relation. If a relation has a degree of 5, then it has 5 columns.

Degree is a synonym for "arity" on page 521.

**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" on page 521.

**EUC** Extended UNIX Code

A character encoding scheme specified by ISO standard ISO-2022.

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.

**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.

**FK** 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 "PK" on page 527.

**FTS** 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.

**Global Index** A join index (see "JI" on page 525) defined with the ROWID keyword to reference the corresponding base table rows.

**HI** Hash Index

A vertical partition of a base table having properties similar to a single-table "JI."

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.

**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).

**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

**JI** 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.

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.

**JIS** Japanese Industrial Standards

A subset of the standards defined, certified, and administered by the Japanese Standards Organization (see http://www.jsa.or.jp/default_english.asp or http://www.jsa.or.jp). The relevant standards for computer language processing are represented by Division X, “Information Processing,” of the JIS standards (see http://www.webstore.jsa.or.jp/webstore/JIS/FlowControl.jsp?lang=en&bumon=X&viewid=JIS/html/en/CartegoryListEn.htm).

**LOB** Large Object

Any data object that is larger than the maximum row size for the Teradata Database. There are two types of LOB: the “BLOB” and the “CLOB.”

**LT/ST** Large Table/Small Table (join)

An optimized join type used to join fact (large) tables with their satellite (small) dimension tables.

**NPPI** Nonpartitioned Primary Index

A "PI" that is not partitioned into range buckets by means of a partitioning expression.

**NUPI** Non-Unique Primary Index

A "PI" 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.

**NUSI** Non-Unique Secondary Index

An AMP-local "SI" designed to be used for set (multirow) selection rather than single row selection.

**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.

**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.

**OLTP** On-Line Transaction Processing

**Operation tree** The parse tree created by the Optimizer by transforming the ResTree for an SQL statement. Synonym for "White tree".

**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.

**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** 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.

Each table in a Teradata database must have one, and only one, primary index, which might or might not be unique.

Compare with "PK."

**Pink tree** A partially transformed "Red tree" that has not become a completed "White tree."

**PK** 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 designed.

A table can have multiple candidate keys, but only one primary key can be defined for it. Those candidate keys that are not used 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.

**PPI** Partitioned Primary Index

A "PI" that is used to first distribute rows to the AMPs as they would be by an "NPPI," then partitioned into a set of ranges determined by the DBA and specified using a PARTITION BY clause in the table definition statement.

PPIs are very useful for various types of range queries.

**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.

**QCD** Query Capture Database

**QCF** Query Capture Facility

**RDBMS** Relational Database Management System

A database management system based on relational set theory and the theorems, axioms, and operators provided by set theory. The set theoretic foundation for an RDBMS provides a scientific, predictable, set of tools for managing data.

**Red tree** The version of the parse tree produced by the Resolver. Synonym for "ResTree."

**ResTree** The version of the parse tree produced by the Resolver. Synonym for "Red tree."

**RI** Referential Integrity

A method of ensuring that no data is ever orphaned in a relational database. Referential integrity uses the parent-child relationships between a "PK" and an "FK" to prevent child table rows from ever being orphaned from deleted parent table rows.

Referential integrity relationships are often used by the Optimizer to enhance query performance.

**SI** 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 also 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.

Secondary indexes come in two types: "USI" and "NUSI."

**SIAM** Society for Industrial and Applied Mathematics (http://www.siam.org).

The leading US-based professional society for applied mathematics professionals.

**SQL Assistant** A Teradata request processing facility based on the ODBC API.

**SynTree** The skeletal parse tree for an SQL request created by the Syntaxer. See "Black tree" on page 522.

**TDQM** Teradata Query Manager

**TLE** Target Level Emulation

**TPA** Trusted Parallel Application

A TPA is an application that Teradata has certified to run the Teradata Database safely. The Teradata Database software itself is a TPA.

**TSET** Teradata System Emulation Tool

**UDT** User-Defined Type

A data type defined by someone other than Teradata. UDTs come in two variations: Distinct and Structured. See *SQL Reference: Data Definition Statements* and *SQL Reference: UDF, UDM, and External Stored Procedure Programming* for further information about UDTs.

**UCS-2** Universal Coded Character Set containing 2 bytes

**UPI** Unique Primary Index

A "PI" 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."

**USI** Unique Secondary Index

An "SI" designed to facilitate single-row access.

**vproc** Virtual Process

The Version 1 Teradata architecture used several different specialized node types to process data, including the following node types:

- 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.

**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 statement. Synonym for "Operation tree" on page 526. A textual version of the white tree is returned to a requestor who submits an EXPLAIN for a request.

# Index

## A



## B



## C

## D



## E



## F



## G

## H



## I



## J

## L



## M



## N



## O

## P



## Q

## R



## S

## T

## U



## V



## W