

Teradata Vantage™ - SQL Request and Transaction Processing

Release 17.10

July 2021

Copyright and Trademarks

Copyright © 2000 - 2021 by Teradata. All Rights Reserved.

All copyrights and trademarks used in Teradata documentation are the property of their respective owners. For more information, see [Trademark Information](#).

Product Safety

Safety type	Description
	Indicates a situation which, if not avoided, could result in damage to property, such as to equipment or data, but not related to personal injury.
	Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury.
	Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury.

Third-Party Materials

Non-Teradata (i.e., third-party) sites, documents or communications ("Third-party Materials") may be accessed or accessible (e.g., linked or posted) in or in connection with a Teradata site, document or communication. Such Third-party Materials are provided for your convenience only and do not imply any endorsement of any third party by Teradata or any endorsement of Teradata by such third party. Teradata is not responsible for the accuracy of any content contained within such Third-party Materials, which are provided on an "AS IS" basis by Teradata. Such third party is solely and directly responsible for its sites, documents and communications and any harm they may cause you or others.

Warranty Disclaimer

Except as may be provided in a separate written agreement with Teradata or required by applicable law, the information available from the Teradata Documentation website or contained in Teradata information products is provided on an "as-is" basis, without warranty of any kind, either express or implied, including the implied warranties of merchantability, fitness for a particular purpose, or noninfringement.

The information available from the Teradata Documentation website or contained in Teradata information products may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

The information available from the Teradata Documentation website or contained in Teradata information products may be changed or updated by Teradata at any time without notice. Teradata may also make changes in the products or services described in this information at any time without notice.

Machine-Assisted Translation

Certain materials on this website have been translated using machine-assisted translation software/tools. Machine-assisted translations of any materials into languages other than English are intended solely as a convenience to the non-English-reading users and are not legally binding. Anybody relying on such information does so at his or her own risk. No automated translation is perfect nor is it intended to replace human translators. Teradata does not make any promises, assurances, or guarantees as to the accuracy of the machine-assisted translations provided. Teradata accepts no responsibility and shall not be liable for any damage or issues that may result from using such translations. Users are reminded to use the English contents.

Feedback

To maintain the quality of our products and services, e-mail your comments on the accuracy, clarity, organization, and value of this document to: docs@teradata.com.

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed nonconfidential. Without any payment or other obligation of any kind and without any restriction of any kind, Teradata and its affiliates are hereby free to (1) reproduce, distribute, provide access to, publish, transmit, publicly display, publicly perform, and create derivative works of, the Feedback, (2) use any ideas, concepts, know-how, and techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, and marketing products and services incorporating the Feedback, and (3) authorize others to do any or all of the above.

Contents

Chapter 1: Introduction to SQL Request and Transaction Processing	8
Changes and Additions	8
Chapter 2: Request Parsing	9
SQL Parser	9
Dictionary Cache	12
Statistics Cache	13
Object Use Count Cache	13
Request Cache	14
Parameterized Requests	17
Dynamically Parameterized Requests	26
Syntaxer	27
Resolver	27
Security Checking	28
Query Rewrites	28
Optimizer	29
Generator	34
OptApply	35
Dispatcher	36
Chapter 3: Query Rewrite, Statistics, and Optimization	38
Query Rewrite	38
Examples of Query Rewrites	38
Query Optimizers	64
Teradata Optimizer Processes	67
Incremental Planning and Execution	69
Translation to Internal Representation	71
Optimizer Statistics and Demographics	77
How the AMP Software Collects Statistics	84
Interval Histograms	91
Sampled Statistics	101
Dynamic AMP Sampling	104
Comparing the Accuracies of Methods of Collecting Statistics	112
Optimal Times to Collect or Recollect Statistics	114
Optimizer Use of Statistical Profiles	116
Using Interval Histograms to Make Initial Cardinality Estimates	119
Determining the Reliability of Statistics From History Intervals	129
Derived Statistics	131
Statistical Inheritance by Hash and Join Indexes	135

Deriving Column Demographics	135
Using Join Index Statistics to Estimate Single-Table Expression Cardinalities	153
Using a Unique Join Index in the Access Path for a Query	159
Estimating Join Cardinality With Single-Row Unique Index Access to One of the Tables	164
Stale Statistics	165
Object Use and UDI Counts	168
Optimizing the Recollection of Statistics	171
Using Extrapolation to Replace Stale Statistics	175
Cost-Based Optimization	177
Environmental Cost Factors	185
Row Partitioning	186
Row Partition Elimination	187
Static Row Partition Elimination	195
Examples of Rewrites Using Row Partitioning	202
Delayed Row Partition Elimination	208
Dynamic Row Partition Elimination	211
Single Partition Scans and BEGIN/END Bound Functions	212
Column Partitioning	214
Column Partition Elimination	215
Access Planning for Column-Partitioned Objects	215
Chapter 4: Join Planning and Optimization	217
Optimizer Join Plans	217
Join Geography	225
Determining the Order of Joins	231
Partial GROUP BY Block Optimization	236
Join Strategies and Methods	243
Product Join	247
Merge Join	252
Direct Row-partitioned PI Merge Join	261
Rowkey-Based Merge Join	263
Single-Window Merge Join	271
Sliding-Window Merge Join	273
Hash Join	278
Nested Join	289
Local Nested Join	289
Slow Path Local Nested Join	290
Fast Path Local Nested Join	293
Remote Nested Join	293
Nested Join Examples	295
Join Plan Without Nested Join	296
Join Plan With Nested Join	297
Exclusion Join	297
Exclusion Merge Join	298
Exclusion Product Join	301

Inclusion Join	303
Inclusion and Exclusion Product Joins With Dynamic Row Partition Elimination	303
RowID Join	311
Correlated Joins	314
Self-Join	315
Chapter 5: Join Optimizations	317
Large Table/Small Table Joins	317
Star and Snowflake Join Optimization	318
LT/ST-J1 Indexed Joins	323
LT-ST-J2 Unindexed Joins	323
Miscellaneous Considerations for Star Join Optimization	324
Selecting Indexes for Star Joins	325
Star Join Examples	327
Chapter 6: Optimization Using Join Indexes	339
Join Indexes	339
Maintaining a Join Index for DELETE, INSERT, and UPDATE Operations	346
General Method of Maintaining a Join Index During Simple DELETE Operations	347
General Methods of Maintaining a Join Index During Joined DELETE Operations	348
Optimized Method of Maintaining a Join Index During DELETE Operations	351
General Method of Maintaining a Join Index During INSERT Operations	352
General Method of Maintaining a Join Index During UPDATE Operations	353
Optimized Method of Maintaining a Join Index During UPDATE Operations	355
Chapter 7: Interpreting EXPLAIN Output	357
EXPLAIN Request Modifier	357
EXPLAIN Confidence Levels	358
EXPLAIN Request Modifier Phrase Terminology	363
EXPLAIN Request Modifier: Examples	386
EXPLAIN Request Modifier and Standard Indexed Access	390
EXPLAIN Request Modifier and Join Processing	394
EXPLAIN Request Modifier and Parallel Steps	400
EXPLAIN Request Modifier and Partitioned Primary Index Access	402
EXPLAIN Request Modifier and Column-Partition Access	409
EXPLAIN Request Modifier and MERGE Conditional Steps	422
EXPLAIN and UPDATE (Upsert Form) Conditional Steps	428
EXPLAIN Request Modifier and Triggers	441
EXPLAIN Request Modifier and Recursion	442
STATIC EXPLAIN	444
DYNAMIC EXPLAIN	449
Chapter 8: Transaction Processing	458
Database Transactions	458
Transactions, Requests, and Statements	462

Transaction Semantics Differences in ANSI and Teradata Session Modes	464
Comparison of Transactions in ANSI and Teradata Session Modes	467
ANSI Session Mode Transaction Processing Case Studies	470
Teradata Session Mode Transaction Processing Case Studies	474
Rollback Processing	478
Database Locks, Two-Phase Locking, and Serializability	479
Load Isolation	486
Lock Manager	492
Database Locking Levels and Severities	496
Client Utility Locks	502
Default Lock Assignments and Lock Upgradeability	503
Blocked Requests	507
Proxy Locks	509
Pseudo Table Locks	512
Deadlock	517
Minimizing Deadlock	519
Example of a Transaction Without Deadlock	523
Example of a Transaction With Deadlock	524
Example of Two Serial Transactions	526
DDL and DCL Requests, Dictionary Access, and Locks	527
DML Requests and Locks	528
Locking Issues With Consume Mode SELECT Queries on a Queue Table	530
Cursor Locking Modes	533
Locking Issues With Tactical Queries	534
Chapter 9: Query Capture Facility	544
Quick Functional Overview of the Query Capture Facility	544
QCD Table Definitions	545
AnalysisLog	545
AnalysisStmts	546
DataDemographics	548
Field	549
Index_Field	551
IndexColumns	552
IndexMaintenance	553
IndexRecommendations	555
IndexTable	559
JoinIndexColumns	561
Partition Recommendations	563
Predicate	566
Predicate_Field	567
QryRelX	568
Query	569
QueryBandTbl	573
QuerySteps	574

RangePartExpr	580
Relation	582
SeqNumber	587
SingleRowRelation	589
StatsRecs	589
TableStatistics	591
User_Database	593
UserRemarks	594
ViewTable	595
Workload	596
WorkloadQueries	597
WorkloadStatus	598
XMLQCD	599
 Appendix A: XML Documents Produced by the Query Logging XMLPLAN Option	 601
 Appendix B: Additional Information	 610

Introduction to SQL Request and Transaction Processing

Teradata Vantage™ is our flagship analytic platform offering, which evolved from our industry-leading Teradata® Database. Until references in content are updated to reflect this change, the term Teradata Database is synonymous with Teradata Vantage.

The primary purpose of *Teradata Vantage™ - SQL Request and Transaction Processing* is to facilitate better understanding of the output reported by the EXPLAIN request modifier.

Teradata Vantage™ - SQL Request and Transaction Processing describes the Teradata SQL parser, including its component parts, the query capture database, the database components of related utilities, and the basics of the Teradata transaction processing environment.

This preface describes the organization of *Teradata Vantage™ - SQL Request and Transaction Processing* and identifies information you should know before using it. This documentation should be used in conjunction with that other SQL information.

Changes and Additions

Date	Description
July 2021	Minor edits.

Request Parsing

This section describes SQL request parsing, including the components of the SQL Parser that deal with request processing.

The information provided is designed to help you to interpret EXPLAIN reports more accurately.

SQL Parser

The SQL Parser

The SQL Parser is a component of the Parsing Engine (PE).

SQL requests are sent to a PE's Parser in CLlv2 request parcels. Request parcels consist of the following elements:

- Zero or more SQL statements
- Control information
- Optional USING request modifier data (data parcels)

The Parsing Engine has two main components related to query processing: the Parser and the Dispatcher.

The Parser includes the following components:

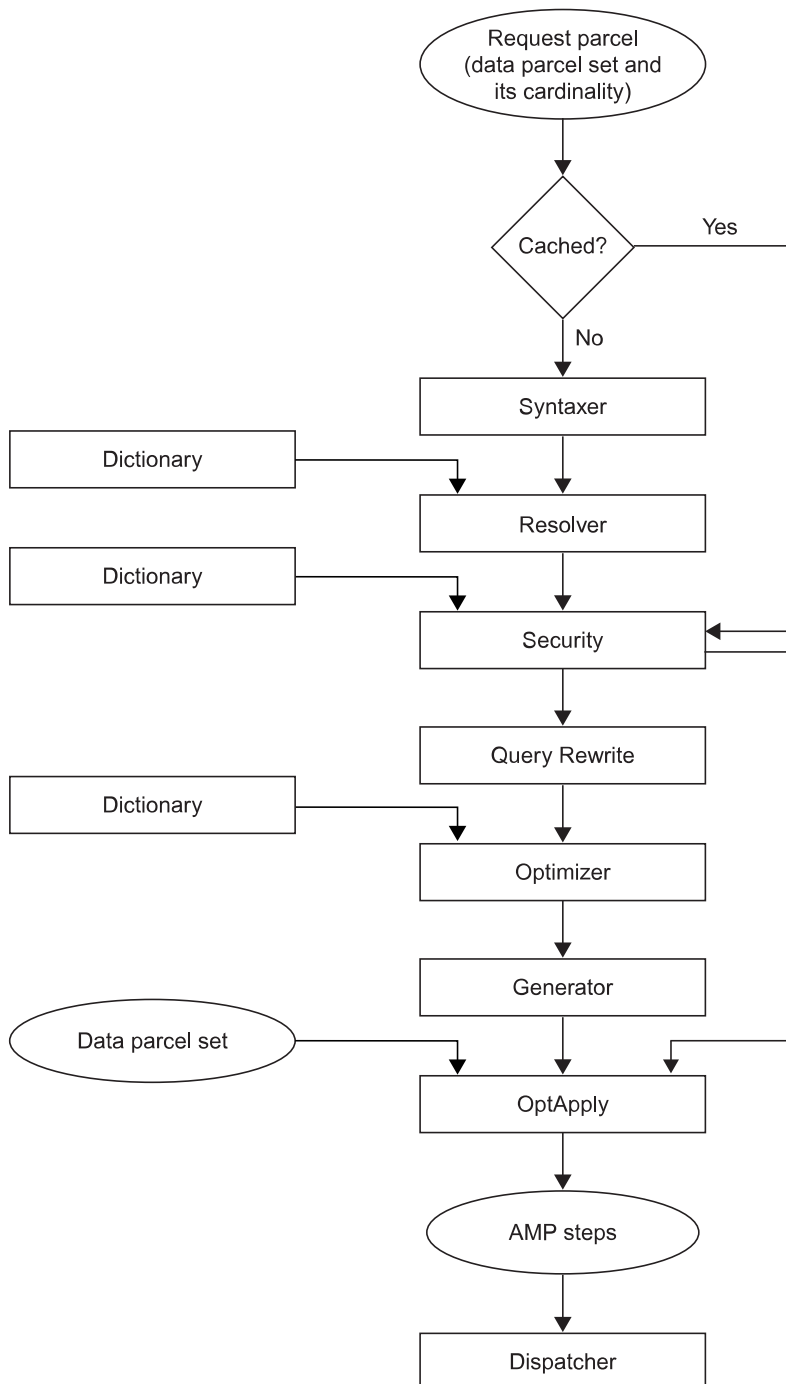
- Dictionary cache
- Statistics cache
- Object use count cache
- Request cache
- Syntaxer
- Dynamic parameterization of literals (DPL)
- Resolver
- Discretionary access control security checking
- Parameterized value peek
- Query rewrite
- Optimizer
- Generator
 - Plastic steps generation
 - Steps packaging
- OptApply
 - Concrete steps generation

The Dispatcher includes the following components:

- Execution control

- Response control
- Transaction and request abort management
- Queue table cache management

The parsing engine generates AMP steps from request parcels, as shown in the following simplified illustration.



Parsing Engine Component Processes

The following is an overview of the major parsing engine components.

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](#)).

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 discretionary access control 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 Parameterized Value Peek.

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

3. The Query Rewrite subsystem takes the *ResTree* and performs various rewrites such as:

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

The Query Rewrite subsystem then passes the revised, semantically equivalent, *ResTree'* to the Optimizer. See [Query Rewrite, Statistics, and Optimization](#) for further information.

4. The Optimizer analyzes the *ResTree'* using various statistical and configuration data about the database and the system hardware components to determine the optimum plans for the request. This subsystem includes:

- Access planning
- Join planning
- Join index planning
- Complex outer join planning
- Subquery planning
- Aggregation planning
- Insert, delete, and update planning
- Optimizer query rewrites

The Optimizer first checks the statistics cache to see if the statistics it needs have already been retrieved from the Data Dictionary. If not, the Optimizer retrieves them from the Data Dictionary.

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

The Optimized Parse Tree, now transformed from a simple statement tree to a complete 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. 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.

When you perform an EXPLAIN of a request, the report the system produces is a verbal description of the White Tree the Optimizer produces for the request plus some additional information about uncosted steps that Vantage inserts into the White Tree for use by the Teradata dynamic workload management software.

See [Query Rewrite, Statistics, and Optimization](#) and the documentation for the EXPLAIN modifier in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for further information.

5. The Steps Generator creates Plastic Steps from the White Tree. Plastic Steps are, except for statement literals, a data-free skeletal tree of AMP directives derived from the Optimized Parse Tree.

The completed Plastic Steps tree is then passed to the Request Cache and to Steps Packaging for further processing.

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

If any data parcels were passed to the Parser via a parameterized request, then that data is also added to the steps tree (in this discussion, 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.).

The final product of this process is referred to as Concrete Steps.

7. Steps Packaging passes the Concrete Steps to the Dispatcher for assignment to the AMPs.
8. The Dispatcher sequentially, incrementally, and atomically transmits the Concrete Steps, called AMP Steps or Spoil Steps at this point in the process, across the BYNET to the appropriate AMPs for processing.
9. The Dispatcher manages any abort processing that might be required.
10. The Dispatcher receives the results of the AMP Steps from the BYNET and returns them to the requesting application.

Dictionary Cache

The Dictionary Cache

The dictionary cache is a buffer that stores the non-demographic Data Dictionary information most recently used to process SQL queries. The database caches the most recently used statistics and demographic data in the statistics cache (see [Statistics Cache](#)).

Vantage uses information from the Data Dictionary about privileges, tables, columns, views, macros, triggers, stored procedures, and other objects to transform an SQL request into the AMP steps needed to process a request. For information about AMP steps, see [OptApply](#).

Caching Data Dictionary information reduces the I/O operations required to perform the following actions:

- Resolve database object names
- Validate object and row-level security privileges

If an SQL request modifies the contents of the Data Dictionary, the database sends a spoil message to every PE on the system, directing them to drop the modified definitions from their respective dictionary caches.

The database purges the dictionary cache periodically, one PE at a time.

For the current default and maximum sizes of the dictionary cache, see the documentation for the DBS Control field DictionaryCacheSize in *Teradata Vantage™ - Database Utilities*, B035-1102.

Statistics Cache

The Statistics Cache

The statistics cache minimizes Parser overhead required to optimize queries. It does this by caching statistics, UDI counts, and other demographic information from the Data Dictionary that has been fetched during a session to optimize SQL queries.

When the Optimizer requires statistics to process a table, it first retrieves the summary information for all the statistics on the table. Vantage stores the retrieved demographic data in the statistics cache, where it is expected to remain cached for an extended period. The purpose of the statistics cache is to retain statistics that have been fetched to optimize queries in memory for as long as possible.

You can use the DBS Control field NumStatisticsCacheSegs to change the size of the statistics cache. See *Teradata Vantage™ - Database Utilities*, B035-1102 for further information about fine tuning the statistics cache.

Unlike the dictionary cache, Vantage does not purge the statistics cache periodically, purging it only when operations to collect or drop statistics occur.

Object Use Count Cache

About the Object Use Count Cache

The object use count cache, or OUC cache, tracks object access use counts, UDI (UPDATE, DELETE, INSERT) counts, and statistics access counts for DML requests made against the database.

The system uses the object use count cache information for various purposes, including:

- Query optimization
- Performance monitoring
- Database object optimization
- Statistics management

Because only table-level insert and delete UDI counts are needed for cardinality estimation, Vantage records that information in the OUC cache. Column-level update counts are only needed when the Optimizer

is deciding whether a COLLECT STATISTICS request can be skipped because it fails to meet one or more USING thresholds, so column-level update counts are only retrieved when necessary.

When Vantage flushes the OUC cache, it writes its data to the dictionary table `DBC.ObjectUsage`. See *Teradata Vantage™ - Data Dictionary*, B035-1092 for the definition of the `ObjectUsage` table.

The following is populated in the object use count cache with both object use count and UDI data:

- Table
- View
- Macro
- Trigger
- Join index
- Hash index
- Statistics

Runtime data on the following step-level operations:

- Delete
- Insert
- Update

Flushing the OUC Cache to Disk

The OUC cache must be flushed periodically to avoid the accumulation of stale use counts in `DBC.ObjectUsage`. The following criteria trigger a flush of the OUC cache to disk:

- When a specific user-defined time interval is reached.

You can specify a time-based flushing interval using the DBS Control field `ObjectUseCountCollectRate`. Vantage flushes the OUC cache when either of the following events first occurs:

- The specified elapsed time since the last flush of the cache is reached.
- The cache is full.

See *Utilities* for more information about the `ObjectUseCountCollectRate` field.

- When a high amount of activity occurs on a set of database objects.

Flushing the cache based on high activity is necessary to avoid accumulating stale counts in `DBC.ObjectUsage`. Fresh use counts are important for determining when statistics need to be recollected.

High activity cache flushing is particularly useful after bulk load jobs where table cardinalities can change radically in a short period of time. For example, the OUC cache is flushed immediately after `FastLoad` and `MultiLoad` jobs complete.

- When the Optimizer makes a specific request to flush the cache.

This is referred to as on-demand flushing.

Request Cache

The request cache stores certain categories of successfully parsed SQL requests and their plastic steps so they can be reused, eliminating the need to parse the same SQL request more than once. The request cache is a PE-local in-memory buffer that stores the steps generated during the parsing of a DML request.

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

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

When the request cache is initialized during Vantage startup, the number of cache entries is set to MaxRequestsSaved, a DBS Control field in the Performance group.

Immediate Caching and Non-Immediate Caching

Not all requests are cached, and not all cached requests are cached immediately.

When a request has a data parcel (specified in Teradata SQL by a USING request modifier), the system may cache it immediately using a generic plan, or it may decide to use a specific plan. In this discussion, 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.

By exposing parameterized values before determining a plan for a request, it becomes possible to generate a more optimal plan for some categories of specific requests than could be generated without first peeking at those values. The term *parameterized* is used instead of USING because it is possible to send both data and DataInfo parcels with a request without specifying a USING request modifier and to specify a USING request modifier with just a data parcel and no DataInfo parcel. See *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417 and *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418 for information about Data and DataInfo parcels.

The result is identical to what would be achieved had you, for example, specified constants instead of USING variables in all points in the request where the Optimizer considers value predicates. See [Parameterized Requests](#) for details about how this is done.

The following examples show SQL 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 Mainframe-Attached Systems*,

B035-2417 or *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418 for more information about iterated requests).

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

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

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

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

If the same request is submitted to the Parser a second time, the request can be considered for dynamic parameterization of literals (DPL). The system may move its entry from the first-seen area into one of the cache entries with the plastic steps. In some cases, a plan is not cached until the request is seen a third time. For more information about DPL, see [Dynamically Parameterized Requests](#).

Purging the Request Cache

About Purging the Request Cache

Cached plans remain in the request cache until the system spoils them, which means they are no longer valid. For example, DDL changes alter the database schema and can make previously valid plans nonvalid.

Only those cached requests that reference the changed database object are purged.

Cached plans are also purged if they are specific to a resolved date value that no longer matches the date returned by the DATE, CURRENT_DATE, or TEMPORAL_DATE built-in functions. In some cases, the TEMPORAL_TIMESTAMP built-in function value is also purged.

Purging Exempt and Non-Exempt Requests

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

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

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

Purging a Full Request Cache

The maximum number of entries possible in the request cache depends on the setting for the DBS Control MaxRequestsSaved field in the Performance group. For more information about the MaxRequestsSaved field, see *Teradata Vantage™ - Database Utilities*, B035-1102.

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 and Individual Request Cache Entries

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

The system purges request cache entries individually under the following conditions:

- The cache becomes full, and space is needed for a new entry.
- A data definition request (for example, an ALTER TABLE request) is submitted for a table that has been specified by a cached request.

Parameterized Requests

Vantage has the ability to peek at the parameterized values of a request (such as USING request modifier values, which are user-supplied constant data) in CLlv2 data parcels, and then to resolve those variables in the Resolver. This enables the Optimizer to generate a specific, uncached plan for such requests when the system determines this is the best way to handle the request rather than always generating a cached generic plan.

Data Parcel Peeking Terminology

The following terminology is specific to the parameter value peeking subsystem:

Term	Definition
Specific Plan	<p>An Optimizer plan that applies only to a single instance of that query and is generated by using one or both of the following:</p> <ul style="list-style-type: none"> • peeking at parameterized values and CURRENT_TIMESTAMP and USER built-in functions • using intermediate incremental planning and execution spool results <p>Single instance plans are called specific query plans. Vantage caches specific query plans. The values obtained from peeking at a parameterized request can then be used to tailor a very specific query plan for that request. Specific plans can be either static or dynamic plans.</p>
Generic Plan	<p>An Optimizer plan generated without peeking at parameterized values and CURRENT_TIMESTAMP and USER built-in functions, intermediate incremental planning and execution spool results, or both, that apply to most, if not all, instances of that query. Generic plans are static plans.</p>

Term	Definition
Specific Always	Parameterized values are resolved and evaluated in the plan for all subsequent requests.
Generic Always	Parameterized values are not resolved until the concrete steps for the request are generated, and the cached generic plan is used for all subsequent requests.

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

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

```

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

```

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

Date-based parameter value peeking, which resolves the current date from `DATE` and `CURRENT_DATE` built-in functions when one of those functions is specified in the request, can be used to generate either a specific or generic plan, depending on several other factors that might be present in the text. The terminology used for these date-specific query plans is as follows:

- A generic parameterized request with a resolved `DATE` or `CURRENT_DATE` value for which a query plan is generated is called a DateSpecific generic plan.
- A specific parameterized request with a resolved `DATE` or `CURRENT_DATE` value for which a query plan is generated is called a DateSpecific specific plan.

In the context of the parameter value peek subsystem, the following definitions apply:

Term	Definition
Parameterized PK statement	<p>A single SQL statement with a set of parameterized variables and/or <code>CURRENT_TIMESTAMP</code>/<code>USER</code> built-in functions that are used in equality predicate conditions on a nonpartitioned table, a row-partitioned PI table, or USI of a single table with no ORed conditions.</p> <p>The term <i>PK</i>, which commonly represents Primary Key, in this case represents a primary or unique secondary index column set.</p>

Term	Definition
Parameterized PK request	<p>A request that contains only the following:</p> <ul style="list-style-type: none"> Parameterized PK statements Null statements <p>See <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146 for a definition of null statements.</p>

Parameterized value peeking does not impact the treatment of parameterized PK requests, so the Optimizer does not generate a specific plan for those cases.

The following table indicates whether various types of parameterized request are parameterized-dependent or parameterized-independent.

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

Parameterized Value Peeking

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

- Satisfiability and transitive closure (see [Predicate Simplification](#))
- Optimum single table access planning
- Row partition elimination (see [Row Partition Elimination](#))
- Using covering secondary, hash, and join indexes in place of base tables (see [Query Rewrite, Statistics, and Optimization](#))

Peeking facilitates the optimization of certain categories of queries by inserting data values that are specific to those queries into the parse tree at an earlier stage of request processing than would otherwise be done. The system always caches generic plans because reusing a cached query plan saves parsing and optimization time. However, reusing a cached generic query plan is not always the best approach to executing a query that provides constant literals that can be used to produce a specific plan tailored closely to that specific request.

The system does not cache specific plans for requests that are parameterized-dependent, because those plans cannot be reused for otherwise identical queries that have different sets of parameterized or built-in function-supplied constant values, but it does cache the other information generated for specific plans. This information includes its SQL text hash, its host character set, its estimated execution costs, its parsing time, and its run time.

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

A specific plan generated by the Optimizer should be an optimal plan, and its runtime performance should be significantly better than the execution time for an equivalent generic plan. However, there can be cases when the runtime costs of a specific plan and its equivalent generic plan do not differ significantly. In this case, the impact is especially high if the parsing cost is high. To avoid these cases, the system monitors all such requests for their parsing and run times, and keeps the information in the request cache. The system then uses this information to decide between generating a specific or generic plan for a request. For more details about this process, see [Request Caching Logic](#).

Enabling and Disabling Parameterized Value Peeking

By default, parameterized value peeking is enabled. To disable it, you can change the value of the DisablePeekUsing field using the DBS Control utility. For more information about DBS Control, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Miscellaneous Considerations for Parameterized Value Peeking

The items in the following list also affect parameterized value peeking:

- If a USING request modifier is being explained, or submitted with an INSERT EXPLAIN or DUMP EXPLAIN request, the system peeks at the parameterized variables if a data parcel is also provided in the request.

If no data parcel is submitted, then the system does not peek at the parameterized variables, and the system generates a generic plan for the request.

- If a parameterized request is submitted under a MultiLoad or FastLoad sessions, the system does not peek at the parameterized variables and current date, so there is no impact on request caching.
- If a parameterized request is submitted under a FastExport session, the system peeks at the parameterized variables and current date for a SELECT request in order to generate a more optimal plan for exporting the data.

Note that requests submitted in a FastExport session are not cached, so peeking at parameterized request variable values does not impact caching behavior.

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

The following performance benefits are realized from parameterized value peeking:

- Vantage generates specific plans for all requests if either of the following conditions are true:
 - The system compares the plan estimates and the estimated cost of the specific plan is less than the estimated cost of the generic plan.
 - Both the parsing and runtime CPU costs of the specific plan is smaller than the runtime CPU cost of the equivalent generic plan.
- The values for the DATE, CURRENT_DATE, USER, or TEMPORAL_DATE (in some cases, the TEMPORAL_TIMESTAMP function is also expanded and passed to the Optimizer) built-in functions are expanded and passed to the Optimizer for optimal plan generation.

This is done even if the request does not specify parameterized data.

- If a specific plan does not provide a performance benefit, the system instead caches the equivalent generic plan and reuses it for subsequent requests.

The following potential performance liabilities may occur with parameterized value peeking:

- In some cases, estimates of generic and specific plans are compared. If the generic plan is worse than the equivalent specific plan, the database parses the request again and generates a specific plan for it.
- A request that is classified as parameterized-dependent might not always generate a different specific plan. Ideally, they would be cached immediately; however, they are instead cached only after their second submission to the system. This can have a performance impact for those requests that are submitted only twice.
- If the specific plan and the generic plan are identical, and there are no potential benefits seen for the specific plan, then the generic plan is used for all subsequent requests. As a result, no performance

benefit can be realized even if parameterized data values in subsequent requests are such that they would result in more optimal specific plans if they were exposed and inserted into the plan as literal data values.

However, once the request cache is purged, the system reevaluates all affected specific and generic plans.

- In some cases, the decision to cache a generic plan is based on the elapsed execution times of the specific and generic plans. The decision can be influenced wrongly because of other workloads running on the system that might block either of the requests being compared, and as a result skew their elapsed time values.
- In some cases, the algorithm that compares specific and generic costs uses estimates. A bad estimate for either plan can influence the caching decision.

Request Caching Logic

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

If parameterized value peeking is not enabled, when a parameterized request is seen and successfully parsed, it is cached immediately. Parameterized value peeking changes the behavior of the request cache for requests submitted in CLIV2 Execute or Both Prepare and Execute modes, as follows:

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

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

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

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

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

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

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

If parameterized value peeking is enabled, the following is the relevant portion of the EXPLAIN text output for the same request. Notice the EXPLAIN output indicates 9 fewer row partitions to scan due to parameterized value peeking (the relevant text is highlighted in boldface type).

```
...
3) We do an all-AMPs RETRIEVE step from 14 partitions of MYDB.t4
   with a condition of ("(MYDB.t4.d <= DATE '2006-03-02') AND
   (MYDB.t4.d >= DATE '2005-02-01')") into Spool 1 (group_amps),
   which is built locally on the AMPs. The size of Spool 1 is
   estimated with no confidence to be 2 rows. The estimated time
   for this step is 0.08 seconds.
...
```

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

The system first recomputes the date.

IF the recomputed date ...	THEN the ...
matches the cached date	cached plan is reused.
does not match the cached date	cached plan is purged and a new plan is generated for the changed date. The new plan is then cached.

- When parameterized value peeking is enabled, requests are cached or not according to the actions described by the following table.

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

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

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

Otherwise, the database parses the request again and generates a specific plan for it.

This means that a generic plan, if executed, is always cached for subsequent reuse if either of the following statements is true:

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

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

A performance benefit is expected to be seen when the system decides to generate specific plans for all subsequent submittals of the request. A performance benefit is expected to be seen when the system decides to use the cached plan for all subsequent submittals of the request.

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

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

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

For example, consider the following table definition.

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

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

Date	Time	Unique Values	Column Names
-----	-----	-----	-----


```
06/02/17 12:51:22          1,537 j
06/02/17 12:56:10          60 k
```

Now consider the following query against table t2, showing the relevant step:

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

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

...
```

Also consider the following query, which is also made against table t2, showing the relevant step:

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

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

...
```

Now create table t3 as a copy of table t2 with data and consider the following join index definition:

```
CREATE JOIN INDEX j1 AS
  SELECT *
  FROM t3
  WHERE j > 58
  AND k > i+3;
```

Consider the following query and assume the USING value for a1 is 4:

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

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

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

Dynamically Parameterized Requests

Dynamic Parameterization of Literals (DPL) reduces parsing time for repeated, nonparameterized requests that differ only in the eligible literals used in the predicates of WHERE and ON clauses. Eligible nonparameterized requests are treated as identical requests after parameterization of these literals. The literals are used as the data values for the parameters.

A plan with DPL applied is cached with the associated dynamically parameterized request text. Later, when a nonparameterized request is submitted with different literal values, and the dynamically parameterized form of this request matches the cached parameterized request, the saved plan of the parameterized request from the request cache is used. This avoids the need to reparse the request and regenerate the plan.

The Parser generates a plan without applying DPL for the first occurrence of an eligible nonparameterized request, then creates a plan with DPL applied if another request (that matches the first request after applying DPL) is seen for the second time. This allows the request caching logic to compare the performance of the plans and decide whether to continue using a cached plan with DPL applied, or revert to always using the plan without DPL, depending on which plan performs better. Most workloads perform better if you enable DPL.

Whether a request is parameterized, nonparameterized, or dynamically parameterized, the decision to cache or not cache a plan is controlled automatically by the request caching logic.

The DPL logic considers only single-statement requests that have a nonparameterized SELECT statement. It determines automatically whether such a request is eligible for DPL, and which literals in the request are eligible for parameterization.

DPL is disabled by default. The DBS Control general field EnableDynamicParameterization controls whether DPL is enabled. For more information on DBS Control, see *Teradata Vantage™ - Database Utilities*, B035-1102.

DBQL Logging for DPL

You can use DBQL logging to monitor the usage and impact of DPL. The CacheFlag column in DBC.DBQLLogTbl includes values related to DPL as indicated in the following table.

CacheFlag	Purpose
D	Generic plan from the request cache is used for the request execution.
P	Request is parsed to generate the plan by peeking the literal values (specific plan).
N	Request is parsed to generate the plan without peeking the literal values (generic plan).
I	Parameterized request caching logic has decided a specific plan is always better and this request is ineligible for dynamic parameterization of literals (DPL).

Syntaxer

The Syntaxer checks the request parcel for high-level syntax. If no errors are detected, the Syntaxer converts the request parcels into a skeletal parse tree referred to as the *SynTree* or *Black Tree*. When this process completes, the Parser passes the SynTree to the DPL component (see [Dynamically Parameterized Requests](#)), and then to the Resolver.

A parse tree is a data structure used by the SQL Parser to represent a request parcel in a form that is simple to annotate with various descriptive and statistical information derived from the Data Dictionary and from derived statistics.

The parse tree also permits a relatively simple transformation of the request parcel by Query Rewrite and the Optimizer into an execution plan.

Resolver

The Resolver annotates the SynTree with information about such things as data conversions, column names, discretionary access control security checks, and underlying relationships, and then produces a more fleshed out parse tree called a *ResTree* or *Red Tree*.

The following provides an overview of the activities performed by the Resolver:

1. The Resolver takes the SynTree as its input from the Syntaxer.
2. Each database or user, table, view, trigger, stored procedure, and macro is assigned a globally unique numeric ID.

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

These IDs are maintained in the Data Dictionary.

3. The Resolver refers to the Data Dictionary to verify all names and privileges and to convert those names to their equivalent numeric IDs.
4. The Resolver takes available information from the Data Dictionary cache, which is used on a least-recently-used or most-recently-used basis.

If the needed information is not cached, it is retrieved from the appropriate system tables.

5. If a request parcel contains views or macros, the Resolver retrieves the view or macro text from the Data Dictionary, resolves it, and then merges the elements of the resulting tree into the request tree.
6. The Resolver produces the ResTree as its output and passes it on to Security.

Security Checking

The database validates security for all requests. 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 security privileges. This includes validating both object-level control privileges and row-level control privileges.

Because object-level security privileges can be granted and revoked dynamically, the database validates user logon security privileges for every request processed, including those that have been cached.

When this process completes, the Parser passes the ResTree to the Parameterized Value Peek subsystem (see [Parameterized Requests](#)), and then to the Query Rewrite subsystem.

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

- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Data Dictionary*, B035-1092
- *Teradata Vantage™ - SQL Data Control Language*, B035-1149

Query Rewrites

A query rewrite is the process of rewriting a query, Q, in a new form, query Q', such that both of the following statements are true:

- Both queries produce the identical result with exceptions that the result order may differ if there is not an ORDER BY clause for the query and a rewrite may expose or hide an error in the query. For example, a divide by zero error may be avoided when the error would have occurred if a join was not eliminated. Also, a divide by zero error may occur if predicates are reordered based on commutative and associative rules of predicate evaluation.
- Q' runs faster than Q.

In many cases, multiple query rewrites can be applied to a query. Also, a query rewrite or feedback from query results can trigger opportunities for further query rewrites. Query rewrites can be applied both in the Query Rewrite subsystem prior to query optimization, and also during query optimization.

The following are some of the query rewrites that may be applied:

- Substituting underlying base tables and covering indexes for views and derived tables (a method called *view folding*)
- Substituting hash and join indexes for underlying base tables, views, or join operations
- Converting outer joins to inner joins
- Converting INNER join syntax to the equivalent comma syntax
- Eliminating unnecessary joins
- Using logical satisfiability and transitive closure either to eliminate terms or to add terms that facilitate further rewrites
- Simplifying predicates
- Pushing projections and predicate conditions into spooled views
- Eliminating set operation branches
- Pushing aggregations and joins into UNION ALL branches

A query rewrite can be rule-based, such as predicate pushdown, or cost-based, such as rewriting a query to use a join index.

Optimizer

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

Questions a Query Optimizer Asks and Their Answers

The Optimizer performs its task of determining a best plan 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.

For more detailed information about query optimization and how Vantage optimizes join requests, see the following sources:

- [Query Rewrite, Statistics, and Optimization](#)
- [Join Planning and Optimization](#)
- [Join Optimizations](#)

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

- Should this request be optimized using incremental planning and execution?
- What is the cardinality of the table?

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

- What is the degree of the table?

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

- Are there interval histogram statistics for the column and index sets required to process the query?
- If there are existing interval histogram statistics, are they fresh enough to provide reasonable cardinality estimates, or are they stale?
- If there are existing interval histogram statistics, can they cover a range query over DATE values, or do they require extrapolation?
- Does the table have a primary index, primary AMP index, or no primary index?
- Is the table partitioned?
- If the table is partitioned, are there PARTITION statistics?
- Is the requested column set indexed?
- If the column set is indexed, is the index unique or nonunique?
- Can row-partition elimination be applied?
- Can column-partition elimination be applied?
- How many distinct values are in the column set?
- How many rows in the column set or index have one or more nulls for the columns on which statistics have been collected?
- How many rows in the column set or index are null for all the columns on which statistics have been collected?
- How many rows per column set value are expected to be returned?
- Can a base table be replaced by a covering secondary, hash, or single-table join index?
- Is a join partly or completely covered by a join index or NUSI?
- Can a join or aggregation be pushed into UNION ALL branches?
- Can an aggregation be partially pushed to the relations of a join?
- 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?
- Is the table load isolated?

Ideally, many of these questions are answered largely based on statistical data that you have generated using the SQL COLLECT STATISTICS statement (see the information about COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144). When database statistics are collected regularly, you can expect the Optimizer to make the best decisions possible.

When the Optimizer needs index or column statistics, it first checks the statistics cache (see [Statistics Cache](#)). If the necessary statistics are not cached, the Optimizer retrieves them from *DBC.StatsTbl* (see *Teradata Vantage™ - Data Dictionary*, B035-1092 for information about *DBC.StatsTbl*).

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 Factors in Deciding How to Collect Statistics](#), [An Example of How Stale Statistics Can Produce a Poor Query Plan](#), and [Stale Statistics](#)). You can set various thresholds for recollecting statistics to ensure that the system does not recollect statistics when it is not necessary to do so. See the information about COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for more information about specifying threshold values for recollecting statistics.

If no statistics have been collected on indexed columns in a request, the Optimizer makes a snapshot sampling of data and uses that estimate to make a best guess about the optimum data retrieval path (see [Dynamic AMP Sampling](#)). Any derived statistics that the Optimizer develops begin with the dynamic AMP sample as a basis for deriving additional statistics. Note that the Optimizer does not use dynamic AMP samples of non-indexed columns to make cardinality estimates.

The degree that this dynamic 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 sample approximates its true global demographics.

Optimizer Processes

The following is a simplified list of the Optimizer processing stages.

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

Type of SQL Request in the Request Parcel	Optimizer Action
DDL or DCL	The Optimizer deletes the original request from the parse tree after it has been replaced with specific Data Dictionary operations. No access planning is required for DDL and DCL requests, so the Optimizer only converts the request parcel into work steps involving dictionary writes, locking information, and so on.
DML	The Optimizer produces access plans, join plans, and execution plans. The Optimizer then uses whatever statistical information it has, whether complete or sampled, to determine which access paths or plans are to be used. If there are no column or index statistics in the statistics cache, then the Optimizer uses dynamic 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, join plans, and aggregation, and the Optimizer selects the best plan based on the available derived statistics and costing data it has.

4. The Optimizer places, combines, and reorders table-level locks to reduce the likelihood of deadlocks, then removes any duplicate locks it finds.
5. Finally, the Optimizer either passes its fully-optimized parse tree, known as the White Tree, on to the Generator for further processing or, if has optimized an EXPLAIN request, produces a verbal explanation of the White Tree to which additional spool size information and costing data that would not otherwise be costed, is added both for human analysis and for Teradata Viewpoint.

Tasks That Query Optimizers Do Not Perform

Query optimizers do not do either of the following things:

- Guarantee that the access, join, and aggregation plans they generate are infallibly the best plans possible.

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

You should not assume that any query optimizer ever produces the single best query plan possible to support a given SQL request.

You should assume that the query plan selected is more optimal than the otherwise unoptimized Resolver ResTree' (also known as the Red Tree) formulation would have been.

- Rationalize poorly formed queries in such a way as to make their performance as effective as a semantically equivalent well-formed query that returns the same result.

A query optimizer always creates the most effective plan it can for the request 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, Statistics, and Optimization](#)) and the Optimizer to increase the efficiency of a given user-written query.

About the Types of Steps

AMP Steps

An AMP 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. The combined AMP steps for a request constitute the plan for that request.

There are several types of steps, the most important of which are the plastic and concrete steps. For more information about plastic and concrete steps, see [Generator](#).

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 Vantage architecture. Each parallel step has an independent execution path, running simultaneously with other steps.

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. Note that the Dispatcher has limits on how many steps may actually be actively running in parallel. Also, checks at the AMP level may block a parallel step until a previous parallel step completes.

Additionally, a step may spawn other steps to which rows are sent. For example, during redistribution. The sending steps and the spawned receiving steps run in parallel, with the message system acting as a pipeline between the sender and receiver.

Common Steps

Common steps are processing steps common to 2 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	The Lock Manager locks both tables (employee_number and call_emp).	Serial	Common
2	Vantage copies the rows from the employee_number table and redistributes them. The system copies the rows from the call_emp table and redistributes them.	Parallel	Common
3	Vantage Merge Joins the results.	Serial	Individual
4	Vantage Exclusion Merge Joins the results.	Serial	Individual
5	The Lock Manager releases the table-level locks on employee_number and call_emp.	Serial	Individual

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

SERIAL	Lock both tables	COMMON
PARALLEL	Copy and redistribute employee rows	Copy and redistribute call_emp rows
SERIAL	Merge join	INDIVIDUAL
SERIAL	Exclusion-merge join	INDIVIDUAL
SERIAL	Release locks	COMMON

Related Information

For more detailed information about query optimization, see [Query Rewrite, Statistics, and Optimization](#).

For information about how Vantage optimizes join requests, see [Join Planning and Optimization](#) and [Join Optimizations](#).

For information about the Viewpoint Stats Manager, which is also a component of the Optimizer, see *Teradata® Viewpoint User Guide*, B035-2206.

Generator

The Generator formulates the AMP processing steps based on the optimized parse tree plan it receives as the output of the Optimizer. These steps are called *plastic steps*, contain column and row information, but no data parcel values, and are later transformed into *concrete steps* by OptApply.

Plastic Steps and Concrete Steps

Plastic Steps

Plastic steps are the output of the Generator. Except for any hard-coded literal values that may be used, the plastic steps contain column and row information, but do not have any data parcel values associated with them. Plastic steps can be cached for potential reuse.

A single request parcel can generate many plastic steps.

Plastic steps allow data values from a data parcel in a parameterized-independent request to be inserted into the optimized query plan by OptApply. For more information on parameterized requests, see [Parameterized Requests](#) and [Dynamically Parameterized Requests](#).

Concrete Steps

Concrete steps are the output of OptApply. They are context- and data-laden AMP directives that contain user- and session-specific information in addition to data parcels.

Each concrete step is composed of the following parts:

- System message header

Contains the message class and kind, length, logical host identifier, session and request number, version number, and character set code.

- Common step header

Contains account numbers and routing information. Specifies who is to receive the step, what should be done with the responses generated by the step, how the AMPs signal task completion, and what is the context for the step (for example, a transaction number).

- Step components

A series of operation-dependent fields including step flags and the step mode.

- Data

Contained in a varying length data area.

Generator Processes

The Generator performs the following 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. The plastic steps are cached, as determined by the request caching logic.
4. The plastic steps are then sent to OptApply.

OptApply

OptApply reformats plastic steps into concrete steps. For definitions of these types of steps, see [Generator](#).

Data from a USING data parcel is set in a parameterized PK request, if present, into the concrete steps. For other parameterized requests, the Parameterized Value Peek subsystem may place USING data into the parse tree prior to the Query Rewrite/Optimizer phases of request processing. If not peeked, or if there are dynamically parameterized literals, parameterized data is attached as part of the concrete step.

Concrete steps are passed to the Dispatcher, which then distributes them to the AMPs via the BYNET.

For more information on the Parameterized Value Peek subsystem, see [Parameterized Requests](#). For more information on dynamic parameterization of literals, see [Dynamically Parameterized Requests](#).

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

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

Concrete steps are also referred to as AMP 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.

Dispatcher

The Dispatcher

The Dispatcher exercises execution and response control for requests and dynamic plan fragments, and manages transaction and request aborts and the queue table cache.

The Dispatcher has the following primary functions:

- Apply TASM rules and classify into workload.
- Route AMP steps to the appropriate AMPs.
- Return result or statistics feedback to the optimizer so it can generate the next plan fragment when executing a dynamic plan.
- Return the results of a request to the user application that submitted that request.
- Notify client applications and Teradata platform processors of aborted transactions and requests.

Execution Control

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

The Dispatcher routes the concrete steps in the plan for a request 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.

Vantage can dispatch a parallel step if the previous step has finished acquiring its task locks, but has not yet completed its work. This is signalled by a response from the affected AMPs, and the Dispatcher determines that it can send another parallel step.

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

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

Response Control

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

Transaction and Request Abort Management

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

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

Transactions and requests can abort for several reasons, both normal and abnormal. Typical normal aborts are caused by the following actions:

- TDP logoff command
- Client application terminates normally

Typical abnormal aborts are caused by the following actions:

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

Queue Table FIFO Cache Management

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

The queue table FIFO cache row entries on a given PE are shared by all queue tables that hash to that PE. Each queue table row entry is a pair of QITS and rowID values for each row to be consumed from the queue, sorted in QITS value order.

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.

For more information about the queue table FIFO cache and its operation, see the CREATE TABLE (Queue Table Form) command in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Query Rewrite, Statistics, and Optimization

The information provided in this discussion is designed to help you to interpret EXPLAIN reports more accurately and to explain and emphasize the importance of maintaining accurate statistical and demographic profiles of your data. This discussion does not describe query optimization in detail.

Join processing is described in [Join Planning and Optimization](#).

Query Rewrite

About Query Rewrite

For the Teradata system, query rewrites are mostly performed in the Query Rewrite Subsystem of the Parser; however, some query rewrites are performed in the later phases. The Query Rewrite Subsystem is invoked by the Parser just prior to query optimization, and is the first phase of optimization performed in the query optimization process.

Function of Query Rewrite

Stated semi-formally, Query Rewrite is the process of rewriting query Q as query Q' such that the following criteria are both true:

- Query Q and Query Q' produce the identical answer set.
- Query Q' runs faster (which means that it is less costly) than Query Q .

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

With many SQL queries now being created by query generator tools that do not write optimally efficient code for a given target SQL processor, Query Rewrite has become more crucial than ever. Even trivial SQL requests can be written in an enormously large number of different ways.

Consider the following verbally expressed query: "Get the names of suppliers who supply part P2." Using the features available in the ANSI SQL-92 version of the language, it is possible to express this query in at least 52 different ways, assuming the system must access two tables in the process of returning the answer set. This example only indicates the number of possible ways of writing the verbally expressed query as an SQL request; it does not include the number of internal rewrites of those 52 individual SQL queries.

Examples of Query Rewrites

Because relational systems are based on rigorous mathematical principles, there is a good deal of algebraic simplification and term reduction that can be made for most queries. Similarly, there are often semantically equivalent operations that can be substituted for analytically more difficult operations in order to simplify processing. The vast majority of these semantically equivalent rewrites cannot be specified in the query

itself, examples being the substitution of a join index for an explicitly specified join operation and the pushing of projections.

The following examples show how Vantage rewrites submitted queries for faster processing.

Converting ANSI Join Syntax To Comma Join Syntax

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

```
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;
```

The conversion of a join from INNER join syntax to the equivalent comma syntax presents one canonical form of inner joins to the Join Planner. This guarantees consistent plans for both INNER join and comma syntaxes and eliminates the need for duplicate code to handle them both.

Projection Pushdown

Pushing projections can allow views to be folded that would remain as spools otherwise. (For more information on view folding, see [View Folding](#).) For example, views containing CASE expressions are not always merged. However, if these expressions can be removed, the system might be able to perform additional rewrites through the use of view folding.

Examples of Projection Pushdown

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

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

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

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

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

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

The request does not reference any columns from `sales_by_product`. As a result, the select list of `sales_by_product` can be rewritten as `SELECT 0`, because the rewrite only needs to ensure that the correct number of rows is returned.

Another special case, again based on the `sales_by_product` view, occurs when no columns of a view or derived table are referenced by the request, and the view is guaranteed to return only one row.

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

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

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

Projection pushdown should help the performance of spooling views by removing any dereferenced columns, thus reducing the size of spools.

IN-List Rewrite

When the number of values in an IN-list for IN or NOT IN predicates exceeds the setting of the `INListRewriteThreshold` field in DBS Control, the IN-list is checked to see whether IN-list rewrite processing is applicable. For more information on DBS Control fields related to IN-list rewrite processing, see *Teradata Vantage™ - Database Utilities*, B035-1102. The IN-list rewrite processing spools the IN-list values and rewrites the IN-list to a subquery referencing the spool.

IN-list query rewrite for a list that is not in a CASE expression

Assume the following query.


```
SELECT SUM(net_rev_amt)
FROM prod_db.fxf_ship_rev_credit_comp a
WHERE a.payor_cust_nbr IN ('14390630' , '15449611' , '15454443' ,...);
```

Because the IN-list is not in a CASE-expression, subquery-based rewrite is applied, resulting in the following query. (The Optimizer will do join planning after the IN-list rewrite is applied.)

```
SELECT SUM(prod_db.a.net_rev_amt )
FROM prod_db.fxf_ship_rev_credit_comp a
WHERE a.payor_cust_nbr IN
      (SELECT InListSpool_1.payor_cust_nbr
       FROM InListSpool_1);
```

IN-list query rewrite for a list that is in a CASE expression

Assume the following query.

```
SELECT SUM(net_rev_amt)
FROM prod_db.fxf_ship_rev_credit_comp a
WHERE CASE WHEN a.payor_cust_nbr IN ('14390630' , '15449611' ,...)
      THEN rev_shp_cnt ELSE rev_pcs END > 120 ;
```

Because the IN-list is in a CASE expression, outer-join-based IN-list rewrite is applied, resulting in the following query.

```
SELECT SUM(prod_db.a.net_rev_amt )
FROM ship_rev a
LEFT OUTER JOIN InListSpool_1
ON ( (a.payor_cust_nbr = InListSpool_1.payor_cust_nbr )
WHERE (( CASE WHEN (NOT (InListSpool_1.payor_cust_nbr IS NULL ))
      THEN (a.rev_shp_cnt )
      ELSE (a.rev_pcs ) END ))> 120 ;
```

Outer Join-to-Inner Join Conversion

In certain cases, an outer join in a query block can be converted to an inner join.

If a SELECT request specifies NORMALIZE, Query Rewrite does not push it to derived tables for a temporal outer join rewrite.

Consider the following example:

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

The outer join in this request can be safely converted to an inner join because the subsequent WHERE predicate filters any non-matching rows. By also applying the inner join rewrite, this request is rewritten as follows:

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

Similarly, the outer join in the following request can be converted to an inner join because the ON clause predicate `b2 < 10` is false if `b2` is null, and therefore the condition removes all non-matching rows of the outer join.

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

View Folding

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

The semantics of views and derived tables are identical; therefore, any assertion 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 assertion.

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

Examples of View Folding

Consider the following view definition and query against it:

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

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

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

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

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

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

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

```
SELECT *
FROM sales;
```

In this example, sales is a view involving 11 UNION ALL operations among the 12 sales months for the year.

This query is rewritten as follows:

```
SELECT *
FROM sales1
UNION ALL
SELECT *
FROM sales2
UNION ALL
...
UNION ALL
SELECT *
FROM sales12;
```

Consider the following view definition and query against it:

```
CREATE VIEW jan_sales AS
SELECT sales_product_key, ZEROIFNULL(quantity) AS qty
FROM sales1;
SELECT product_name, SUM(qty)
FROM product LEFT OUTER JOIN jan_sales
      ON product_key=sales_product_key
GROUP BY product_key, product_name ;
```

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

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

Predicate Simplification

Predicate simplification is a part of the query rewrite system that transforms a query into an equivalent form, but one that is simpler and more amenable to query optimization. Several of the key categories of predicate simplification are outlined in this section.

Satisfiability and Transitive Closure

The SAT-TC (SATisfiability-Transitive Closure) rewrite analyzes a set of predicates to determine if either of the following can be exploited to rewrite a request:

- A contradiction.

A simple example is a condition like `a=1 AND a=0`.

- Inferring new predicates from the existing predicate set by using transitive closure.

For example, consider a request that specifies the predicate `a=1 AND a=b`. This implies that `b=1`, which might be useful information for rewriting or otherwise optimizing the request.

Other examples of deriving new conditions by using transitive closure include the following samples:

```
A=B AND A=C --> B=C
A=5 AND A=B --> B=5
A=5 AND A IS NULL --> FALSE
A=5 AND A IS NOT NULL --> A=5
X > 1 AND Y > X --> Y >= 3
X IN (1,2,3) AND Y=X --> Y IN (1,2,3)
```

Transitive closure is also applied in the context of extract predicates. For example, for the set of conditions `{o_orderdate='1999-05-01' AND EXTRACT(MONTH FROM o_orderdate)>2}`, the system

adds `EXTRACT(MONTH FROM o_orderdate)=5` based on `o_orderdate='1999-05-01'`, which can then be used to simplify the existing `EXTRACT` predicate in the query.

For some categories of DML requests specified with `BEGIN` `END` constraints on a column with a `Period` data type, the query rewrite system adds the implied constraint `BEGIN(column_1) < END(column_1)` to help to identify unsatisfiable conditions. When these unsatisfiable conditions occur, `EXPLAIN` text for the query identifies them using the `EXPLAIN` text phrase *unsatisfiable* (see [EXPLAIN Request Modifier Phrase Terminology](#)).

As an example, consider the following table definition:

```
CREATE TABLE t1 (a INTEGER b PERIOD (DATE));
```

Suppose that you use the following query:

```
SELECT *
FROM t1
WHERE BEGIN(b) > DATE '2010-02-03' AND
      END(b) < DATE '2010-02-03';
```

The query rewrite system adds an implied constraint `BEGIN(b) < END(b)`, and is able to derive unsatisfiability in conjunction with the query predicates.

The query rewrite system can also apply transitive closure across query blocks, meaning transitive closure between outer and inner query blocks. This allows conditions to be pushed into and out of subqueries. The basic approach is to combine the query block conditions before computing the transitive closure. The `IN` and `NOT IN` operators are treated as `=` and `≠` operators, respectively. Derived conditions are added, as appropriate, to each query block.

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*. A connecting condition is one that connects an outer query with a subquery. See [Connecting Predicates](#) for further information about connecting conditions.

Applications of Transitive Closure

Transitive Closure (TC) can optimize date ranges and IN clauses. The following request 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;
```

The new set of constraints that can be derived is as follows:

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

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

You might notice performance improvements for some queries because of the extra predicates TC adds. The extra predicates can also be seen in the EXPLAIN reports for requests that use them.

SAT-TC and Query Rewrite

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

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

For this request, transitive closure adds the inferred predicate sales_product_key=10 to the ON clause. This is particularly effective when the predicate added to the ON clause is a constraint on the primary index of the inner table in a join.

Another important property of transitive closure is its ability to infer new predicates across the ON clauses of consecutive inner joins. For example, consider the following request.

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

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

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

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

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

One way to take advantage of a contradictory condition is to add CHECK constraints to tables, enabling Query Rewrite to eliminate unnecessary conditions, thus permitting the Optimizer to later construct better execution plans (see [Applications of Transitive Closure](#)).

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 that is a UNION ALL of `orders1`, `orders2`, ..., `orders12`, and the query looks like this:

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

Without CHECK constraint, the system must access all of the tables orders1, orders2, ... orders12 using the constraint `EXTRACT(MONTH FROM o_orderdate)<=3`, even though it only needs to access orders1, orders2, and orders3 to satisfy the request. The only way Query Rewrite knows to filter out the other nine tables is to add CHECK constraints for every table, and then to determine the contradiction between the CHECK constraints and the query constraint.

For example, if the CHECK constraint on order4 is added, Query Rewrite sees the following compound predicate, which is a contradiction.

```
EXTRACT(MONTH FROM o_orderdate)<= 3      -- from query
AND
EXTRACT(MONTH FROM o_orderdate)=4        -- from CHECK constraint
```

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

Of course, you would rarely submit a contradictory query that does not return results regardless of the data like `a=1 AND a=2`. The example in the previous paragraph indicates the need for such checks. As previously stated, this issue is referred to as the satisfiability problem. Any solution to satisfiability generates a set of conditions 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 conditions is satisfiable.

You should be aware of the cost of enforcing CHECK constraints if you decide to use them for horizontal table partitioning. This type of table partitioning is not identical to the row partitioning that you can specify with a partitioning expression in a PARTITION BY clause for a table or join index, though it is similar conceptually.

SAT-TC and Query Optimization

Query optimization introduces the following additional applications of satisfiability in the usage and maintenance of join indexes:

- Determining whether a join index must be updated to keep it synchronized with an update operation on one or more of its base tables. The term update operation here signifies an insert, update, or delete operation against the base table in question.
- Determining whether a join index partly or fully covers a query.

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

Assume the following sparse join index definitions:

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

CREATE JOIN INDEX j2 AS
```



```
SELECT *
FROM lineitem
WHERE EXTRACT(MONTH FROM l_shipdate)>=7;
```

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

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

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 the following predicate:

```
EXTRACT(MONTH FROM l_shipdate)=12
AND
EXTRACT(MONTH FROM l_shipdate)>=7
```

but FALSE for this predicate.

```
EXTRACT(MONTH FROM l_shipdate)=12
AND
EXTRACT(MONTH FROM 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 when more complex conditions, like constants in WHERE clause predicates, are specified in the join index definition. This happens, for example, when you create a sparse join index. See the information about hash and join indexes in *Teradata Vantage™ - Database Design*, B035-1094 and the documentation for CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information about sparse join indexes.

Constant Predicate Evaluation

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

```
t1.pi=t2.pi
OR 'a' IN ('b','c')
```

The system identifies and evaluates constant predicates and then replaces the predicate with the result if it is either TRUE or FALSE.

The IN predicate in this example can be evaluated to FALSE when the database parses the request, so Query Rewrite transforms it as shown below:

```
t1.pi=t2.pi
```

Domain-based Simplification

Domain-based simplification uses the domain range of the underlying column to simplify a predicate. For example, consider the following table definition:

```
create table t1 (a1 smallint)
```

With this table definition, the predicate `a1 = 64000` leads to unsatisfiability, as the constant 64000 is not within the smallint range of values:

```
a1 = 64000 => FALSE
```

Similarly, consider the following predicate:

```
a1 in (32800, 80000, 1, 2, 3)
```

This predicate can be simplified as shown below, as the constants 32800 and 80000 are not within the smallint range of values:

```
a1 in (1, 2, 3)
```

Check Constraint-based Simplification

Check constraint-based simplification is similar to domain-based simplification, but the range of the underlying column is based on check constraints specified on the column. For example, consider the following table definition:

```
create table t1 (a1 int NOT NULL check (a1 < 10))
```

With this table definition, the following predicate leads to unsatisfiability, as the predicate is outside the allowed range of values based on the check constraint:

```
a1 > 30 => FALSE
```

Consolidating Single-Table Predicates

Query rewrite evaluates single-table predicates and rewrites them to eliminate overlapping or redundant conditions. For example, consider the following set of predicates:

```
a1 NOT IN (1,3) AND
a1 <4 AND
a1 >= 2
```

If a1 is an INTEGER column these predicates can be simplified as follows:

```
a1=2
```

Similarly, consider the following predicate:

```
a1 IN (1, 3, 5, 7, 9) AND
a1 > 4
```

where a1 is an INTEGER column.

The predicate can be simplified as follows:

```
a1 IN (5,7,9)
```

Consolidation of single-table predicates enables the Optimizer to calculate more accurate selectivity estimates while at the same time eliminating the execution of redundant predicates.

Query Rewrite also supports simplification of single-table predicates for the BEGIN and END bound functions on Period columns.

For example, consider the following predicate based on the BEGIN bound function:

```
BEGIN(b)>DATE '2005-02-03' AND
BEGIN(b)>DATE '2010-02-03'
```

Query Rewrite simplifies this to the following predicate:

```
BEGIN(b)>=DATE '2010-02-04'
```

Similarly, consider the following predicate based on the END bound function:

```
END(b)< DATE '2005-02-03' AND
END(b)< DATE '2010-02-03'
```

Query Rewrite simplifies this to the following predicate:

```
END(b)<= DATE '2005-02-02'
```

Consolidation of single table predicate can lead to unsatisfiability as follows:

```
a1 NOT IN (1, 2, 3) AND
a1 >= 1 AND
a1 <= 3
```

If a1 is an INTEGER column, the above conditions are non-overlapping and the predicate is rewritten as follows:

```
0=1
```

0=1 evaluates to FALSE.

In other cases, the predicate can be simplified to a predicate that does not constrain the set of possible values, as shown:

```
a1 > 1 OR a1 < 2
```

The predicate can be rewritten as follows:

```
a1 IS NOT NULL
```

For a more complex example, consider the following predicate:

```
(a1 >= 1 AND a1 <= 3) OR
(a1 >= 4 AND a1 <= 10)
```

If a1 is an INTEGER column, the predicate can be rewritten as follows:

```
a1 >= 1 AND a1 <= 10
```

Constant Movearound

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

```
<column> <±> <constant_1> <comparison_operator> <constant_2>
```

Constant movearound rewrites predicates of the form above to predicates of the following form:

```
<column> <comparison_operator> <constant_3>
```

The method does this by removing the plus or minus operation from the column side of the predicate and then adding its negation to the constant-only side of the predicate. The constant expression is then folded.

Any errors that occur during folding, such as overflow or underflow, cause the rewrite to be rejected in favor of using the original expression.

This transformation is only done for the plus and minus arithmetic operators.

For a simple example, consider $a1 + 1 > 4$, which can be rewritten as $a1 > 3$.

The constant expression that is moved can also be an INTERVAL expression, as in DATE operations.

For example, consider the following predicate:

```
date_col + INTERVAL '3' MONTH <= '2007-03-31'
```

Using constant movearound, this predicate can be rewritten as follows:

```
date_col <= '2006-12-31'
```

Constant Substitution

The constant substitution rewrite substitutes the values of columns, if they are available, and attempts to derive unsatisfiability or to simplify the predicate, if possible. Predicates of the following form are simplified by substituting the values for column_1 and column_2 whenever possible:

```
<column_1> <±> <constant_1> <comparison_operator>  
<column_2> <±> <constant_2>
```

As an example, consider the following predicate:

```
a = 10 AND  
b = 20 AND  
a + 2 = b + 1
```

Substituting the value 10 for a and 20 for b, the predicate can be simplified to $12 = 21$, which is FALSE.

For additional examples, consider $a > a + 1$, which simplifies to FALSE and $a >= a - 1$, which is rewritten as $a \text{ IS NOT NULL}$.

Simplification by Distribution

This simplification by distribution rewrite attempts to simplify complex AND/OR predicates by using distributive properties.

Consider the following predicate:

```
a > 5 AND b < 6 AND (a < 2 OR b > 9)
```

By distributing the AND into the OR branch, the predicate can be simplified as follows:

```

a > 5 AND b < 6 AND ((a > 5 AND a < 2) OR (b < 6 AND b > 9)) =>
a > 5 AND b < 6 AND ( FALSE OR FALSE) =>
a > 5 AND b < 6 AND FALSE =>
FALSE

```

Factoring Common Predicates

The factoring of common predicates is another rewrite based on distributive properties. If a complex AND/OR predicate has some common predicates across the AND/OR branches, these common predicates can be factored out to simplify the predicate.

Consider the following predicate:

```

(a1 = 1 AND b1 = 1 AND c1 = 1) OR
(a1 = 1 AND b1 = 1 AND d1 = 1) OR
(a1 = 1 AND e1 = 1)

```

Note that $(a1 = 1 \text{ AND } b1 = 1)$ is common to the first two branches of the overall OR predicate. In the first step of factoring, $(a1 = 1 \text{ AND } b1 = 1)$ can be factored out, resulting in the following equivalent predicate:

```

(a1 = 1 AND b1 = 1 AND (c1 = 1 OR d1 = 1)) OR
(a1 = 1 AND e1 = 1)

```

After the first step, note that $(a1 = 1)$ is common to the remaining two branches of the OR predicate. In the second step of factoring, $(a1 = 1)$ can be factored out, resulting in the following equivalent predicate:

```

a1 = 1 AND ((b1 = 1 AND (c1 = 1 OR d1 = 1)) OR e1 = 1)

```

Simplification Based on Containment

Simplification based on containment attempts to simplify complex AND/OR predicates by using the set containment properties of set algebra. There are two kinds of containment rewrite rules: the OR containment rule and the AND containment rule.

The OR containment rule is as follows:

```

If A contains A',
  A OR (A' AND ..) => A

```

For example, $a > 5 \text{ OR } (a > 7 \text{ AND } b < 6)$ can be simplified to $a > 5$.

The AND containment rule is as follows:

```
If A' contains A,
  A AND (A' OR ..) => A
```

For example, $a < 10 \text{ AND } (a < 12 \text{ OR } b < 6)$ can be simplified to $a < 10$.

Duplicate Predicate Removal

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

Predicate Pushdown and Pullup

The justification for predicate pushdown is to move operations as close to the beginning of query processing as possible in order to eliminate as many rows and columns 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 (The concept of parse trees is described in [Translation to Internal Representation](#), as is predicate pushdown).

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

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

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

Examples of Predicate Pushdown and Pullup

Suppose you have the following request:

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

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

Pushing predicates helps performance by reducing the cardinality of spools for views and derived tables that cannot be folded.

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

For example, consider the derived table `dt(x,y,z)` 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 it. The 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 required a full-table scan of `t1`, while the rewritten query requires only a single-AMP access of `t1`, assuming `a1` is the primary index for `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 rewritten select list of the view definition for this request. This action reduces the size the spool for the view (assuming the view is spooled) and eliminates 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 request 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;
```

Eliminating Set Operation Branches

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

Eliminating Branches With Unsatisfiable Conditions

Suppose you have created table sales2 with the following partial definition:

```
CREATE TABLE sales2 (
  ...
  sales_date DATE
  ...}
CONSTRAINT feb_only CHECK (EXTRACT(MONTH FROM sales_date)=2));
```

Consider the following example:

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

UNION ALL

SELECT *
```

```
FROM sales2
WHERE EXTRACT(MONTH FROM sales_date)=1;
```

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

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

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

Eliminating Redundant Joins

Joins are among the most frequently specified operations in SQL queries. They are also among the most demanding resource consumers in the palette of an SQL coder. Join elimination removes unnecessary tables from queries.

Join elimination is most commonly used in the following situations:

- Inner joins based on any form of referential integrity between 2 tables.
- Left and right outer joins can be eliminated if the join is based on unique columns from the right table in the join specification.

In both cases, a join and a table (the table eliminated is the parent table in inner joins and the inner table in outer joins) 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, Vantage processes joins with relative ease. Nevertheless, when joins can be eliminated or made less complex, the result is always better performance. For an example of how the Optimizer recognizes and eliminates unnecessary or redundant joins, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Primary key-foreign key relationships between tables are very common in normalized data models and their corresponding physical databases. Define the parent tables in such a relationship as *PK-tables* and the child tables as *FK-tables*. You can define these relationships explicitly by specifying referential integrity constraints using CREATE TABLE and ALTER TABLE (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - Database Design*, B035-1094).

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

Vertical Partitioning

An obvious way to partition tables vertically is to define them as column-partitioned tables. This should be your first choice for vertical partitioning, and you should only use vertical table partitioning for those cases where column-partitioning is not possible, for example when partitions require different attributes.

Another application where you might want to implement referential integrity constraints between 2 tables is the vertical partitioning of a single wide table. Vertical partitioning means that a table with a very large number of columns, many of which are seldom queried, is split into 2 tables where one table retains the frequently accessed columns and the other table has the less frequently accessed columns. To ensure consistency between the tables, thereby maintaining a robustly consistent virtual table that contains correct data in all of the columns, a referential integrity constraint is needed between the tables. The choice of the PK-table and the FK-table in this case is arbitrary, but the best practice is to define the table with the frequently accessed columns as the FK-table.

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

Example of Join Elimination for Redundant Joins

The following request illustrates redundant joins:

```
SELECT s.supkey, s.address, n.nationkey
FROM supplier AS s, nation AS n
WHERE s.nation_key=n.nation_key
ORDER BY n.nation_key;
```

The join is redundant in this request because every row in supplier has exactly one match in nation based on the following referential integrity definition for the supplier table.

```
FOREIGN KEY (nation_key) REFERENCES nation (nation_key).
```

Also, only n_nationkey is referenced in the request, and it can be replaced by s.nation_key. The modified query after Join elimination looks as follows:

```
SELECT s.sup_key, s.address, s.nation_key
FROM supplier
ORDER BY s.nation_key;
```

Conditions for Eliminating a PK-FK Join

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

- There is a referential integrity constraint defined between the 2 tables.
- The query conditions are conjunctive.
- No columns from the PK-table other than the PK-columns are referenced in the request. This includes the SELECT, WHERE, GROUP BY, HAVING, ORDER BY, and other clauses.
- The PK-columns in the WHERE clause can only appear in PK=FK (equality) joins.

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

Be aware that the cost of referential integrity maintenance can exceed the benefit of join elimination. To mitigate this possibility, you can substitute either a batch referential integrity constraint or a Referential Constraint. For details of these referential integrity types, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - Database Design*, B035-1094.

The following table outlines the differences between these 2 referential integrity mechanisms.

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

Consider this SELECT request.

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

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

The following example illustrates outer join elimination:

```
SELECT sales1.*
FROM sales1 LEFT OUTER JOIN product
ON sales_product_key=product_key;
```

Pushing Joins Into UNION ALL Branches

Representing a fact table as a UNION ALL view of horizontal partitions of fact table rows is a commonly used technique.

These UNION ALL views are often joined to constrained dimension tables as shown in the following example view and query against that view:

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

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

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

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

SELECT quantity, amount
FROM sales2, product
WHERE sales_product_key=product
AND   product_name LIKE 'French%' ) AS jan_feb_sales ;
```

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

Other Query Rewrites

Not all query rewrites are performed by the Query Rewrite subsystem. Some, such as the Partial GROUP BY optimization (see [Partial GROUP BY Block Optimization](#)), are performed as late in the optimization process as join planning.

The topics that follow briefly describe some of the rewrites and the parse tree transformations that are done by the Optimizer after the Query Rewrite subsystem has completed its tasks.

Predicate Marshaling

The first step in the process of query rewrites undertaken by the Optimizer is to marshal the predicates, both ordinary and connecting, for the DML operations presented to the Optimizer by the rewritten ResTree'. Note that this particular application of transitive closure is not undertaken by the Query Rewrite subsystem.

In the process of marshaling predicates, the query predicates are first, whenever possible, isolated from the parse tree, converted to conjunctive normal form, or CNF (see [Path Selection](#) 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.

Applying Predicate Conditions

Conditions on multiple relations are converted by expansion. For example, consider the following:

```
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. The maximum number of ORed expressions that can be specified is 1,048,546.

Transitive Closure

Where possible, the Optimizer derives new conditions using transitive closure. See [Predicate Simplification](#) for additional information about transitive closure.

Connecting Predicates

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.

Pushing Predicates Down the Parse Tree

The next step is to push the marshaled predicates down the parse tree as far as possible. See [Translation to Internal Representation](#) and [Predicate Pushdown and Pullup](#) for additional information on predicate push down and pullup.

View Materialization and Other Database Object Substitutions

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

For example, all views referenced by a query must be resolved into their underlying base tables or covering indexes before the query can be performed. These rewrite methods also replace base tables and table joins with hash, join, or even secondary indexes whenever the substitution makes the query perform more optimally. These substitutions also apply to view materialization if base tables referenced by the view can be replaced profitably, Partial GROUP BY rewrites, and Common Spool Usage rewrites. 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.

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

Query Optimizers

The Goal of Query Optimization

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

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

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

Some of these factors are predetermined algorithms and heuristics inside the Optimizer and beyond your control, but you can control and manipulate other factors by modifying the physical database design and by fine tuning your queries.

Among the user-controllable factors are the following items:

- Table layout and view choices
- The choice of whether a table has a primary [AMP] index or not
- The primary [AMP] index column set
- For a row-partitioned table, the choice of partitioning expressions
- For a column-partitioned table, the choice of columns in each of the column partitions and the column partition options
- The selection and type of secondary, join, and hash indexes
- The availability and accuracy of column and index set statistics
- The size of the statistics cache
- The query. While Vantage attempts to provide an extensive set of query rewrites, in some cases rewriting a query yourself may improve performance.

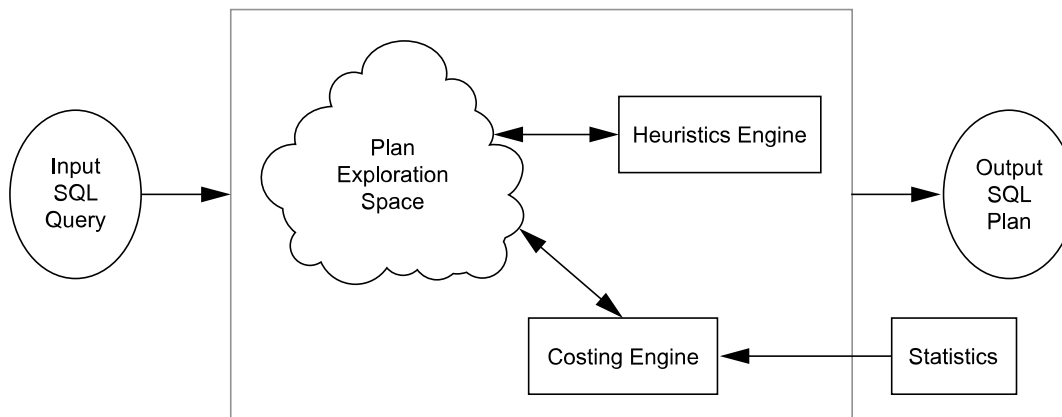
Why Relational Systems Require Query Optimizers

Because any query that meets the syntactical and lexical rules of SQL is valid, a relational system must be able to access, aggregate, 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.

Query Optimizer Overview

The following graphic is a very high-level representation of an SQL query optimizer:



The input to the Optimizer ("Input SQL Query" in the graphic) is the *ResTree* output from Query Rewrite. Because the original SQL request text has already been reformulated as an annotated parse tree, the ResTree input to the Optimizer is sometimes called ResTree or *Red Tree* to differentiate it from the ResTree/Red Tree that was input to Query Rewrite.

The *Plan Exploration Space* is a workspace where various query and join plans and subplans are generated and costed (see [Bushy Search Trees](#)).

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-Based Optimization](#)).

The Costing Engine primarily uses cardinality estimates based on summary demographic information, whether derived from collected statistics or from dynamic AMP sample estimates, on the set of relations being evaluated to produce its choice of a given "best" plan. The statistical data used to make these evaluations are maintained in the data dictionary (see [Optimizer Statistics and Demographics](#)).

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, Query Rewrite and the Optimizer perform the following actions in, roughly, the following order:

1. Translate an SQL query into some internal representation.
2. Rewrite the translation into a canonical form. The conversion to canonical form is often referred to as *Query Rewrite*. 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, Statistics, and Optimization](#) for more information about Query Rewrite.
3. Assess and evaluate candidate methods for accessing, joining, and aggregating database tables.

Join plans have the following 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 are joined can have a powerful impact on join cost. In this context, a table could be joined with a spool rather than another table. The term *table* is used in the most generic sense of the word. Both tables and spools are frequently categorized using the logical term *relations* when discussing query optimization.

4. Generate several candidate query plans and select the least costly plan from the generated set for execution. A plan is a set of low-level instructions called AMP steps (see [AMP Steps](#)) that are generated by the Optimizer to produce a query result in the least expensive manner (see [Statistics and Cost Estimation](#) for a description of plan costing).

Types of Query Optimizers

Following are the basic types of query optimizers:

- Rule-based optimizers

The determination of which candidate plan to use is based on a set of ironclad rules. Rule-based optimizers are also referred to as heuristic optimizers.

- Cost-based optimizers

The determination of which candidate 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.

In practice, rule-based optimizers have proven to be less high-performing in decision support environments than cost-based optimizers. The database query optimizer is cost-based.

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 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.

The Teradata Optimizer examines the following three cost criteria when it determines which of the candidate plans it has generated shall be used.

- Interconnect (BYNET communication) costs.
- I/O costs, particularly for secondary storage.

- CPU (computation) costs.

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 quickly find a superior strategy from a set of candidate strategies.
- To avoid strategies that are obviously poor.

Teradata Optimizer Processes

This topic provides a survey of the stages of query optimization undertaken by the 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 performs them.

Query Optimization Processes

The following processes list the logical sequence of the processes undertaken by the Optimizer as it optimizes a DML request. The processes that are listed here do not include the influence of parameterized value peeking to determine whether the Optimizer should generate a specific plan or a generic plan for a given request (see [Parameterized Requests](#)) other than to note that it does make that determination.

The input to the Optimizer is the Query Rewrite ResTree' (see [Query Rewrite, Statistics, and Optimization](#)). The Optimizer then produces the optimized white tree, which it passes to an Optimizer subcomponent called the Generator (see [Generator](#)).

The Optimizer generates a *static plan* for all of the requests it processes by computing the cost of all the possible plan variations and selecting the plan with the lowest cost, and often chooses to use a static plan for a request. During the process of generating a static plan, the Optimizer assumes that all of the compilation-time demographic information it has is accurate and generates the plan for the entire request. However, this assumption is not always true, particularly for complex queries, where even all the advanced demographic estimation methods used by Vantage such as derived statistics and enhanced costing formulas can generate inaccurate cardinality, CPU usage, and I/O counts for the intermediate steps of a plan that lead to poor static plans.

If a request happens to contain some independent components such as noncorrelated scalar subqueries, nonfolded derived tables or views, remote table operators, or some combination of these, the Optimizer can fragment the request into smaller components called request fragments. The demographics and inferred data from the execution of the plan fragments are used to plan subsequent request fragments. The request fragments are planned and executed incrementally. See [Incremental Planning and Execution](#) for more information about this process.

The Optimizer first generates a static plan for a request and then determines whether to execute the static plan or to generate and execute a dynamic plan. Information gathered while generating the static plan is used as part of this determination.

The Optimizer either sends the complete static plan or the summary information from the static plan as part of the first dynamic plan fragment to the dispatcher to apply workload filters, throttles, and classification criteria for the request. This behavior is controlled by an internal DBS Control field. Contact Teradata Services if this field needs to be changed.

If the request passes the filters and throttles, the Optimizer continues processing the request with incremental planning and execution (IPE). The system does not apply the workload filters, throttles, and classification criteria to a plan fragment of a dynamic plan. Instead, the dynamic plan is executed using the workload definition that was determined for the request based on the static plan. Workload exceptions based on accumulated runtime metrics are applied during the execution of plan fragments of a dynamic plan.

The system employs the following process stages to optimize a request:

1. Receives the Query Rewrite ResTree' as input.
2. If the request has been processed previously, determines whether to generate a specific plan or a generic plan for it.
3. Generates a static plan for the request in the following stages.
 - a. Processes correlated subqueries by converting them to unnested SELECTs or simple joins.
 - b. Processes noncorrelated scalar subqueries by materializing the subquery and placing its value in the USING row for the query.
 - c. Searches for a relevant join or hash index.
 - d. Materializes subqueries to spools.
 - e. Analyzes the materialized subqueries for optimization possibilities in the following stages:
 - a. Separates conditions from one another (see [Predicate Marshaling](#)).
 - b. Pushes down predicates (see [Predicate Pushdown and Pullup](#)).
 - c. Generates connection information.
 - d. Locates any complex joins.
 - e. Discovers aggregations and opportunities for partial group by optimizations.
 - f. Generates size and content estimates of spools required for further processing (see [Optimizer Use of Statistical Profiles](#)).
 - g. Generates an optimal single-table access path.
 - h. Simplifies and optimizes any complex joins identified in stage 3.e.iv.
 - i. Generates information about local connections. A connecting condition is one that connects an outer query and a subquery. A direct connection exists between 2 tables if either of the following conditions is found.
 - ANDed bind term: miscellaneous terms such as inequalities, ANDs, and ORs; cross, outer, or minus join term that satisfies the dependent information between the 2 tables
 - A spool of an noncorrelated subquery EXIST predicate that connects with any outer table

- j. Generates 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.
 - k. Performs row and column partition elimination for partitioned tables.
 - l. Uses a recursive greedy 1-table lookahead algorithm to generate the best join plan (see [Determining the Order of Joins](#)).
 - m. If the join plan identified in stage 3.m does not meet the heuristics-based criteria for an adequate join plan, generate another best join plan using an n -table lookahead algorithm (see [Determining the Order of Joins](#)).
 - n. Selects the better join plan of the 2 plans generated in stages 3.m and 3.n.
 - o. If the join plan identified is heuristically determined to be adequate, generates a star join plan (see [Star and Snowflake Join Optimization](#)).
 - p. Selects the better plan of the selection in stage o and the star join plan generated in stage 3.p.
4. Determines whether to execute the static plan or to use IPE to generate a dynamic plan.

IF the Optimizer decides to ...	THEN it continues with this process stage ...
use the static plan	7
generate a dynamic plan	6

- 5. Sends either the static plan or the summary of the static plan to the Dispatcher to apply the workload filters, throttles, and classification criteria for the request. If the request passes the filters and throttles, the Optimizer continues to evaluate the IPE processing.
- 6. Generates a dynamic plan for the request under the IPE framework based on the static plan and using the following process stages.
- 7. Passes the optimized white tree to the Generator.

The Generator (see [Generator](#)) then generates plastic steps for the plan chosen in stage 3.q or stage 6.

Incremental Planning and Execution

Incremental Planning and Execution (IPE) is an adaptive query optimization framework in which the query optimizer can partially plan and execute a request incrementally. Vantage dynamically collects the intermediate results or statistics from each partial execution. The optimizer uses results feedback to rewrite the remaining portion of the request; this enables advanced optimizations such as partition elimination, join elimination, join index rewrite, and predicate simplification. The optimizer uses statistics feedback to produce a more accurate cost estimate, which is used to choose an optimal execution plan.

The following are some types of dynamic feedback in the two categories.

Results feedback:

- Single-row relations: base relations with equality predicates on UPI or USI.
- Noncorrelated scalar subqueries.
- Noncorrelated EXISTS or NOT EXISTS subqueries in a WHERE clause.

- Noncorrelated single-column IN, NOT IN, ANY, or ALL subquery in a WHERE clause.
- Single-row spools: nonfolded derived tables or views for which the result is determined to be one row.

Statistics feedback:

- Nonfolded derived tables or views.
- Remote tables, table operators, and table functions.
- Intermediate spools input to aggregations or analytic functions.

Based on execution, the kind of feedback is dynamically switched between results and statistics. AMPs choose the kind of feedback based on the actual execution and result size, regardless of the feedback kind requested by the optimizer:

- For the original results feedback, if the size of the target spool is too big, switching from results feedback to statistics feedback reduces processing loads on the optimizer to minimize parsing time.
- For the original statistics feedback, if the target spool is empty or contains only one row, switching from statistics feedback to results feedback enables complex queries to benefit from results feedback.

Example: IPE results feedback for a noncorrelated scalar subquery

First the optimizer detects the noncorrelated scalar subquery (bolded in the example).

```
SELECT *
FROM t1, (SELECT t2.a2
          FROM t2, t3
          WHERE t2.a2 = t3.a3
          GROUP BY t2.a2) AS dt (a2)
WHERE t1.a1 < (SELECT MAX(t4.a4)
             FROM t4)
AND t1.a1 = dt.a2;
```

Then the optimizer forms a fragment to execute the scalar subquery. The result of the scalar subquery is fed back to the optimizer and the result is substituted for the scalar subquery.

Assume that the result is 20. The substitution enables the following query rewrites:

- Using transitive closure to derive predicates.

```
SELECT *
FROM t1, (SELECT t2.a2
          FROM t2, t3
          WHERE t2.a2 = t3.a3
          GROUP BY t2.a2) AS dt (a2)
WHERE t1.a1 < 20
AND t1.a1 = dt.a2
AND dt.a2 < 20;
```

- Pushing the derived predicate into a derived table.

```

SELECT *
FROM t1, (SELECT t2.a2
          FROM t2, t3
          WHERE t2.a2 = t3.a3
          AND   t2.a2 < 20
          GROUP BY t2.a2) AS dt (a2)
WHERE t1.a1 < 20
AND   t1.a1 = dt.a2
AND   dt.a2 < 20;

```

- Using transitive closure to derive predicates inside the derived table.

```

SELECT *
FROM t1, (SELECT t2.a2
          FROM t2, t3
          WHERE t2.a2 = t3.a3
          AND   t2.a2 < 20
          AND   t3.a3 < 20
          GROUP BY t2.a2) AS dt (a2)
WHERE t1.a1 < 20
AND   t1.a1 = dt.a2
AND   dt.a2 < 20;

```

Further advanced optimizations as discussed above, such as partition elimination, can then occur.

Translation to Internal Representation

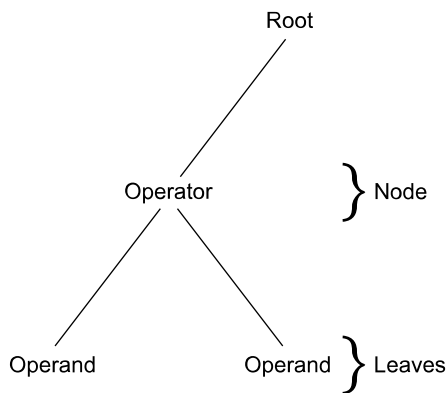
Parse Tree Representations of an SQL Request

Like most cost-based optimizers, the Optimizer represents SQL requests internally as parse trees. The foundation node 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, delimiters, 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 shown here with the root at the top with the leaves at the bottom., 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 subtrees are the operands of this operation. The subtree operands are typically expressions of some kind, but might also be various database objects such as tables.

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



Tables Used for the Examples

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

parts

part_num	description	mfg_name	mfg_part_num
PK		FK	
PI			

manufacturer

part_num	mfg_name	mfg_address	mfg_city
PK			
PI			

customer

cust_num	cust_name	cust_address	cust_city
PK			
PI			

order

order_num	cust_name	mfg_part_num	order_date
PK	FK		
PI			

Example SQL Request

Consider the following example SQL request:

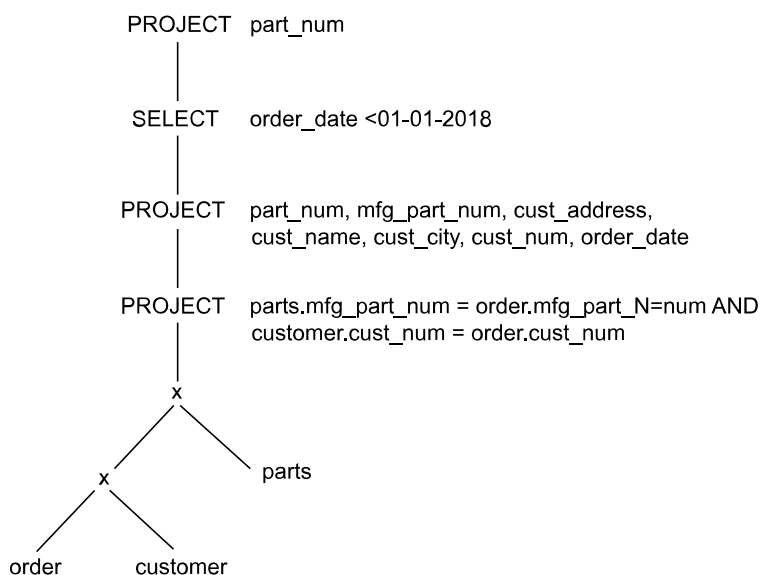
```
SELECT part_num, description, mfg_name, mfg_part_num, cust_name,
       cust_address, cust_num, order_date
FROM order INNER JOIN customer INNER JOIN parts
WHERE customer.cust_num = order.cust_num
AND parts.mfg_part_num = order.mfg_part_num
AND order_date < DATE '2001-01-01' ;
```

The initial translation of the request into a parse tree is performed by the Syntaxer (see [Syntaxer](#)) after it finishes checking the request text for syntax errors. Query Rewrite receives a processed parse tree as input from the Resolver, then produces a rewritten, but semantically identical, parse tree as output to the Optimizer. This request tree is just a parse tree representation of the original request text.

The Optimizer further transforms the tree by determining an optimal plan, and, when appropriate, determining 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 request. The parse tree is now an operation tree. It is a textual form of this tree, also referred to as a white tree, that the Optimizer uses to produce EXPLAIN text when you explain a request. Note that a separate subsystem adds additional costing information about operations the Optimizer does not cost to the white tree before any EXPLAIN text is produced for output.

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



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

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

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

The first set of processes is performed by Query Rewrite.

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

```
SELECT customer.cust_num = order.cust_num
SELECT parts.mfg_part_num = order.mfg_part_num
```

2. By commutativity, the SELECTION `order.order_date < 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 `order_date` is an attribute of the order table only, is as follows:

- a. Begin with the following predicate:

```
SELECT order_date < 2001-01-01((order X customer) X parts)
```

- b. Transform it to the following form:

```
SELECT (order_date < 2001-01-01(order X customer)) X parts
```

- c. Transform it further to the following form:

```
SELECT ((order_date < 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.

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

```
SELECT parts.mfg_part_num = order.mfg_part_num
```

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:

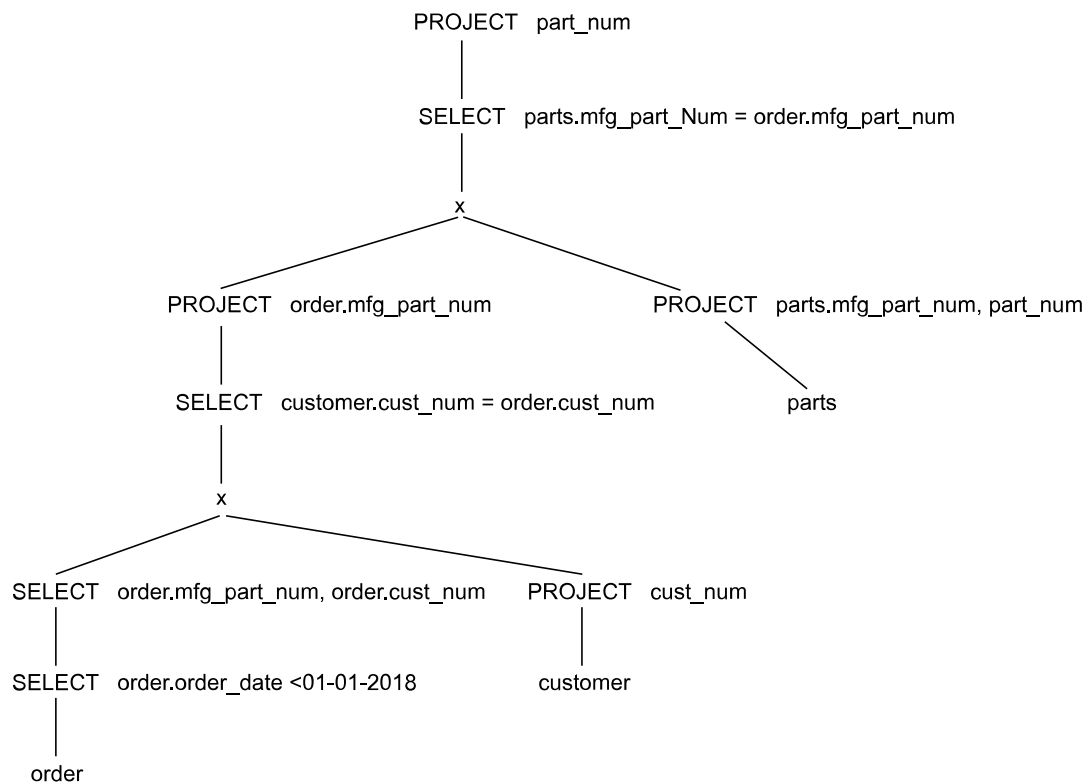
```
SELECT customer.cust_num = order.cust_num
```

This expression can be moved down to apply to the product: `order_date < 2001-01-01 (order)` X `customer`.

`order.cust_num` 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 2 PROJECTION operations in the original parse tree into the single PROJECTION `part_num`.

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.part_num
SELECT parts.mfg_part_num = order.mfg_part_num
PROJECT parts.part_num, parts.mfg_part_num, order.mfg_part_num
```

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

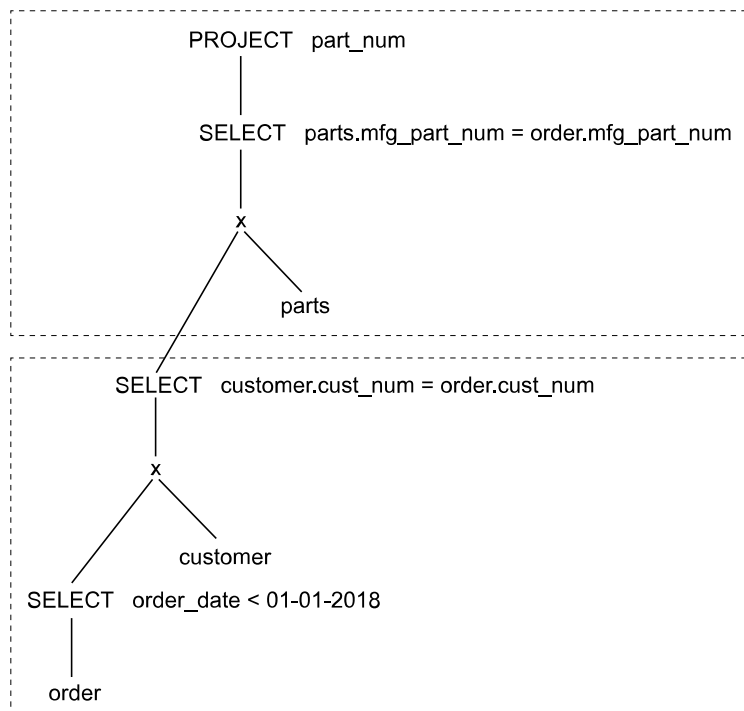
```
PROJECT part_num, parts.mfg_part_num
```

8. Similarly, apply PROJECT order.mfgpartnum to the left operand of the higher of the 2 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.mfg_part_num
SELECT customer.cust_num = order.cust_num
PROJECT order.mfg_part_num, customer.cust_num, order.cust_num
```

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.mfg_part_num, order.custnum, orderdate is redundant with respect to its PROJECTION term, so it is removed from further consideration in the query.

The transformed parse tree that results from all these transformations is shown in the following illustration:



The 2 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.

Optimizer Statistics and Demographics

Functions of Optimizer Statistics and Demographics

The Optimizer uses statistics and demographics for several different purposes. Without full or all-AMP sampled statistics, query optimization must rely on extrapolation (see [Using Extrapolation to Replace Stale Statistics](#)), object use counts (see [Object Use and UDI Counts](#)), or dynamic AMP sample estimates of table cardinality, which does not collect all of the statistics that a COLLECT STATISTICS request does.

Statistics and demographics provide the Optimizer with information it uses to reformulate queries in ways that permit it to produce the least costly access and join plans. 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 are you willing to accept?

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 stale, 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.

The Optimizer avoids many of the problems that histogram statistics have presented historically in two ways.

- By collecting object use and UDI counts (see [Object Use and UDI Counts](#)).
- By saving the dynamic AMP sample statistics, PARTITION statistics, or both, as attributes in the interval histogram. PARTITION statistics can be collected more quickly than other statistics and can be collected for row- and column-partitioned tables and for nonpartitioned tables.

Because dynamic AMP statistics for a table are always collected from the same AMP set (see [Dynamic AMP Sampling](#)), they can also be used to detect table growth with a high degree of accuracy (see [Using Extrapolated Cardinality Estimates to Assess Table Growth](#)).

Purposes of Statistics and Demographics

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

- The Optimizer uses statistics and demographics to determine whether it should generate a query plan that uses an index instead of performing a full-table scan.
- The Optimizer uses statistics and demographics to estimate the cardinalities of intermediate pools based on the qualifying conditions specified by the request.

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 spools be redistributed and then merge joined, or should one of the tables or spools 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.

- Statistics collected on the PARTITION system-derived column permit the Optimizer to better estimate costs and cardinalities. This is also true for nonpartitioned tables and column-partitioned tables.

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

Because the system usually can collect SUMMARY statistics very rapidly, you should collect these statistics for all tables when Vantage updates them after ETL jobs complete. You should also recollect PARTITION statistics for row-partitioned tables after ETL jobs complete or any time that you modify a partitioning expression using an ALTER TABLE request.

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

For information about the Viewpoint Stats Manager, see the *Teradata® Viewpoint User Guide*, B035-2206.

In some cases, dynamic all-AMP (see [Sampled Statistics](#)) or dynamic single-AMP sampled statistics (see [Dynamic AMP Sampling](#)) are not accurate enough for the Optimizer to generate an optimal, or even a good, join plan. However, it is often true that statistics collected by sampling small subpopulations of table rows can be as good as those collected using 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 plan cost estimates. In most cases, the creation and propagation of derived statistics by the Optimizer can balance all but the most stale statistics (see [Derived Statistics](#)).

Statistical accuracy is fundamentally important for any 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 plan generated using full-table statistics is expected to be at least as good as a plan generated using any form of sampled statistics. There is a high probability that a plan based on full-table statistics will be better, and sometimes significantly better, than a plan based on any form of sampled statistics.

Automatic Collection and Recollection of Statistics

The THRESHOLD options for the COLLECT STATISTICS statement enable you to set periodic recollection thresholds for recollecting statistics. Importantly, if the thresholds you set are not met, the existing statistics are considered not to be stale, so Vantage does not recollect them even if you submit an explicit COLLECT STATISTICS request. The metadata that Vantage uses to manage the automatic recollection of statistics is stored in several tables in the TDSTATS database.

Several features work with the COLLECT STATISTICS statement to manage the statistics that you collect, including the STATSUSAGE and USECOUNT options for the BEGIN QUERY LOGGING and REPLACE

QUERY LOGGING statements and a number of APIs that can analyze database objects to identify situations where statistics management can be improved by the system. For documentation of these APIs, see *Teradata Vantage™ - Application Programming Reference*, B035-1090.

Relative Benefits of Collecting Full-Table and 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 dynamic AMP sample is not only based on an even smaller, more cardinality- and skew-sensitive sample of the domain, it also estimates fewer statistics.

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

The optimal compromise is provided by collecting object use and UDI counts (see [Object Use and UDI Counts](#)). The Optimizer can make a very accurate estimate of current table cardinality by adding the current INSERT count to the current cardinality estimate from residual collected statistics and then subtracting the current DELETE count from that sum.

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

Level	Type of Optimization	Considerations
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?

The following table compares the various characteristics of the three methods of collecting statistics and documents their respective most productive uses:

Method	Characteristics	Best Use
Full statistics	<ul style="list-style-type: none"> Collects all statistics for the data. Time consuming. Most accurate of the three methods of collecting statistics. Stored in interval histograms in the Data Dictionary. 	<ul style="list-style-type: none"> Best choice for columns or indexes with highly skewed data values. Recommended for tables with fewer than 1,000 rows per AMP. Recommended for selection columns having a moderate to low number of distinct values.

Method	Characteristics	Best Use
		<ul style="list-style-type: none"> Recommended for most NUSIs, PARTITION columns, and other selection columns because collection time on NUSIs is very fast. Recommended for all column sets or indexes where full statistics add value, and where sampling does not provide satisfactory statistical estimates.
Sampled statistics	<ul style="list-style-type: none"> Collects all statistics for the data, but not by accessing all rows in the table. Significantly faster collection time than full statistics. Stored in interval histograms in the Data Dictionary. 	<ul style="list-style-type: none"> Acceptable for columns or indexes that are highly singular; meaning that their number of distinct values approaches the cardinality of the table. Recommended for unique columns, unique indexes, and for columns or indexes that are highly singular. Experience suggests that sampled statistics are useful for very large tables; meaning tables with tens of billions of rows. Not recommended for tables whose cardinality is less than 1000 times the number of AMPs in the system.
Dynamic AMP sample	<ul style="list-style-type: none"> Estimates fewer statistics than COLLECT STATISTICS does. Statistics estimated include Cardinalities and Average rows per value for all columns. For nonunique secondary indexes only, the following additional statistics are estimated: Average rows per index, Average size of the index per AMP, and Number of distinct values. Extremely fast collection time, so is not detectable. Stored in the file system data block descriptor for the table, not in interval histograms in the Data Dictionary. Occurs automatically. Cannot be invoked by user. Automatically refreshed when batch table INSERT DELETE operations exceed a threshold of 10% of table cardinality. Cardinality is not refreshed by individual INSERT or DELETE requests even if the sum of their updates exceed the 10% threshold. Cached with the data block descriptor. Not used for non-indexed selection criteria or indexed selection with non-equality conditions. 	<ul style="list-style-type: none"> Good for cardinality estimates when there is little or no skew and the table has significantly more rows than the number of AMPs in the system. Collects reliable statistics for NUSI columns when there is limited skew and the table has significantly more rows than the number of AMPs in the system. Useful as a temporary fallback measure for columns and indexes on which you have not yet decided whether to collect statistics or not. Dynamic AMP sampling provides a reasonable fallback mechanism for supporting the optimization of newly devised ad hoc queries until you understand where collected statistics are needed to support query plans for them. Vantage stores cardinality estimates from dynamic AMP samples in the interval histogram for estimating table growth even when complete, fresh statistics are available.

To avoid recollecting statistics when the existing statistics are still fresh, you should use one or more of the THRESHOLD options that you can specify for a COLLECT STATISTICS request. When you specify a threshold that must be met for statistics on an index or column set to be recollected, you enable Vantage to use various methods to determine whether the statistics it already has have become stale or not. If the existing statistics meet your criteria for freshness, Vantage ignores a request you submit to recollect those statistics and only recollects them when they meet your specified criteria. You can specify similar criteria for recollecting statistics using sampling rather than a full-table scan. The Optimizer uses these directives together with object use count information to determine when unused statistics should be recollected or dropped. See [Object Use and UDI Counts](#) and the information about COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for further information about using thresholds and sampling criteria with your recollections of index and column statistics.

Using Interval Histogram Statistics or a Dynamic AMP Sample for NUSI Subtable Statistics

Because the statistics contained in the interval histograms for a NUSI subtable (see [Interval Histograms](#)) can be so stale that a dynamic AMP sample would produce more accurate cardinality estimates, the Optimizer bases its selection of which statistics to use based on the following set of rules.

- The number of NUSI subtable index rows in an AMP is the number of distinct values in the index columns.

The Optimizer assumes that all of the distinct values it samples from a single AMP exist on all of the AMPs in the system.

- If a NUSI has an interval histogram, and if that histogram has fewer distinct values than a dynamic AMP sample gathers, then the system increases the number of index values to the number of values obtained from the dynamic AMP sample, with the exception of the primary index.

This counters the problem of stale NUSI statistics.

- The system sets the number of distinct values for unique indexes to the table cardinality by overriding the values from either the interval histogram or from a dynamic AMP sample.

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

Note:

You must collect or refresh NUSI statistics for the following situations:

- Tables have a high correlation between the primary index and NUSI column sets.
- Workloads contain frequently submitted queries with single-table predicates.

The Optimizer needs interval histograms to make accurate single-table cardinality estimates.

Time and Resource Consumption 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 possibly result in less optimal query plans.

You cannot collect sampled single-column PARTITION statistics. The system allows you to submit such a request, but does not honor it. Instead, Vantage sets the sampling percentage to 100%. Sampled collection is permitted for multicolumn PARTITION statistics.

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

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

If you collect statistics on multiple columns and indexes of the same table, the process can easily take two to four times longer hours to complete, compared to collecting statistics on a single column. When you add the time required to collect statistics for numerous smaller tables in the database to the mix, the time required to collect statistics for all the tables can be surprisingly large. You might even decide that the necessary time for collection is excessive for your production environment, particularly if you have a narrow time window for collecting statistics.

Collecting full-table statistics is not just time consuming; it also places a performance burden on the system, consuming CPU and disk I/O resources that would otherwise be devoted to query processing and other application workloads. 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.

Because collecting and recollecting statistics can be so time consuming, you should consider specifying one or more of the sampling or threshold options (or both) when you first collect statistics on a table.

These options enable the Optimizer to decide when statistics have become stale rather than forcing a database administrator into the position of making the determination. When these collection options are in place, the Optimizer does not honor requests to recollect statistics that are not stale, so you can submit COLLECT STATISTICS more frequently, knowing that if the Optimizer does not need to freshen the statistics it requires to optimize a request, it does not recollect those statistics needlessly. See the information about COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information about the USING clause sampling and threshold options.

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

The optimal way to determine whether statistics that have been collected on a database object are stale is to use the THRESHOLD logic of the COLLECT STATISTICS (Optimizer Form) statement (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

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

Consider two tables, A and B:

Table Name	Statistics Collected?	Cardinality When Statistics Were Collected	Current Cardinality
A	Yes	1,000	1,000,000
B	No	unknown	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 75,000 rows of table B). As a result, the query runs much longer than necessary.

There are 2 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 an index or a column of a table for which statistics have been collected has changed significantly.

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

The best way to handle this is to enable the Optimizer to determine when statistics are stale by specifying one or more threshold options when you collect or recollect statistics on a table or single-table view.

Collecting and recollecting statistics using one or more of the threshold options makes it possible for the Optimizer to reject recollecting statistics when it determines that the current statistics are not stale.

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

```
HELP STATISTICS table_name;
```

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 table_name;
```

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 table_name;
```

Teradata Viewpoint Stats Manager

The Stats Manager portlet allows you to manage statistics collection, which includes the ability to collect and analyze statistics, create and control jobs, and manage recommendations.

You can use Stats Manager to perform the following tasks:

- View statistics on your system
- Identify statistics to be collected and schedule statistic collection
- Identify and collect missing statistics
- Detect and refresh stale statistics
- Identify and discontinue collecting unused statistics
- View when statistics were last collected and are scheduled for collection again

See *Teradata® Viewpoint User Guide*, B035-2206 for information about the uses and capabilities of this portlet.

How the AMP Software Collects Statistics

When you collect statistics and demographics, the AMP software creates statistics rows in *DBC.StatsTbl* that stores various statistics and demographics that summarize the table columns specified by the COLLECT STATISTICS statement. These rows come in the following basic types:

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

The information taken from these rows is used to populate the statistical interval histograms used by the Optimizer to make its initial cardinality estimates (see [Interval Histograms](#)).

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

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

You can report these statistics using a HELP STATISTICS request (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144). HELP STATISTICS reports only summary statistics. To report detailed statistics, use a SHOW STATISTICS request.

Null and All-Null Statistics and Demographics

Null and all-null statistics provide the following cardinality information:

Statistic	Definition
NumNulls	Number of rows in the column set for which one or more of the columns are null.
NumAllNulls	Number of rows in a composite column set for which all of the columns are null.

Vantage only collects the NumAllNulls statistic for composite column sets when all of the columns in the composite set are null. In other words, the only time it is possible to collect a NumAllNulls value is when you collect multicolumn statistics or collect statistics on a composite index, and none of the columns in that set have a value.

Consider the following example of what is meant by null and all-null rows. Suppose you create a 3-column USI for table `t_ex` on columns `b`, `c`, and `d`. When you collect statistics on the USI, one row is found in which all 3 columns making up the USI are null. This would be reported as an *all-nulls* instance in the statistics. This example is degenerate because there could never be more than one row having a USI with all null columns in a table, but it is useful as an example, even though it does not generalize.

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

t_ex					
a	b	c	d	e	f
PI	USI				
355	?	?	?	25000.00	375000.00

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

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

All of the following rows in `t_ex` would be reported as *null fields*, because all have at least one null in one of the USI columns, while only the first row would be reported for the all-null fields statistic.

t_ex					
a	b	c	d	e	f
PI	USI				
355	?	?	?	25000.00	375000.00

t_ex						
a	b	c	d	e	f	
PI	USI					
685	?	155	?	45000.00	495000.00	<-- null fields (orange)
900	325	?	?	32000.00	400000.00	
275	?	?	760	55000.00	575000.00	
597	?	254	893	10000.00	150000.00	
322	679	?	204	75000.00	650000.00	
781	891	357	?	32000.00	400000.00	

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

For example, suppose you have table t1 with columns x1 and y1, and values for x1 and y1.

t1				
x1	y1			
10	10			
20	?		<-- null fields (orange)	
?	30			
?	?		<-- all-null fields (tan)	
?	?			

Again, the all nulls composite fields are shaded orange and the partly null composite fields are shaded green.

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

For example, if you could collect only the NumNulls statistic, the histogram on (x1, y1) would indicate that the number of nulls is 4, the number of unique values is 2, and the total number of rows is 5. If columns x and y are used as join columns, the Optimizer would evaluate the join costing by incorrectly assuming there

are only 2 unique values in the data when there are actually 4. This can cause problems in scenarios such as redistribution costs, skew detection, and the like.

To circumvent this problem, Vantage computes the true number of distinct partial nulls and saves them in the histogram as NumPNulls.

The following equation calculates the true distinct values for partial nulls:

$$\text{NumUniqueValues} = \text{NumValues} + \text{NumPNullValues} + 1$$

Equation Element	Description
NumUniqueValues	the number of true distinct values for partially null rows.
NumValues	the number of non-distinct values for partially null rows.
NumPNullValues	the number of true distinct values among the partially null rows.
1	an adjustment for the estimation error when the cardinality of all nulls > 0.

To make this computation, Vantage injects a marker and uses the following aggregation query to collect statistics on (x1,y1). For the following request, the fieldID for null_marker is set to 2 and the fieldID for cnt is set to 3:

```
SELECT CASE
    WHEN x1 IS NULL
    AND y1 IS NULL
    THEN 1
    ELSE IF x1 IS NULL
    OR
        y1 IS NULL
    THEN 2
    ELSE 0
    END AS null_marker ,x1, y1, COUNT(*) AS cnt
FROM t_coll_stats
GROUP BY 1, 2;
```

Sampled Statistics

To collect sampled statistics, Vantage generates a retrieve request to do the sampling. It then passes the sampled rows to the subsequent aggregation step. The retrieve request can use TOP*n* operations to retrieve the rows if the table is hash distributed and either hard ordered or a NoPI table. For PPI tables that are sorted on their RowKey value, Vantage can generate a true sample step.

PARTITION Statistics

The aggregation step builds a detailed row with the partition number and the cardinality for queries such as the following:

```
SELECT PARTITION, COUNT(*)
FROM t_coll_stats;
```

UDF Statistics

You can collect statistics on deterministic UDFs, but not on nondeterministic UDFs.

Consider the following deterministic UDF:

```
CREATE FUNCTION months_between(date1 TIMESTAMP, date2 TIMESTAMP)
RETURNS FLOAT
LANGUAGE C
NO SQL
DETERMINISTIC
SPECIFIC months_between_tt
EXTERNAL NAME 'CS!months_between_tt!$PGMPATH$/months_between_tt.c'
PARAMETER STYLE SQL;
```

The following statement collects statistics on UDF `months_between()`:

```
COLLECT STATISTICS
COLUMN months_between(BEGIN(policy_duration),
                      END(policy_duration)) AS
                      Stats_MthsBetweenBegAndEnd
ON policy_types;
```

The crucial issue for UDF statistics is not collecting them, but dropping them. If you drop a deterministic UDF after you have collected statistics on it, those statistics become unusable. Because UDFs do not belong to any table or column, it is very expensive to drop all the statistics collected on them. Instead, Vantage uses the following lazy approach to dropping statistics that have been collected on deterministic UDFs:

- When Vantage loads statistics header information from the `DBC.StatsTbl` for a table to handle a user query or another statistics-related statement, the system detects nonvalid expressions at the time the query is resolved.

For statistics collected on an unresolvable expression, the system updates the `validStats` field to `FALSE` for the corresponding statistics in `DBC.StatsTbl`.

- When retrieving histograms, Vantage only retrieves those for valid statistics. This minimizes the overhead of parsing nonvalid statistics during query processing time.
- The `SHOW STATISTICS` and `HELP STATISTICS` statements report only valid statistics.

- A DROP STATISTICS request on a table forces Vantage to drop all statistics, both valid and nonvalid.
- A SHOW STATISTICS COLUMN request returns a warning message if it is made on nonvalid statistics.
- A DROP STATISTICS COLUMN request returns a warning message if it is made on nonvalid statistics.
- Recollecting statistics on a UDF revalidates the statistics if the UDF exists at the time the statistics are recollected.
- You can query the DBC.StatsV system view to determine nonvalid statistics by checking the validStats column

You cannot drop individual nonvalid statistics.

To drop nonvalid statistics, you must first drop all statistics on the table and then recollect valid statistics.

- If you drop a UDF and then recreate it to return a different type, Vantage ignores the existing statistics for that function during optimization.

Average AMP Rows Per Value Statistic

The average AMP rows per value statistic is an exact system-wide average of the cardinality per value for each individual AMP over the number of rows per value for a NUSI column set on which statistics have been collected. This statistic is computed only for NUSI columns and is used for nested join costing.

Example EXPLAIN Text for First Time Statistics Collection

The following EXPLAIN text reports the steps taken to build a histogram and collect statistics for the first time on column x1 from table t1 with embedded comments to clarify exactly what happens at critical steps in the process.

```
EXPLAIN COLLECT STATISTICS COLUMN x1 ON t1;
```

Explanation

- ```

```
- 1) First, we lock QSTATS.t1 for access.
  - 2) Next, we do an all-AMPs SUM step to aggregate from DF2.t1 by way of an all-rows scan with no residual conditions, grouping by field1 ( RK.t1.x1). Aggregate Intermediate Results are computed locally, then placed in Spool 9. The size of Spool 9 is estimated with low confidence to be 2 rows (44 bytes). The estimated time for this step is 0.03 seconds.
  - 3) We do an all-AMPs RETRIEVE step from Spool 9 (Last Use) by way of an all-rows scan into Spool 5 (all\_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with low confidence to be 2 rows (52 bytes). The estimated time for this step is 0.04 seconds.
  - 4) Then we save the UPDATED STATISTICS from Spool 5 (Last Use) into Spool 3, which is built locally on the AMP derived from DBC.StatsTbl by way of the primary index. /\*Hash of t1\*/

```

5) We lock DBC.StatsTbl for write on a RowHash.
 /*Raise a retryable error if the generated stats id is already
 used*/
 /*by some other collect stats on the same table.*/
 /*The error step can also be done by USI on (TableId, StatsId)*/
6) We do a single-AMP ABORT test from DBC.StatsTbl by way of the
 primary index "DBC.StatsTbl.TableId = <t1 Id> with a residual
 condition of ("DBC.StatsTbl.StatsId = <generated stats id>").
 /*Insert the histogram row with generated stats id*/
7) We do a single-AMP MERGE into DBC.StatsTbl from Spool 3 (Last
 Use). The size is estimated with low confidence to be 1 row. The
 estimated time for this step is 0.31 seconds.
 /*Update or insert the master record with the updated table level
 demographics*/
8) We do a single-AMP UPDATE from Spool 3 by way of the primary
 index "DBC.StatsTbl.TableId = <t1 Id>" with a residual condition
 of DBC.StatsTbl.StatsId = 0. The size is
 estimated with low confidence to be 1 row. The estimated time for
 this step is 0.02 seconds. If the row cannot be found, then we do
 an INSERT into DBC.StatsTbl.
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.

```

Note the following additional aspects of this EXPLAIN text:

- In step 7, Vantage inserts the histogram row with the generated StatsId.
- In step 8, Vantage updates the master record with new table-level demographics with a fixed StatsId of 0.
- Note that master records always have a fixed StatsId of 0.

### Example EXPLAIN Text for Recollecting Statistics

The following EXPLAIN text reports the steps taken to recollect statistics on column x1 from table t1 with embedded comments to clarify exactly what happens at critical steps in the process:

```
EXPLAIN COLLECT STATISTICS COLUMN x1 ON t1;
```

#### Explanation

- ```

-----
1) First, we lock QSTATS.t1 for access.
2) Next, we do an all-AMPs SUM step to aggregate from RK.t1 by way of
   an all-rows scan with no residual conditions,
   grouping by field1 ( RK.t1.x1). Aggregate Intermediate Results

```

are computed locally, then placed in Spool 9. The size of Spool 9 is estimated with low confidence to be 2 rows (44 bytes). The estimated time for this step is 0.03 seconds.

- 3) We do an all-AMPs RETRIEVE step from Spool 9 (Last Use) by way of an all-rows scan into Spool 5 (all_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with low confidence to be 2 rows (52 bytes). The estimated time for this step is 0.04 seconds.
 - 4) Then we save the UPDATED STATISTICS from Spool 1 (Last Use) into Spool 3, which is built locally on the AMP derived from DBC.StatsTbl by way of the primary index.
 - 5) We lock DBC.StatsTbl for write on a RowHash. /*Hash of (t1)*/
 - 6) We do a single-AMP UPDATE from Spool 3 by way of the primary index "DBC.StatsTbl.TableId = <t1 Id>" with a residual condition of DBC.StatsTbl.StatsId = <existing stats id>. The size is estimated with low confidence to be 1 row. The estimated time for this step is 0.02 seconds. If the row cannot be found, then we do an INSERT into DBC.StatsTbl.
 - 7) We do a single-AMP UPDATE from Spool 3 by way of the primary index "DBC.StatsTbl.TableId = <t1 Id>" with a residual condition of DBC.StatsTbl.StatsId = 0. The size is estimated with low confidence to be 1 row. The estimated time for this step is 0.02 seconds. If the row cannot be found, then we do an INSERT into DBC.StatsTbl.
 - 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.

Note the following additional aspects of this EXPLAIN text:

- In step 6, Vantage updates the histogram row with the existing StatsId.
- In step 7, Vantage updates the master record with new table-level demographics with a fixed StatsId of 0.
- Master records always have a fixed statistics ID of 0.

Interval Histograms

A histogram is a count of the number of occurrences, or *cardinality*, of a particular category of data that fall into defined disjunct value range categories (referred to as *intervals*).

Interval Histograms

Vantage uses equal-height, high-biased, and history interval histograms to represent the cardinalities and other statistical values and demographics of columns and indexes for all-AMPs sampled statistics and for

full-table statistics. The greater the number of intervals in a histogram, the more accurately it can describe the distribution of data by characterizing a smaller percentage of its composition per each interval.

Vantage determines the number of high-biased, equal-height intervals; history records that can be accommodated using a BLOB with maximum size 64 KB.

You can determine the maximum number of intervals (between 10 and 500) used for a histogram by specifying the `MAXINTERVALS USING` option when you collect statistics. The default maximum is 250.

The number of intervals used to store statistics is a function of the number of distinct values in the column set represented. For example, if there are only 10 unique values in a column or index set, Vantage does not store the statistics for that column across 500 intervals, but across 11 (the master record plus the 10 additional intervals containing the frequencies for the distinct values in the set).

Vantage employs the maximum number of intervals only when the number of distinct values in the column or index set for which statistics are being captured equals or exceeds the maximum number of intervals.

The statistical and demographic information maintained in the histograms is used to estimate various attributes of a query, most importantly the cardinalities of various aspects of the relations that are specified in the request.

Note that Vantage also uses derived statistics to estimate cardinalities and selectivities. For more information, [Derived Statistics](#). These statistics are based on the initial values stored in the interval histograms, but are then adjusted for accuracy at each stage of query optimization by incorporating additional information such as CHECK and referential integrity constraints, query predicates, and hash and join indexes.

See [Derived Statistics](#) for details.

Freshness of Interval Histogram Statistics

The statistics contained in interval histograms are, by their very nature, out of date as soon as the data changes.

The Teradata query optimizer uses derived statistics to reduce the worst-case error of estimating query cardinalities at the various stages of optimization (see [Derived Statistics](#)) and provides several threshold and sampling options for collecting statistics that enable you to avoid recollecting statistics that do not need to be refreshed.

For details, see the information about `COLLECT STATISTICS` (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Interval Histogram Terminology

The terms used to describe the intervals and histograms are defined in the following table.

Term	Definition
Cardinality	The number of rows per AMP that satisfy a predicate condition. Note that in this context, cardinality generally is not the number of rows in the entire table, but the number of rows that qualify for a predicate condition. See Using Interval Histograms

Term	Definition
	to Make Initial Cardinality Estimates for a fuller explanation of what cardinality means in this context.
Compressed interval histogram	A histogram that combines high-biased intervals and equal-height intervals or high-biased intervals only, or both, with a master record.
Equal-height interval	<p>An interval containing column statistics normalized across the distribution in such a way that the graph of the distribution of the number of rows as a function of interval number is flat.</p> <p>This is achieved by varying the width of each interval so it contains approximately the same number of rows (but with different attribute value ranges) as its neighbors. An equal-height interval does not include loner values or NULLs and their frequencies.</p> <p>Equal-height intervals are also known as equal-depth intervals.</p> <p>The definitions for some of the fields in an equal-height are indicated by the following list.</p> <ul style="list-style-type: none"> • Values is the total number of non-mode values in the interval. <p>When an equal-height interval represents a single value, the value is saved as a Mode Value. Vantage sets the number for Values, which is the total number of non-mode values, to 0 in this case.</p> <ul style="list-style-type: none"> • Mode is the most frequent value in the interval. • Mode frequency is the number of rows having the mode value. • Maximum value is the maximum value covered by the interval. • Rows is the total number of rows for the non-mode values in the interval.
Equal-height interval histogram	A histogram characterized by an array of ordered, equal-height intervals. Also known as equal-depth interval histograms.
High-biased interval	<p>An interval used to characterize a non-NULL skewed value (that is, a <i>loner</i>) for a column. Any value that is significantly skewed is summarized by a high-biased interval. See Loner.</p> <p>The definitions for some of the fields in a high-biased interval are indicated by the following list:</p> <ul style="list-style-type: none"> • The loner value in the interval. • The number of rows having the loner value.
High-biased interval histogram	A histogram characterized by an array of ordered, high-biased intervals of loners.
Histogram	<p>A means of representing distributions as a function of the number of elements per a determined interval width.</p> <p>Histograms are often called bar charts. Each bar in a histogram represents the number of rows for the defined interval.</p> <p>In relational query optimization theory, the term is used to describe the rows in a Data Dictionary table or system catalog that store the particular intervals used to characterize the frequency distribution of the attribute values for a column set.</p> <p>All histograms described in this topic are frequency histograms. Each interval in a frequency histogram contains fields representing the number of rows that have the attribute values belonging to its range.</p>

Term	Definition
History record	<p>An interval containing historical information about a histogram including MinValue, MaxValue, HighModeValue, and HighModeFrequency statistics as well as other relevant historical data.</p> <p>When you recollect statistics, Vantage saves the summary information from the previous histogram as a history record in the new histogram. Vantage determines the number of history records a histogram maintains based on a trend analysis. When history records are no longer needed, the system purges them.</p>
Interval	<p>A bounded, non-overlapping set of attribute value frequencies.</p> <p>Sometimes called a bin or bucket.</p>
Loner, or high-biased value	<p>A loner is a non-NULL value with a high frequency; a highly frequent value can indicate significant skew.</p> <p>To be considered a loner, a value must satisfy the following condition:</p> $f \geq \frac{T}{500}$ <p>where:</p> <ul style="list-style-type: none"> • f defines the frequency of the value. • T defines the cardinality of the relation. <p>Based on the maximum size of 64 KB, Vantage determines the number of high-biased intervals and equal-height intervals that can be accommodated in a histogram, while also leaving space for the history records.</p>
Master Record	<p>A row in DBC.StatsTbl with a StatsId column value of 0 that acts as a statistics identifier within each source column or index statistics set.</p> <p>The Optimizer uses the master record for each column or index set to maintain the cardinality, average row size, dynamic AMP sample estimates, and other object-level attributes.</p> <p>Whenever statistics are collected or updated for any column or index set, Vantage updates the master record of the source to reflect the latest object-level statistical information.</p> <p>When statistics are refreshed on a column set, the Optimizer resets the UDI counts. Because different statistics can be collected at different times, the current logs of delete and insert counts are retained with the master record before Vantage resets them. The UPD counts are retained at the level of individual column statistics. The individual column statistics refer to master record delete and insert counts, column statistics-level UPD counts, or both, and the latest system UDI counts from DBC.ObjectUsage to determine the overall effective UDI counts for a given statistics.</p>
Skew	<p>A measure of the asymmetry of the distribution of a set of attribute values or their frequencies.</p> <p>With respect to skew in parallel databases, there are several possible types.</p> <ul style="list-style-type: none"> • <i>Attribute value skew</i> refers to skew that is inherent in the data. An example might be a column that can have only 2 values such as TRUE or FALSE. • <i>Partition skew</i> refers to skew that results from an uneven distribution of data across the AMPs. <p>The difference is apparent from the context. As used in this document, the term usually refers to partition skew.</p> <p>Skew is somewhat more likely to be seen with NoPI because of the way the system distributes their rows or with intermediate results.</p>

Term	Definition
	For information about how the Optimizer deals with skew when performing an equality join on skewed tables, see Strategies for Joining Skewed Tables on an Equality Join Condition .

Types of Interval Histograms

Depending on the distribution of values, which is also referred to as the degree of skew, in a column or index set, any one of four varieties of histogram types can be used to represent its statistics.

- Equal-height interval histogram

Equal-height interval histograms have much lower worst-case and average errors for a wide variety of queries than do equal-width interval histograms.

The statistics for a column set are expressed as an equal-height interval histogram if none of the frequencies of its values are skewed.

Vantage determines the number of high-biased intervals and equal-height intervals that can be accommodated while also leaving space for history records.

In an equal-height interval histogram each interval represents approximately the same number of rows having that range of values, making their ordinate, cardinality, an approximately constant value or height. If a row has a value for a column that is already represented in an interval, then it is counted in that interval, so some intervals may represent more rows than others.

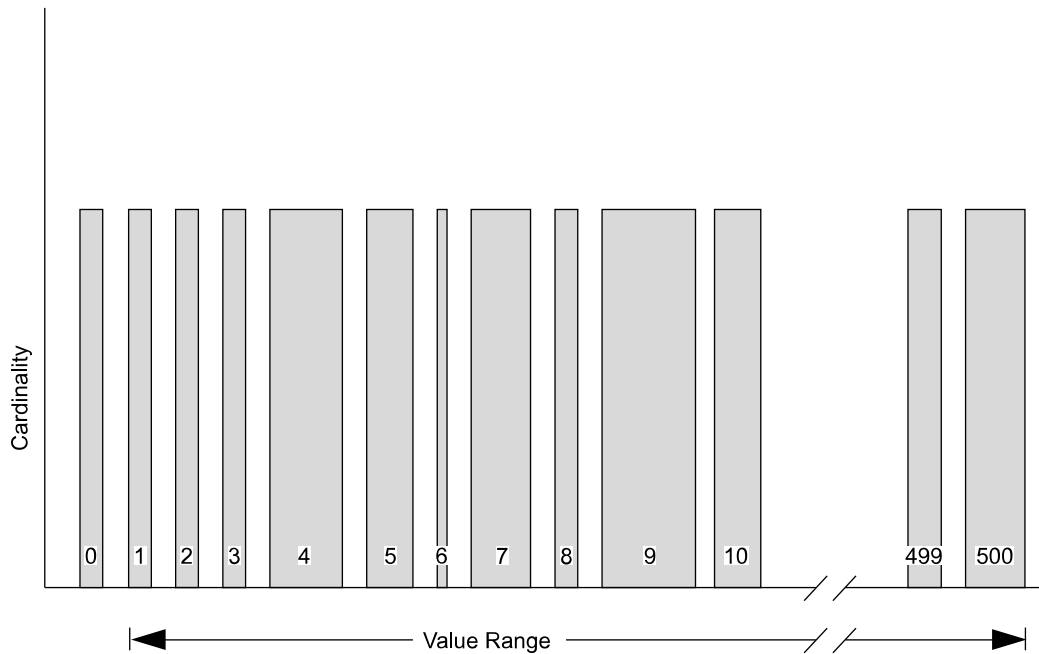
For example, suppose there are approximately 2.5 million rows in a table. Then there would be approximately 10,000 rows per interval, assuming a 250 interval histogram.

Suppose that for a given interval, Vantage processes the values for 9,900 rows and the next value is present in 300 rows. Those rows would be counted in this interval, and the cardinality for the interval would 10,200. Alternatively, the rows could be counted in the next interval, so this interval would only represent 9,900 rows.

A COLLECT STATISTICS request divides the rows in ranges of values such that each range has approximately the same number of rows, but it never splits rows with the same value across intervals. To achieve constant interval cardinalities, the interval widths, or value ranges, must vary.

If a histogram contains 250 equal-height intervals, each interval effectively represents 0.40% for the population of attribute values it represents.

The following graphic illustrates the concept of an equal-height interval histogram. Note that the number of rows per equal-height interval is only approximately equal in practice, so the graphic is slightly misleading with respect to the precise equality of heights for all equal-height intervals in the histogram.



- High-biased interval histogram

High-biased intervals are used to represent a column or index set only when there is significant skew in the frequency distribution of its values. The Optimizer does not maintain pure high-biased interval histograms. Instead, it mixes high-biased intervals with equal-height intervals in a compressed histogram whenever some values for a column or index set are skewed significantly.

In a high-biased interval histogram, each high-biased interval contains only one value and its frequency.

- Compressed histogram

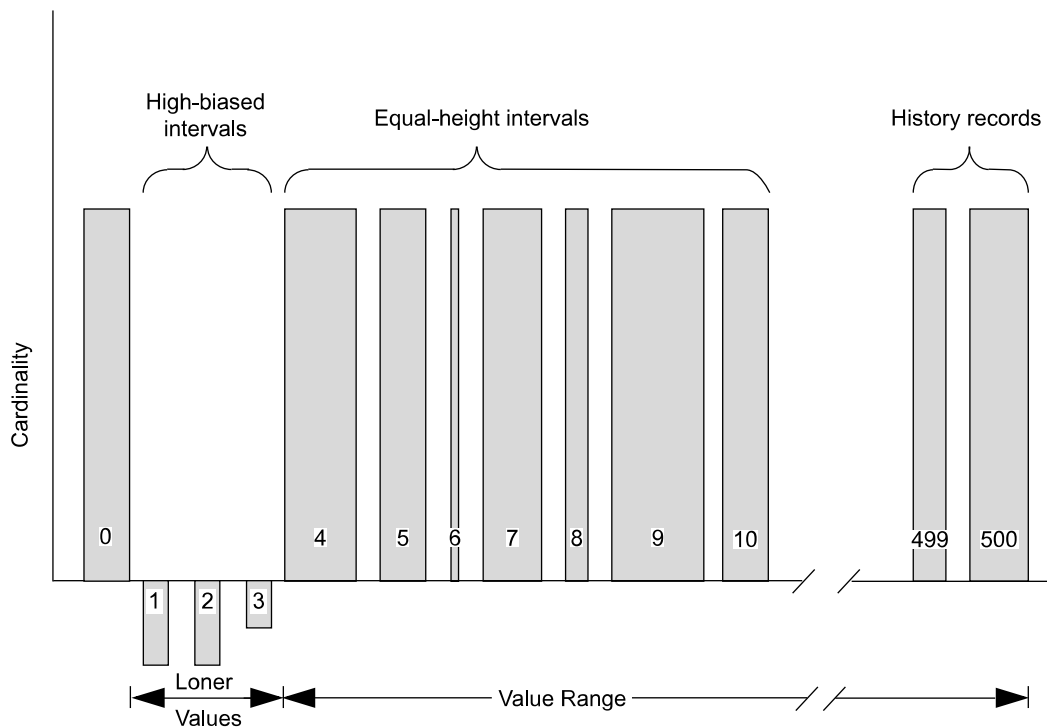
Compressed histograms contain a mix of equal- and high-biased intervals plus a master record.

Compressed histograms are an improvement over pure equal-height histograms because they provide exact statistics for the largest values in the data. This is useful for join estimation, for example, which compares the largest values in relations to be joined.

Any high-biased intervals always precede equal-height intervals in a compressed histogram.

The following graphic illustrates the concept of a compressed histogram with 201 intervals. Note that the number of rows per equal-height interval is only approximately equal in practice, so the graphic is slightly misleading with respect to the precise equality of heights for all equal-height intervals in the compressed histogram.

In this example of a compressed interval histogram, the first three intervals define three loner values and the last several define history intervals.



- History record

History records archive a chronicle of the histogram. When you recollect statistics, Vantage saves the summary information from the previous histogram as a history interval in the new histogram. The system determines the number of history intervals based on a trend analysis. Vantage purges the history intervals that are no longer needed.

A history interval contains MinValue, MaxValue, HighModeValue, HighMode frequency and some additional data.

The number of history intervals maintained for each histogram is based on the following three criteria:

- The significance of the history interval.

The determination of number of history records to retain or purge is determined by their significance.

The significance of a history interval depends on the significance of the interval's participation in the most recent trend analysis. If the significance of a history interval is less than a user- or system-defined threshold, Vantage purges it from the histogram.

The significance of a history record is calculated based on the significances of statistical values such as NumRow, UV, AllNull, NumNull, MinVal, and MaxVal from the history interval. The significance of a single statistical value is determined by the following rules.

- If Vantage does not use the statistical value in recent trending analyses, its significance is 0.
- If the statistical value does participate in recent trending analyses, its significance is proportional to an absolute difference of the reliabilities with and without the statistical value.

- The minimum number of history records, which Vantage sets to 10 by default, indicates the minimum number of history intervals to retain that is determined to be significant. This default is chosen because a minimum of 10 history intervals is required to achieve the necessary reliability factor of a value greater than 0.9.
- The maximum number of history records, which Vantage sets to 20 by default, indicates the maximum number of history intervals that are determined to be significant. This default is chosen to be twice the minimum value, which should be sufficient for any trend analysis.

Content and Storage of Histograms

Index and column histograms are stored in the *Histogram* column of *DBC.StatsTbl*. This column has a BLOB data type.

Rows for each histogram are separated into the following categories:

- Nulls
- Loners
- Non-loners

Recall that a histogram can be composed of either of the following types of intervals in addition to a master record:

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

The system stores histogram values in a format that is similar to a row structure. All the variable data is stored toward the end of the histogram, and their corresponding offsets are saved in the interval buckets. Data value sizes are constant in the histogram buckets because of the offset usage for variable-length columns. This enables the histogram buckets to have a fixed size, which makes it possible to randomly position to any bucket.

A value descriptor which is saved in the histogram header is used to encode and decode the data values of the histogram.

- Summary statistics are recorded in the histogram header.

Use the HELP STATISTICS statement to report the summary statistics for a histogram.

- High-biased intervals.
- Equal-height intervals.
- History intervals.

Use the SHOW STATISTICS statement to report the detailed statistics for a histogram.

Vantage uses a value descriptor from the histogram header to encode and decode the data values of the histogram. If the histogram is built on multiple columns, Vantagemaintains it as an array with each component element describing a single column. The array uses a single element to store single-column statistics and multiple elements to store multicolumn statistics. If an array element is a variable-length data type, Vantage stores a 4-byte offset in the interval for quick access. For fixed length types, the actual data is saved in the intervals. All the variable offsets are saved first followed by the fixed type data.

The histogram header contains all of the summary information and the information to encode and decode the interval buckets. The high-level structure of a histogram bucket is as follows, with data being stored in the order indicated.

1. Biased values [(value_1), (value_2) ...]
2. Biased value frequencies []
3. Equal height values [(Max, Mode) ...]
4. Equal height bucket counts [(modal frequency, other Vals, other rows) ...]
5. History record values [(Min, Max, HighMode, HighModeFreq) ...]
6. History record bucket counts [(OptHistoryIntervalDataType) ...]
7. Data values for variable-length columns.

Vantage stores the corresponding offsets to variable-length columns in the histogram buckets.

There are 3 types of histograms in an interval histogram: high-biased, equal-height, and history. There is also the compressed histogram type, which mixes equal-height and high-biased intervals.

The offset to each bucket in the histogram is computed from its base offset, which is saved in the histogram header.

- High-biased intervals

High-biased intervals are used to represent a column or index set only when there is significant skew in the frequency distribution of its values.

Vantage saves the loner values and their corresponding frequencies for high-biased intervals in the following general structure.

Statistic	Description
Loner value	The loner value stored in the interval.
Loner value frequency	The number of rows that have this loner value.

- Equal-height intervals

Each equal-height interval contains a mode value for the interval and its frequency, the number of values other than the modal value in the interval, the number of other rows in the interval other than those having the mode frequency, and the lowest frequency among the Other values. Vantage uses the lowest frequency to determine the standard deviation in various estimates.

The mode value is saved based on the value descriptor. The frequency and other values/rows are saved in the following general structure.

Statistic	Description
Mode frequency	The frequency of occurrence of the modal numeric value in the interval.
Other values	The cardinality of non-modal values in the interval.
Other rows	The cardinality of rows other than those having the modal frequency.

Statistic	Description
Lowest frequency	The lowest frequency of other (non-modal) values in the interval.

- Compressed histogram

Compressed histograms contain a mix of equal-height and high-biased intervals. Compressed histograms are an improvement over pure equal-height histograms because they provide exact statistics for the values with the largest frequencies in the data. This is useful for join estimation, for example, which compares the largest values in relations to be joined.

- History records

This interval type records the history of the histogram. When you recollect statistics, Vantage saves the summary information from the previous histogram as a history interval in the histogram.

Vantage determines how many history intervals to keep based on a linear trend analysis, and purges history records when they are no longer needed.

The general structure of a history record is as follows:

Statistic	Description
ANSI timestamp	Statistics collection timestamp.
Number of high-biased (loners) values	Number of high-biased values (loners) in the histogram.
Number of equal-height intervals	Number of equal-height intervals in the histogram.
Usage type	Description of whether the collected statistics are summary or detailed.
Sampling percentage	If the collected statistics were sampled, states the sampling percentage used.
Sampling decision by	If sampling was used to collect the statistics, states whether the sampling was system-determined or user-determined.
Equal-height interval deviations	Deviation of value frequencies within the equal-height intervals in the histogram.
Number of nulls	Number of partly null rows in the histogram.
Number of all nulls	Number of all null rows in the histogram.
High mode frequency	Frequency of occurrence of the highest mode numeric value in the histogram.
Number of values	Number of distinct values in the histogram.
Number of rows	Number of rows in the histogram.
CPU cost	States how much CPU time was consumed collecting the statistics for the histogram.

Statistic	Description
I/O cost	States how much input/output was consumed collecting the statistics for the histogram.
None	Reserved for future use.
None	Reserved for future use.

Data Types of Stored Statistics

Vantage stores all statistics using the native data type for the column. Because of this, there are no problems with loss of precision, truncation, or with oversized histograms caused by the data types of the values being stored.

Sampled Statistics

Reasons to Use Sampled Statistics Instead of Full Statistics

There are occasions when it becomes difficult to expend the effort required to collect full-table statistics. For example, suppose you have a multibillion row table that requires several hours collection time, and a collection window that is too narrow to permit the operation. If you encounter situations like this when it is not possible to collect full-table statistics in a reasonable amount of time, you can opt to collect statistics on a sample of table rows instead.

You can specify several different statistics sampling mechanisms with the USING option of a COLLECT STATISTICS request, but you cannot collect sampled statistics with a SAMPLE USING option if a column that you specify is a component of the partitioning expression of a row-partitioned table or column-partitioned table.

You can collect statistics on such a column using the COLLECT STATISTICS USING option if the specified column is both a member of the partitioning expression column set and a member of an index column set.

Sampled statistics are different from dynamic AMP samples in that you can specify the percentage of rows you want to sample explicitly in a COLLECT STATISTICS (Optimizer Form) request to collect sampled statistics, while the number of AMPs from which dynamic AMP samples are collected and the time when those samples are collected is determined by Vantage. You can also specify that Vantage should determine the sampling percentage of COLLECT STATISTICS samples. Furthermore, sampled statistics produce a full set of collected statistics, while dynamic AMP samples collect only a subset of the statistics that are stored in interval histograms (see [Dynamic AMP Sampling](#) for details).

Statistical sampling is known to be a reliable method of gathering approximate statistical estimates when the appropriate preconditions are met.

NOTICE

The quality of the statistics collected with full-table sampling is not guaranteed to be as good as the quality of statistics collected on an entire table without sampling. Do not think of sampled statistics as an alternative to collecting full-table statistics, but as an alternative to never, or rarely, collecting statistics.

When you use sampled statistics rather than full-table statistics, you are trading time in exchange for what are likely to be less accurate statistics. The underlying premise for using sampled statistics is usually that sampled statistics are better than no statistics.

Comparing the Accuracy of Sampled Statistics and Dynamic AMP Samples

Do not confuse statistical sampling with the dynamic 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 dynamic AMP samples.

The following table describes some of the differences between the 2 methods of collecting statistics:

Statistical Sampling	Dynamic AMP Sampling
<p>Collects statistics on a small sample of rows from all AMPs.</p> <p>If the columns are not indexed, then the rows are organized randomly on each AMP, so Vantage just scans the first <i>n</i> percent of rows it finds, where the value of <i>n</i> 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.</p> <p>If the columns are indexed, then more sophisticated sampling is performed to take advantage of the hash-sequenced row ordering.</p>	<p>Collects statistics on a small sample of rows from a single AMP.</p> <p>This is the system default.</p> <p>You can change the number of AMPs from which a dynamic AMP sample is taken by altering the value of an internal DBS Control field. Ask your Teradata Support Center representative for details.</p>
<p>Collects full statistics and stores them in interval histograms in DBC.StatsTblStatsTbl.</p> <p>See also Interval Histograms.</p>	<p>Collects estimates for cardinality, number of distinct index values, and a few other statistics only and stores them in the data block descriptor.</p> <p>Also collects and stores single-AMP and all-AMP cardinality estimates in interval histograms in DBC.StatsTblStatsTbl.</p>
<p>Expands sample size dynamically to compensate for skew.</p>	<p>Sensitive to skew.</p>
<p>Provides fairly accurate estimates of all statistical parameters.</p>	<p>Provides fairly accurate estimates of base table and NUSI cardinality if the following conditions are met.</p> <ul style="list-style-type: none"> • The table is large • The distribution of values is not skewed • The data is not taken from an atypical sample

Statistical Sampling	Dynamic AMP Sampling
	Other standard statistical parameters are less likely to be as accurate.

You cannot sample single-column PARTITION statistics at a level lower than 100%. You can submit a COLLECT STATISTICS request that specifies a lower percentage without the request aborting and without receiving an error message, but Vantage does not honor the specified percentage, and it automatically changes the sampling percentage to 100 internally. For further detail, see the information about COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Recollecting Sampled Statistics

When you recollect statistics on a column or index, whether for the Optimizer or for a QCD analysis, Vantage uses the same method of collection to recollect the statistics as was used for the initial collection, either full-table scan or sampling. You can control the default sampling percentage for recollecting statistics using the DBS Control field SysSampleOption. For more information about DBS Control fields, see *Teradata Vantage™ - Database Utilities*, B035-1102.

The only exception to this is the case where you specify USING SYSTEM SAMPLE when you first collect statistics, but do not specify the FOR CURRENT option for a recollection. The USING SYSTEM SAMPLE option initially collects statistics using a full-table scan, but then downgrades the initial 100% sampling to increasingly lower sampling percentages until it reaches a point where the sampling percentage is not reduced any further.

If you had collected statistics on the column as part of an index on a row-partitioned table, then Vantage follows the general rule and recollects sampled statistics on the index column.

If you would like to change the collection options, you must submit a new COLLECT STATISTICS request with the new collection options fully specified or change the default sampling percentage using the DBS Control field SysSampleOption. This does not apply for sampled COLUMN statistics on a component of the partitioning expression for a row-partitioned table or column-partitioned table, which are not valid, and which you cannot specify in a COLLECT STATISTICS request. Vantage always collects (and recollects) statistics on all the rows in the table for this particular case.

If you want to recollect statistics for a histogram, but override the options that have been saved for that histogram, you can specify the FOR CURRENT option. FOR CURRENT recollects statistics using whatever options you specify for the current COLLECT STATISTICS request only, and reverts to the saved options for subsequent recollections.

You can collect sampled statistics using either a system-determined sampling percentage or a user-specified sampling percentage. For a user-specified sample percentage, Vantage reads the specified percent of rows to collect statistics. All the existing rules apply if the sample percent is explicitly specified.

If you specify the USING SYSTEM SAMPLE option when you collect statistics, the Optimizer determines the appropriate time to switch from collecting full statistics to collecting sampled statistics. With this approach, the Optimizer decides both when to honor sampling and the degree of sampling, up to collecting full

statistics, without sacrificing the quality of the collected demographics. The downgrade process works as follows.

1. When you first submit a request to collect sampled statistics, Vantage collects full statistics, which is equivalent to specifying a 100% sample. This provides the Optimizer with data it can use to determine when to downgrade to a smaller percentage when recollecting statistics.
2. On subsequent requests to recollect statistics, the Optimizer collects full statistics until it determines that it has captured enough historical data to enable it to know when to downgrade the sampling percentage for later recollections.
3. With this historical data, the Optimizer can determine whether a column is skewed, rolling, or static.
4. The Optimizer then considers the column usage (detailed or summary data) it maintains in the `DBC.StatsTbl.UsageType` column and evaluates the number of intervals you have specified to judge when to downgrade from collecting full statistics to collecting sampled statistics.

Vantage is more aggressive about downgrading to sampled statistics for those histograms whose summary data is used, but whose detailed data is not.

5. The Optimizer next determines a sampling percentage to use for recollecting statistics, ranging from 2% to 100%.

It determines the best formula to use to scale the sampling percentage based on the history and nature of the column.

6. In the final stage of the process, Vantage compares the recollected statistics to the column history for quality. The Optimizer only lowers the sampling percentage for subsequent recollections if it knows that a smaller sampling percentage provides sufficiently high quality results.

This process removes the burden of determining which columns can accurately use sampled statistics and which cannot. By collecting full statistics initially, the process provides the Optimizer with the information it needs to make intelligent decisions about how much to sample in subsequent recollections of statistics.

For more information about `COLLECT STATISTICS` (Optimizer Form), see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Dynamic AMP Sampling

Uses of Dynamic AMP Sampling by Vantage

The Optimizer generally relies on statistics collected on primary indexes and row partitions along with UDI counts to make cardinality estimates rather than using dynamic AMP samples because the data provided by UDI counts tends to be accurate than that obtained from dynamic AMP samples. However, there continue to be cases where dynamic AMP sampling has to be relied upon to make cardinality estimates.

When there are no statistics available to quantify the demographics of a column set or index, the Optimizer can select a single AMP to sample dynamically for statistics using an algorithm based on the table ID.

By inference, these numbers are then assumed to represent the global statistics for the column or index.

The cardinality estimates collected by a dynamic AMP sample are stored in the *NumRows* column of the file system data block descriptor (DBD) for the table as well as in the *OneAMPsSampleEst* and *AllAMPsSampleEst* fields when collected from the primary index or from PARTITION statistics for a table.

Vantage collects a new dynamic AMP sample each time a given DBD is fetched, but if interval histogram statistics exist for the columns or indexes of interest, they override the statistics collected by this dynamic AMP sample by default.

Note that the statistics collected by a dynamic AMP sample apply to indexed columns only. If you do not collect statistics on non-indexed columns, then the Optimizer uses various situation-specific heuristics to provide arbitrary cardinality estimates.

Do not think of dynamic AMP samples as a substitute for collecting full-table statistics. For example, while it is true that a dynamic AMP sample generally produces reasonable estimates of the cardinality and number of unique values for a column or index set, it does not generally produce a good estimate of the selectivity of a given predicate in a query.

Assumptions Underlying the Method

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

- The sampled table has enough rows that a snapshot of any one AMP in the system accurately characterizes the demographics for the entire table across all AMPs.

An important implication of this assumption is that dynamic AMP samples can be poor estimators of the true population statistics of a table whenever the cardinality of the table is less than the number of AMPs on the system. Because of this, you should always collect statistics on small tables.

- The rows of the sampled table are distributed evenly, without skew.

If the data is skewed, a dynamic AMP sample can provide significantly erroneous demographics. System UDI counts provide the Optimizer with far better cardinality estimates than dynamic AMP sampling can for tables with skewed distributions.

- The rows of the sampled table are not compressed.

If the columns of an index or column set are compressed using multi-value compression, algorithmic compression, block-level compression, or row-compressed hash and join indexes, the demographics returned by a dynamic AMP sample are not reliable because the Optimizer tends to underestimate the cost of reading a compressed table.

Even though dynamic AMP sampling provides reasonable cardinality estimates for many scenarios, it is prone to significant errors for tables compressed using algorithmic compression, block-level compression, column-partitioned tables, and compressed hash and join indexes, and are ultimately unusable.

For cardinality estimates on these sorts of tables, system UDI counts can provide the Optimizer with accurate change data based on the last time their statistics were updated.

- The rows are not from a column-partitioned table.

As noted in the previous bullet, the statistics returned by a dynamic AMP sample of the rows of a column-partitioned table, like those returned from compressed column sets, are not reliable, and system UDI counts provide consistently more accurate cardinality estimates.

When these assumptions are valid, the cardinalities estimated using dynamic AMP sampling are generally fairly accurate. Sampled statistics do not work well with the partitioning columns of row-partitioned tables, however, and should not be used to collect statistics for them.

Estimating Cardinalities From a Single-AMP Dynamic Sample

The process used to gather statistics on an indexed column using a dynamic AMP sample is as follows:

1. Select an AMP to be used to collect a cardinality estimate by hashing on the value of the table ID to generate an AMP number.
2. Read the master index to locate the cylinders containing data for the desired table, column, or index.
3. Count the number of cylinders that contain the desired data.
4. Randomly select one of those cylinders and read its cylinder index to locate the data blocks containing data for the desired table, column, or index.
5. Count the number of data blocks in the cylinder that contain the desired data.
6. Randomly select one of those data blocks and count the number of rows it contains.
7. Compute an estimate of the cardinality of the table using the following equation:

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 dynamic AMP sample.

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

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.

As a result, the potential for error introduced by dynamic AMP sampling from a small table (too few rows per AMP to provide an accurate measure) or a skewed value set is much higher than the estimates gained by the full-table statistics gathered using the Optimizer form of the COLLECT STATISTICS statement. (For a list of those statistics, see [Statistics and Demographics Collected and Computed](#).) The term *skewed* here means having outlier values that bias the value distribution in the statistical sense. It does not refer to an unbalanced distribution of table rows among the AMPs of a system.

Because the Optimizer understands that the statistics and demographic information it collects using a single AMP sample is less reliable, it assumes Low confidence (as reported by an EXPLAIN of a query where dynamic AMP sampling would be used) in the outcome and is less aggressive in its pursuit of optimal query plans, particularly join strategies.

Actually, the EXPLAIN text would report No confidence for this example because there are no statistics on the condition. When the request is actually performed, Vantage would perform a dynamic AMP sample,

and the resulting cardinality estimate would then be expressed with Low confidence. See [About Optimizer Confidence Levels](#) for more details about Optimizer confidence levels.

This means that in some 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.

Estimating Cardinalities From a Multiple-AMP Dynamic Sample

The only procedural difference between dynamic single-AMP and dynamic 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 IDs. Vantage selects sequential AMP IDs to obtain the most efficient load distribution. 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.

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.

Dynamic AMP Sampling of USIs

Dynamic AMP sampling assumes that the number of distinct values in a USI equals its cardinality, so it does not read the index subtable for USI equality conditions. The number of distinct values in the USI is assumed to be identical to the table cardinality taken from the dynamic AMP sample on the primary index.

Because equality conditions on a unique index return only one row by definition, the Optimizer always chooses the direct USI path without costing it or using statistics. However, if a USI will frequently be specified in non-equality predicates, such as range constraints, then you should collect statistics on it.

Dynamic AMP Sampling of NUSIs

Dynamic AMP sampling for NUSIs is very efficient. The system reads the cylinder index that supports the index subtable rows on the sampled AMP and determines the number of rows on that cylinder. Except for situations where the NUSI is very nonunique, there is one subtable row for each distinct value in the NUSI. Following through on that knowledge, the sampling process assumes that each subtable row it finds translates to one index value.

The sampling process assumes the following assertions about the data are true.

- The same values occur on all AMPs
- The number of distinct values found on the sampled AMP represents the total number of distinct values for the NUSI

Because of these assumptions, dynamic AMP samples are less accurate for fairly singular NUSIs than they are for fairly nonunique NUSIs. If more than one AMP is sampled, the system calculates an average number of distinct values across all sampled AMPs.

NUSI estimates are always made after the total cardinality of the table has been estimated. The sampling process divides the estimated total cardinality by the NUSI cardinality to estimate the approximate number of rows per value in the index. If a request passes a single value in an equality condition to match to the NUSI, and skew, if present, is not extensive, then query plans are generally quite good.

The following table summarizes some cases illustrating dynamic-AMP sampled NUSI estimates where there is no significant skew in the data. Note that in the cases where there are very few rows per value in the index (c_phone and c_acctbal), the system makes less accurate cardinality estimates.

Table	NUSI Col	Query Text	Result Cardinality	Est. Result Cardinality (AMP Sample)	% Difference	Est. Result Cardinality (Full Stats)	% Difference	% Difference Between Estimates
parttbl	p_type	SELECT * FROM parttbl WHERE p_type = 'economy brushed copper';	67,103	67,123	0.03	66,667	0.65	0.68
partsupp	ps_supkey	SELECT * FROM partsupp WHERE ps_supkey = 555;	80	82	2.47	81	1.24	0.01
customer	c_acctbal	SELECT * FROM customer WHERE c_acctbal = 7944.22	10	24	82.35	7	35.29	109.68
customer	c_phone	SELECT * FROM customer WHERE c_phone = '25-548-367-9974';	1	21	81.82	2	165.22	158.33

Sampling Efficiency for Non-Indexed Predicate Columns

While dynamic AMP sampling provides reasonable cardinality estimates for UPIs, USIs, and distinct NUSI values, it fails to provide useful cardinality estimates for unindexed predicate columns in a query. When the WHERE clause of a request specifies a predicate of `zip_code=90230` and there are no statistics on the column `zip_code`, Vantage uses various heuristics to apply default selectivities to estimate default cardinalities for those columns. For this example, the heuristic is to estimate the cardinality to be 10% of the table rows.

You should collect statistics on frequently selected columns, particularly if their values are skewed, rather than relying on the default cardinality estimates.

The table on the next page shows three queries with simple selection criteria. The actual response cardinalities from the queries are presented, along with the cardinality estimates derived from the heuristic defaults, the cardinality estimates derived from the collection of full statistics, and the percentage difference between the 2 estimates. There is considerable divergence between the true cardinalities and the heuristic estimates in all cases, while the estimates made by collecting full statistics are very close to reality in all cases.

If you compare the actual cardinalities to the default estimates, you can see that with a single selection column in an equality condition, the Optimizer assumes that about 10% of the rows in the table will be returned. With 2 selection criteria, as seen in the third request in the table, the Optimizer assumes that about 7.5% of the rows will be returned. In every case, the default cardinality estimates significantly overestimate the number of rows returned, potentially leading to poor join planning if the example selection criteria were part of a more complex query where additional tables were involved in join operations with them.

If NUSIs had been defined on those columns, the Optimizer would use a dynamic AMP sample estimate of their cardinalities even if the NUSI column had not been used to develop the query plan.

For selection criteria on unindexed columns, the identical poor cardinality estimates are made whether dynamic AMP sampling is defined for one, few, or all AMPs in the system. The cardinality estimates for unindexed columns do not improve when all AMPs are sampled because Vantage always samples 10% of the table rows irrespective of the number of AMPs on which the sampling is done.

The significant conclusion to be drawn from these results is that it is important to collect statistics on non-indexed columns that are frequently specified in selection predicates. Dynamic AMP samples never provide adequate cardinality estimates for unindexed columns, even if those columns are not skewed.

Query Text	Table Cardinality	Response Set Cardinality	Est. Response Set Cardinality (Heuristic)	% Difference	Est. Response Set Cardinality (Full Stats)	% Difference
SELECT * FROM lineitem WHERE l_commitdate = '1998-01-06';	300,005,811	123,959	30,017,594	198.36	124,591	0.25
SELECT * FROM partsupp WHERE ps_availqty = 7874;	40,000,000	3,936	3,991,526	199.61	4,000	1.62
SELECT * FROM parttbl WHERE p_size = 22 AND p_brand = 'Brand#23';	10,000,000	8,061	750,398	195.75	7,997	0.80

Comparing the Accuracies of Methods of Collecting Statistics

Comparative Accuracy of Sampled and Population Statistics

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. 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.

Although this is undeniable, it is not possible to know a priori whether it is necessary to collect a full set of new statistics in order to ensure that the Optimizer produces the best query plans.

Ranking the Relative Accuracies of the Various Methods

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

1. Dynamic AMP samples are better than residual statistics in the majority of cases.

Dynamic AMP samples are also recollected each time a DBD is retrieved from disk, so they are nearly always more current than residual statistics.

2. Dynamic all-AMPs samples are better than dynamic AMP samples in most cases.
3. Full-table population statistics are usually better than any form of sampled statistics.

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

Collection Method	Relative Elapsed Time to Collect	Accuracy Rank (Higher Number = Higher Accuracy)	Comments
None. Use residual statistics	None.	1	<ul style="list-style-type: none"> • Impossible to predict with certainty if residual statistics will produce a good query plan. At worst, can produce a very poor query plan. At best, can produce as good a query plan as freshly collected statistics. • Because statistics exist, Optimizer does not collect fresh statistics using a dynamic AMP sample. • Optimizer does use derived statistics, but their starting point is the existing statistics. The derived statistics subsystem has ways to compensate for stale statistics, but it is still better to begin with the freshest set of statistics that can be made available

Collection Method	Relative Elapsed Time to Collect	Accuracy Rank (Higher Number = Higher Accuracy)	Comments
			(see Using Derived Statistics to Compensate for Stale Statistics).
Dynamic AMP sample	Almost none.	2	<ul style="list-style-type: none"> Data is collected from a subset of the rows on a single AMP and might not be representative of full table demographics. Collecting data from a subset of the rows on a single AMP is the default method. Depending on an internal DBS Control variable, the default number of AMPs sampled ranges over 1, 2, 5, all AMPs on a node, or all AMPs on a system. Consult your Teradata support representative for details. Collects statistics for table cardinality, average cardinality per value, average cardinality per index, average size of each index on each AMP, and number of distinct index values only. Because the sample size is small, there is a high bias in the statistics. Accuracy is very sensitive to skew. Provides fairly accurate estimates of base table and NUSI cardinality if table is large, distribution of values is not skewed, and data is not taken from an atypical sample. <p>Other standard statistical parameters are less likely to be as accurate.</p>
All-AMP sample	Approximately 5% of time to perform a full-table scan. When the data is skewed, this percentage is larger, depending on how much the system dynamically increases its sample size	3	<ul style="list-style-type: none"> Data is collected from a system-determined subset of the rows on all AMPs. Collects identical statistics to full-table scan, including interval histogram creation. Percentage of sampled rows is small. Percentage of sampled rows is increased dynamically to enhance the accuracy of collected statistics if skew is detected in the samples.
Full-table scan	Approximately 195% of the time to perform sampled statistics.	4	<ul style="list-style-type: none"> Data is collected from all rows on all AMPs so there is no sample bias. Collects full statistics and creates interval histograms. Skew is accounted for using high-biased intervals in the statistical histogram for a column or index.

Collection Method	Relative Elapsed Time to Collect	Accuracy Rank (Higher Number = Higher Accuracy)	Comments
			<ul style="list-style-type: none"> Up to 100 histogram intervals are used, depending on the data.

Optimal Times to Collect or Recollect Statistics

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 provides the best table statistics over the long run is collecting and recollecting full-table statistics frequently. You also have the choice of specifying a sampling percentage, an elapsed number of days threshold, or a data demographics change percentage threshold, which instruct Vantage not to recollect statistics unless the specified thresholds are met or exceeded. Also be aware that table cardinalities are updated whenever rows in a table are updated or deleted or when new rows are added to a table by the Object Use Count and UDI system, so the Optimizer has access to accurate cardinality information whether statistics are current or not. For details about these systems, see [Object Use and UDI Counts](#).

Teradata recommends always collecting full-table statistics on a regular basis unless you specify USING clause sampling or threshold options. In this case, the Optimizer determines whether a request to recollect statistics for a column or index set should be honored. The responsibility for submitting the COLLECT STATISTICS request remains yours, but if you specify one or more of the sampling or threshold options, Vantage makes the decision whether to recollect the specified statistics based on various historical criteria for the relevant histogram. For more information, see [Recollecting Sampled Statistics](#) and the information about COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Just how frequently statistics should be recollect is contingent on several factors, and if you take advantage of the sampling and threshold options that are available to you when you initially collect statistics, you need not be concerned about determining when it is necessary to recollect statistics because Vantage makes that determination for you.

To avoid making this decision arbitrarily, you should submit your requests to collect and recollect statistics using one or more of the sampling or threshold options that are described briefly in this topic and more extensively in the documentation for the COLLECT STATISTICS (Optimizer Form) statement in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. Then all you need to do is submit your requests to recollect statistics and Vantage makes the determination of whether statistics need to be recollect or not based on the threshold and sampling criteria you have established. If the system determines that statistics need not be collected, it does not honor your request to recollect them, and you do not need to worry about recollecting statistics unnecessarily.

Depending on 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 Dynamic AMP Sampled Statistics for Static Columns](#) 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 dynamic 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.

Using the Teradata Viewpoint Stats Manager

The Teradata Viewpoint Stats Manager portlet provides a method of determining when fresh statistics should be collected.

See [Teradata Viewpoint Stats Manager](#) and the *Teradata® Viewpoint User Guide*, B035-2206 for information about the uses and capabilities of this portlet.

Policies for Collecting Statistics

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

Following such policies is not as critical if you collect statistics using thresholds because Vantage does not recollect statistics when the thresholds you specify are not met. You must still submit COLLECT STATISTICS requests, but statistics are not recollected unless the specified thresholds are met or exceeded. This prevents you from wasting valuable system resources to recollect unnecessary statistics.

Policy	Required or Recommended
Recollect all statistics when you upgrade to a new database release.	Strongly recommended.
<p>By the Ten Percent Rule, you should recollect statistics whenever table or partition demographics change by 10% or more. This rule applies to both row partitioning and column partitioning.</p> <ul style="list-style-type: none"> • For a nonpartitioned table, recollect statistics whenever the demographics of the table change by 10% or more. • For a row-partitioned table, recollect statistics whenever the demographics of the row partitions change by 10% or more. • For a column-partitioned table, recollect statistics whenever the demographics of the column partition change by 10% or more. <p>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%.</p>	Strongly recommended.

Policy	Required or Recommended
Specify appropriate sampling or threshold options or both when you collect and recollect statistics. This enables the Optimizer to determine whether fresh statistics need to be collected or not. See COLLECT STATISTICS (Optimizer Form) information in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144 for information about how to specify sampling and threshold options in the USING clause of COLLECT STATISTICS requests.	Strongly recommended.
Collect statistics on newly created, empty tables 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 minimal and essential.

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

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.
- Non-indexed columns frequently referenced in WHERE clause predicates, particularly if those columns contain skewed data.
- The partitioning column set of a row-partitioned table.
- The system-defined PARTITION column for all partitioned (column or row) tables.

The value for the column partition number of the system-derived column PARTITION for a column-partitioned table is always 1, so collected statistics on a PARTITION column equate to the number of rows in each combined row partition.

Optimizer Use of Statistical Profiles

The COLLECT STATISTICS (Optimizer Form) statement creates histograms for, and updates statistics and demographics about, a specified column set or index. It then uses this information to compute a statistical

synopsis or profile of that index or column set to summarize its characteristics in a form that is useful for the Optimizer when it generates its access and join plans.

Sometimes it must seem as if every page you read in the Teradata Vantage SQL manual set recommends that you 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.

For syntax and usage information about COLLECT STATISTICS (Optimizer Form), see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Statistics and Demographics 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 summary statistics for a column or index by submitting a HELP STATISTICS (Optimizer Form) request. To view the detailed statistics for an index or column set, you must submit a SHOW STATISTICS request.

For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The description of some statistics depends on whether they describe an equal-height interval or a high-biased interval. For more information, see [Interval Histograms](#).

Different statistics are stored for a column depending on whether its values are highly skewed or not. If the distribution of column values is not skewed, then its statistics are stored in an equal-height interval histogram. If the distribution of column values is highly skewed, then its statistics are stored in a High-biased interval histogram.

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 of its value distributions.

Attribute	Description
Statistics Maintained for All Interval Types	
Date collected	Reported by HELP STATISTICS and SHOW STATISTICS as Date. The date on which statistics were last collected.
Time collected	Reported by HELP STATISTICS and SHOW STATISTICS as Time. The time at which statistics were last collected.
Number of rows	Reported by SHOW STATISTICS as Number of Rows. An estimate of the cardinality of the table.
Number of nulls	Reported by SHOW STATISTICS as Number of Nulls. An estimate of the number of rows with partial or completely null columns for the column or index column statistics set.

Attribute	Description
Number of intervals	Reported by SHOW STATISTICS as Number of Intervals. The number of intervals in the frequency distribution histogram containing the column or index statistics.
Number of all nulls	Reported by SHOW STATISTICS as Number of All Nulls. An estimate of the number of rows with all columns null for the column or index column statistics set.
Sampled percent	Reported by SHOW STATISTICS as Sampled Percent. The approximate percentage of total rows in the table included in the sample. Null or 0 indicates that sampling is not active, which means that full statistics (a 100% sample) are being collected.
Version number	Reported by SHOW STATISTICS as Version. The version number of the statistics structure in effect when the statistics were collected.
Table cardinality estimate from a single-AMP sample	Reported by SHOW STATISTICS as OneAMPSampleEst for SUMMARY statistics. Used by the Optimizer to extrapolate cardinality estimates and to detect table growth. Updated whenever summary statistics or statistics on any column set are collected or refreshed.
Table cardinality estimate from an all-AMP sample	Reported by SHOW STATISTICS as AllAMPSampleEst for SUMMARY statistics. Used by the Optimizer to extrapolate cardinality estimates and to detect table growth. Updated whenever summary statistics or statistics on any column set are collected or refreshed.
Number of distinct values	Reported by HELP STATISTICS and SHOW STATISTICS as Number of Uniques. An estimate of the number of unique values for the column.
Minimum value for the interval	Reported by SHOW STATISTICS as Min Value. 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 or SHOW STATISTICS. 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 or SHOW STATISTICS. 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 SHOW STATISTICS as MaxValue. An estimate of the largest value for the column or index in the interval.
Modal value for the	Reported by SHOW STATISTICS as ModeValue. An estimate of the most frequently occurring value or values for the column or index in the interval.

Attribute	Description
Number of rows having the modal value	Reported by SHOW STATISTICS as Mode Frequency. 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 SHOW STATISTICS as Non-Modal Values. 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 SHOW STATISTICS as Non-Modal Rows. An estimate of the skewness of the distribution of the index or column values within the interval.
High-Biased Interval Statistics	
Biased value	Reported by SHOW STATISTICS as BiasedValue.
Biased value frequency	Reported by SHOW STATISTICS as BiasedValueFreq.

Using Interval Histograms to Make Initial Cardinality Estimates

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.

To simplify the examples., the data demographics of the tables examined in these examples are such that their data is stored in equal-height intervals.

Note that what is described here is only the initial estimation of cardinalities, which is based on the statistics stored in the interval histograms. The Optimizer adjusts its cardinality estimates dynamically during the course of query optimization based on information derived from various database constraints, query predicates, and hash and join indexes. Because of the way the system calculates these statistics, they are referred to as *derived statistics* (see [Derived Statistics](#) for details).

The example sets that follow do not deal with derived statistics (see [Derived Statistics](#)). They are meant to indicate only how the Optimizer would make its cardinality estimates if it only had the statistics in the interval histograms as a basis for making those estimates.

The Optimizer also uses UDI count data from DBC.ObjectUsage to estimate cardinalities. See [Object Use and UDI Counts](#) for further information.

Interval Histogram Data

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

- Maximum value in the interval.
 - Most frequent value in the interval.
- Recorded as the modal value for the interval.

- Cardinality of the most frequent value in the interval.

Recorded as the modal frequency for the interval.

- Cardinality of distinct values in the interval as determined from a full-table scan.
- Cardinality of values in the interval not equal to the most frequent value.

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

Its value is calculated as follows:

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.

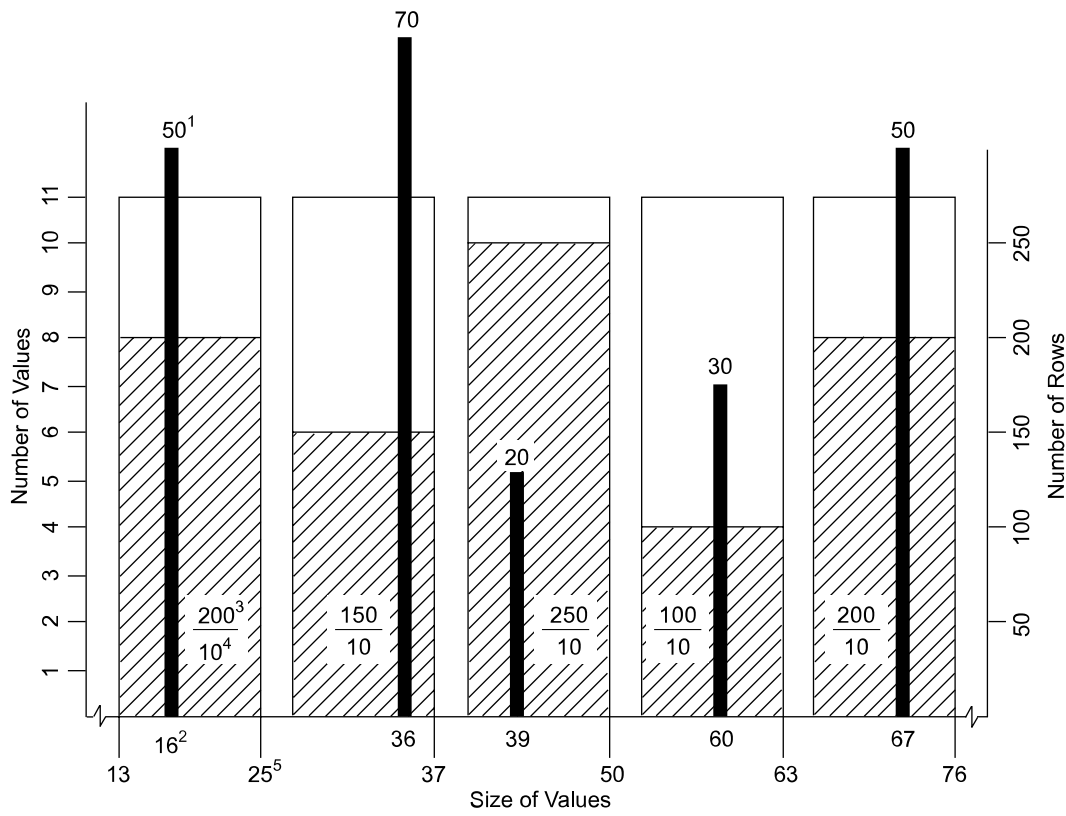
- Cardinality of rows not containing the most frequent value in the interval.

Recorded as the non-modal frequency for the interval.

The data used for the examples is tabulated in the following table. Notice that the total number of distinct values (the shaded cells of the table) is the same for all five intervals. This is definitive, in theory, for equal-height interval histograms. In practice, the cardinalities of intervals of an equi-height interval are only approximately equal. To simplify the examples, no history intervals are shown.

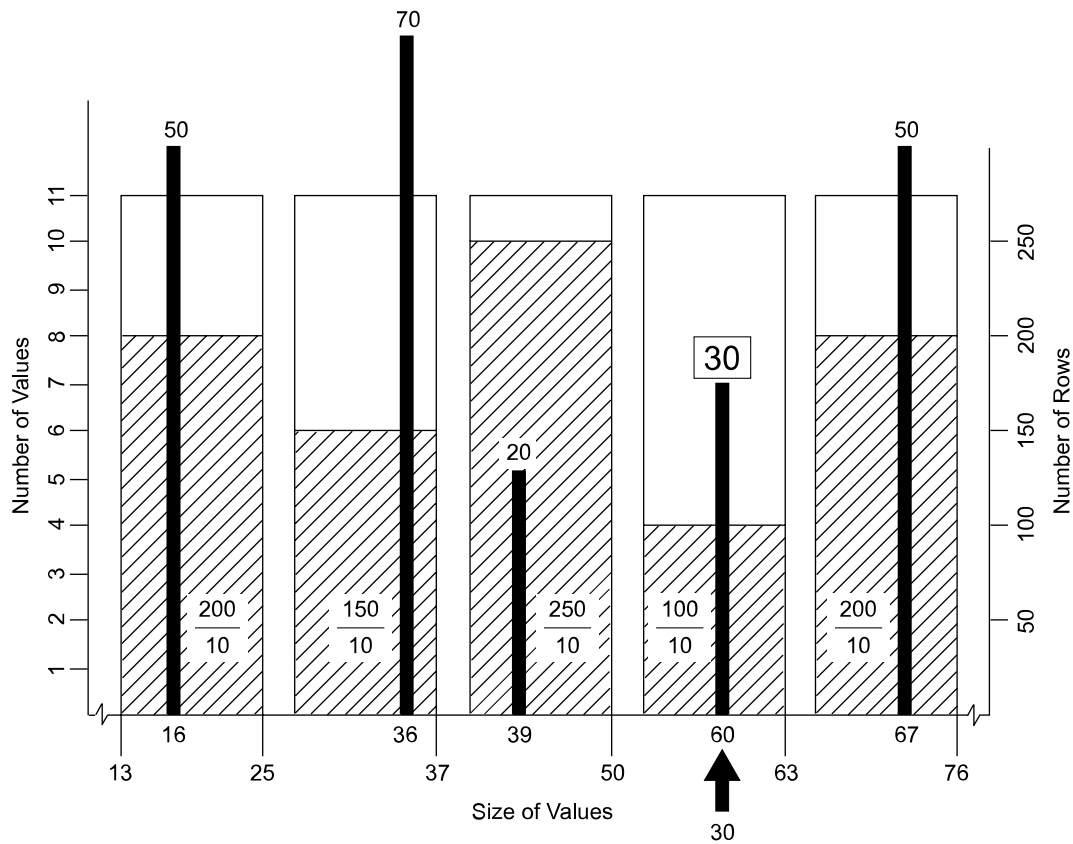
Variable	Interval Number				
	1	2	3	4	5
Instances of the most frequent value in the interval	16	36	39	60	67
Number of values not equal to the most frequent value in the interval	10	10	10	10	10
Number of distinct values in the interval	11	11	11	11	11
Number of rows in the interval having the most frequent 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 diagram illustrates these numbers graphically. The hatched area represents the number of rows for other values in the interval. Some of the values have superscript explanatory notes that are defined in the table following the illustration.



Graphic Note 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 distinct values in the interval that are not equal to the most frequent value for the column or index.
5	Maximum value in the interval.

If the cardinality of the response set for this simple query need not be estimated because 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.

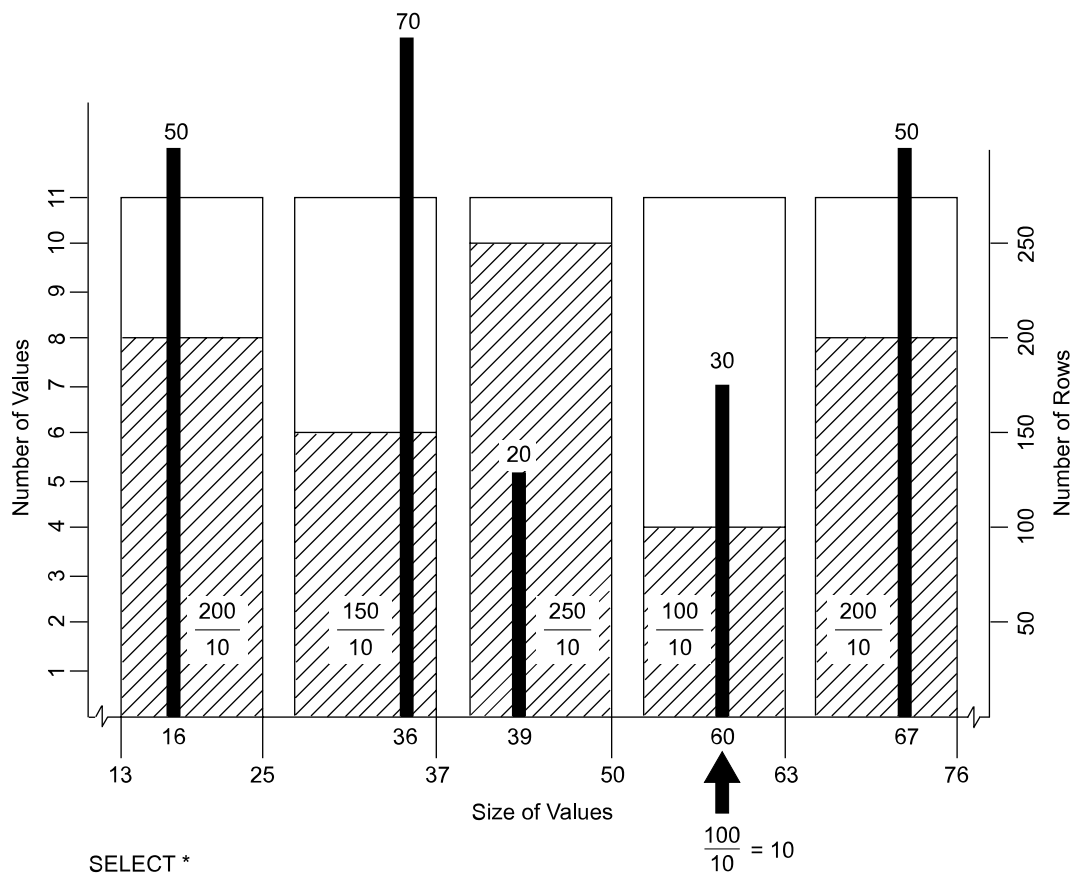


```
SELECT *
FROM table
WHERE condition = 60;
```

[30 rows]

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 must estimate the cardinality of the response set for this example because unlike the previous example, 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.

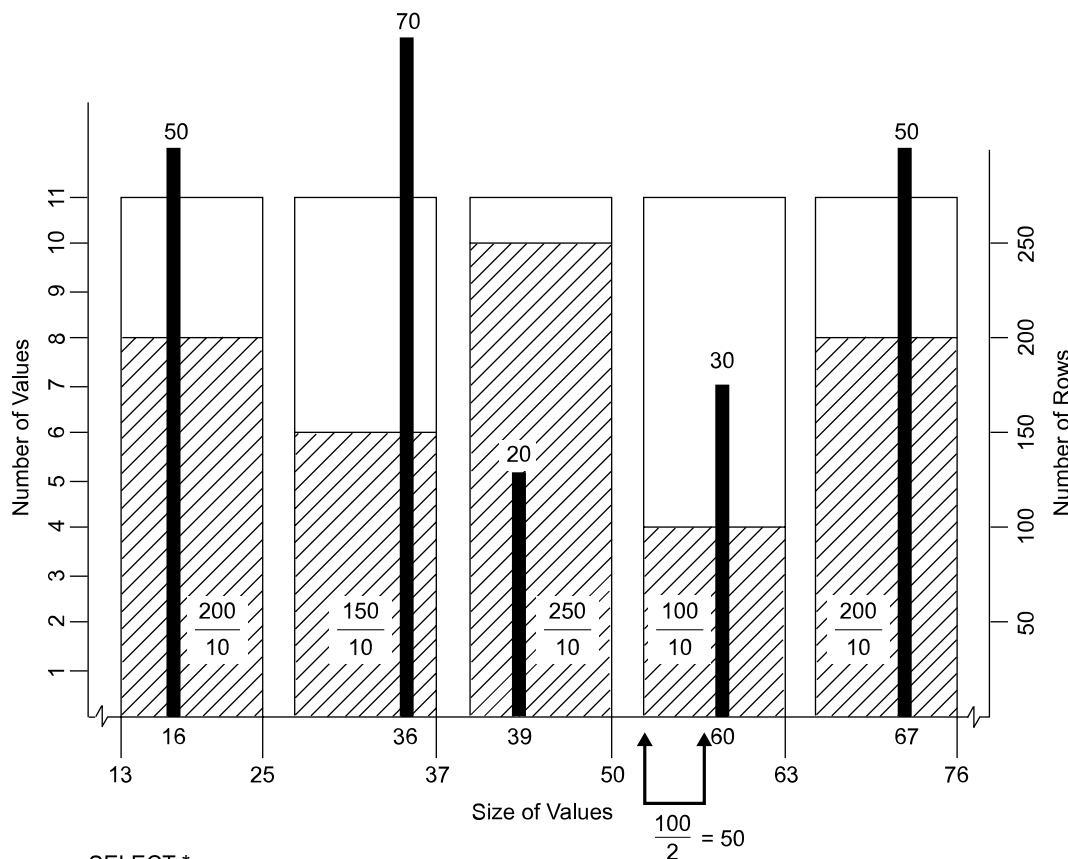
$$\text{Estimated cardinality of the response set} = \frac{\text{Number of rows not having the most frequent value}}{\text{Number of values not the most frequent value}}$$

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

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

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 $value \geq lower_limit$ AND $\leq upper_limit$, 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.

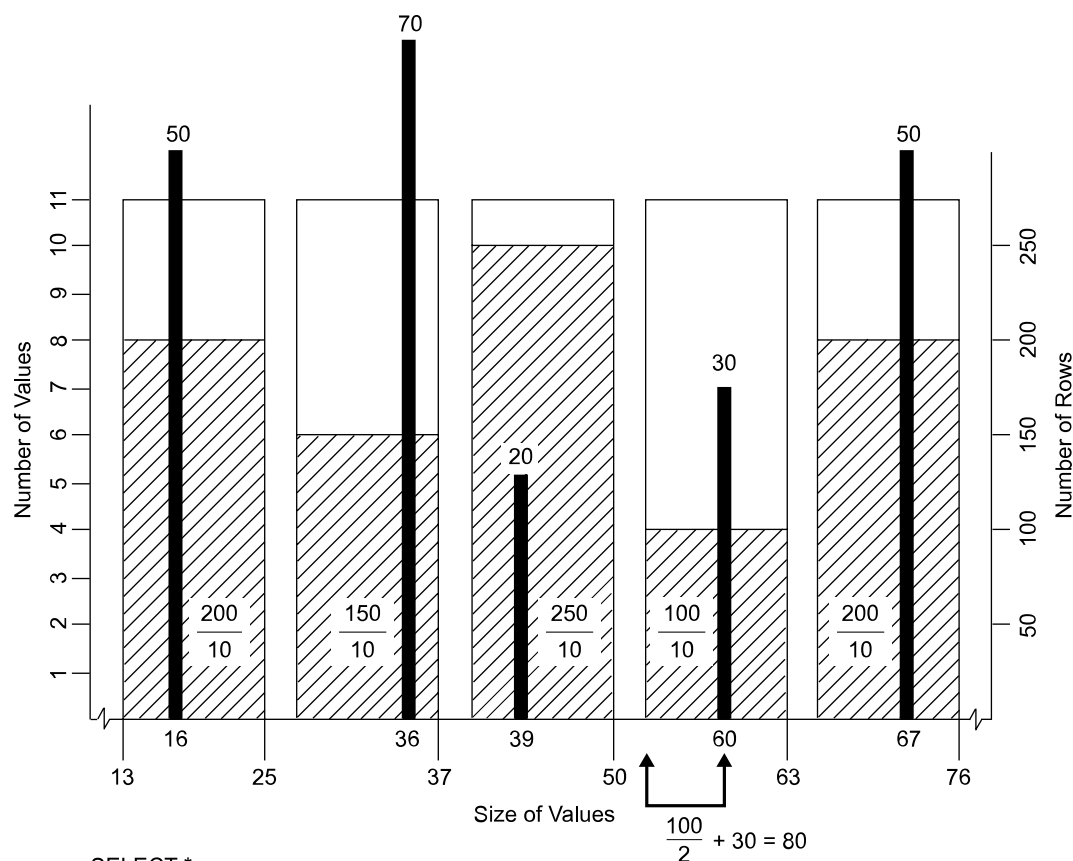
$$\text{Estimated cardinality of the response set} = \frac{\text{Number of rows not having the most frequent value}}{2}$$

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.

This example is slightly more sophisticated than the previous example 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.

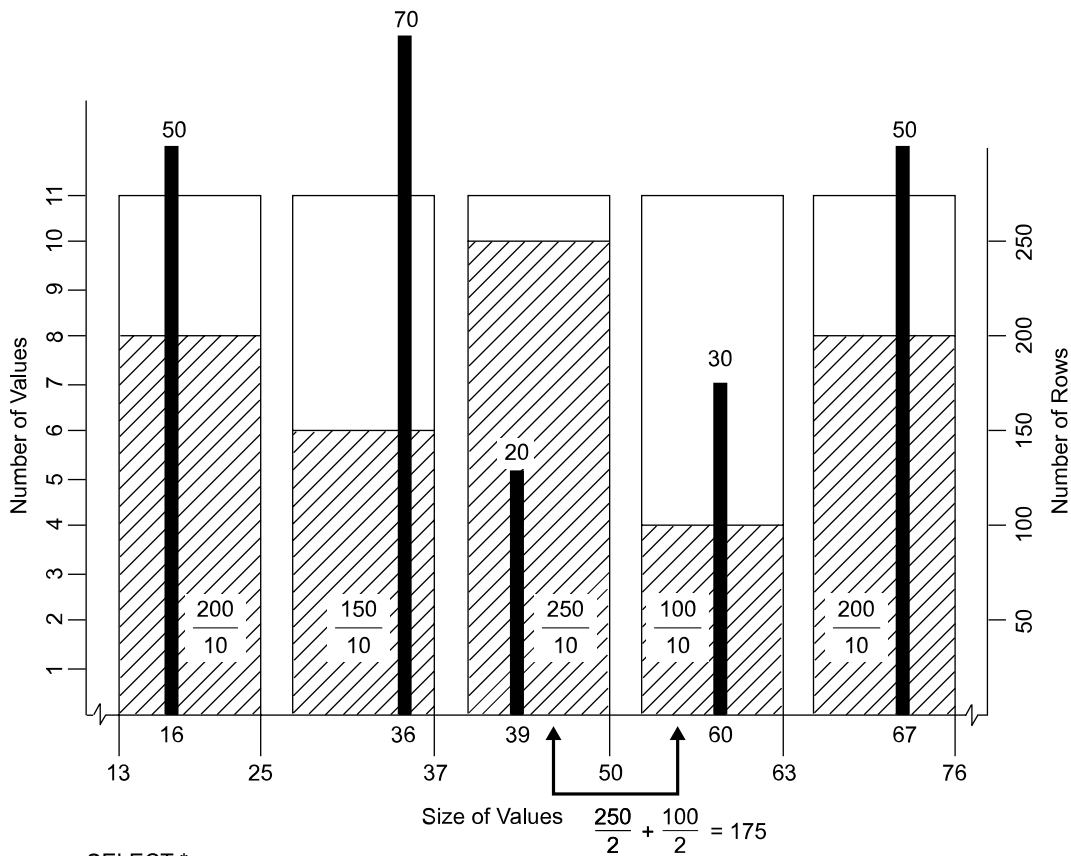
The Optimizer has to estimate the cardinality of the response set for this example because there are only partial exact statistics to describe it. The heuristic used to estimate the response set cardinality is to divide the number of rows not having the most frequent value in the interval in half and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

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

This example is a slightly more complicated range query than the previous one because the response set spans 2 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 2 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 2 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 2 cardinalities.

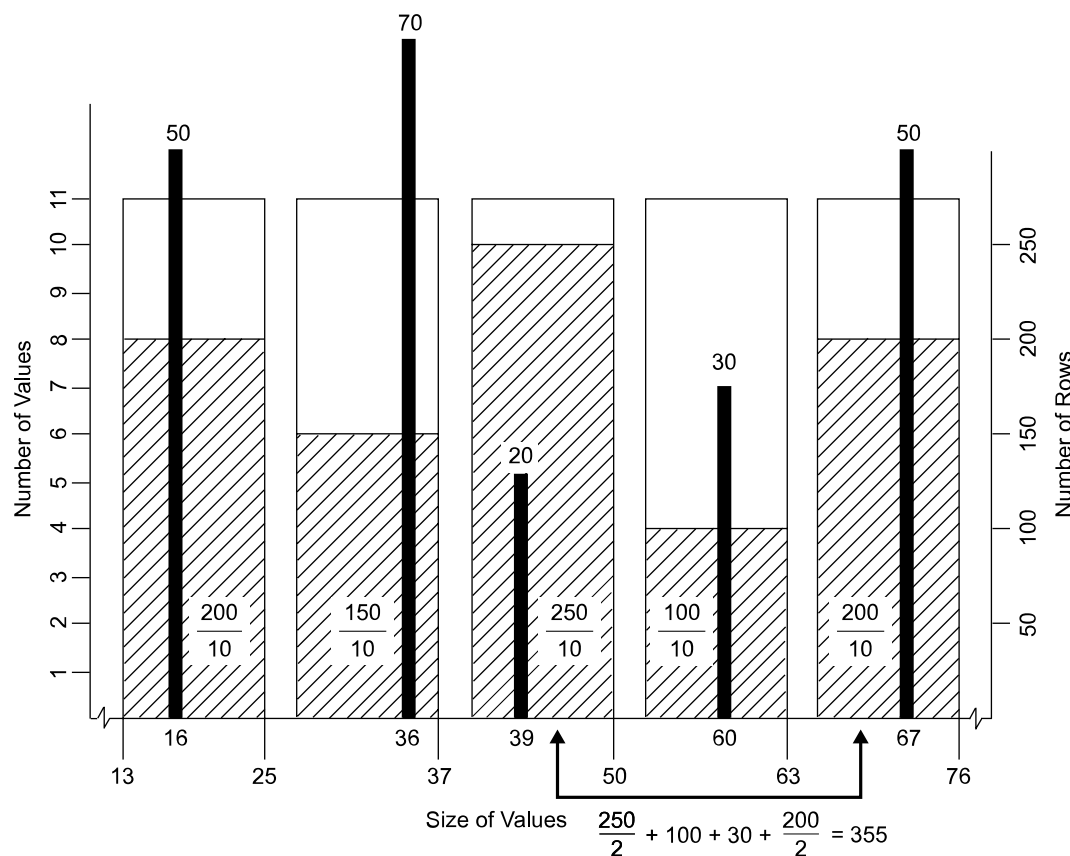
There are 250 rows in the lower interval that do not have a value of 20 for the column specified by condition, so the Optimizer divides the number of rows not having the value 20, which is 250, in half, producing an estimate of 125 rows satisfying the condition for that interval.

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 2 intervals.

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

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 computed 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 value 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 as in the list that follows:

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

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

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

Note that all of these examples are meant to show only how interval histograms are used in query optimization. They do not factor in the methods used by derived statistics and UDI counts to make cardinality estimates even more accurate.

Determining the Reliability of Statistics From History Intervals

Vantage performs linear trend analyses (linear regression) of historical demographics to determine the reliability factor for downgrading to sampled statistics, applying threshold techniques, stale statistics extrapolations. The reliability factor is derived from a combination of various stability and consistency factors.

Vantage retains history records and analyzes changes in the trends of the cardinalities for collected statistics. The Optimizer attempts to find a linear relationship between each statistic and its cardinality trends. If it finds a reliable linear relationship, it models the relationship as a linear function. Vantage uses

the linear functions it develops to estimate the cardinality for a current statistic at a given time. If the Optimizer does not find a reliable trend, it uses heuristics to estimate the cardinality of a statistic. When the historical change trends for statistics do not match their recent changes, Vantage purges the past statistics automatically.

This approach assumes that the current and future changes of statistics will be similar to any recent changes. For situations when that assumption is clearly not valid, such as when newly added or updated data is significantly different from recent past insertions deletions, or updates, you should recollect statistics.

Using a Trend Line in Estimating Statistics Values

The history of a statistic is a set of history records. Vantage maintains the following types of statistics history records:

- Those based on timestamps and cardinalities
- Those based on statistics values and cardinalities

Among the historical statistics values maintained are the number of unique values, the number of nulls, the high modal frequency, the minimum value, and the maximum value.

The criteria Vantage uses to select a single best trend line from the available candidate trend lines are as follows:

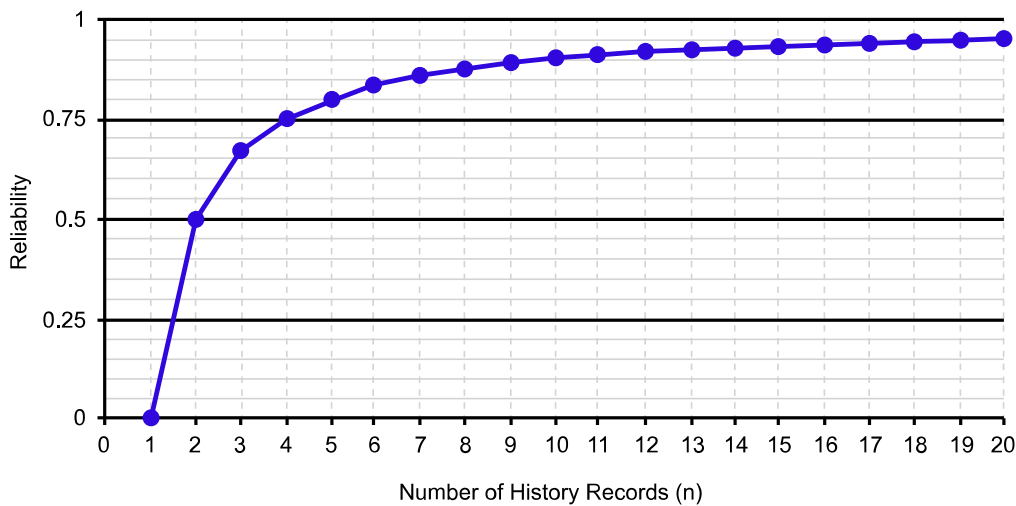
- The trend line must be derived from the most recent history records
- The trend line must be stable
- The trend line must be consistent

The reliability of an estimated statistical value is determined by a weighting function. A reliability weighting function with values that range between 0 and 1 represents how reliable a statistic computed using a linear regression method is.

Vantage computes this reliability based on information it derives from stability and consistency estimates made for the most recent trend. Given a stability threshold, a trend analysis locates all of the recent trends that satisfy a given stability threshold.

Based on a rule that states that a trend line is stable if its stability value between the bounds of 0 and 1 is greater than some specified threshold, Vantage selects the trend that has the largest number of history records.

The following graph shows a trend analysis where the trend line indicates that there should be a minimum of 10 history intervals that satisfy the stability threshold to reach a reliability value greater than 0.9.



Estimating Statistics Values Using Linear Regression

When history data is available, Vantage extrapolates or interpolates statistics values using linear regression.

For example, Vantage uses linear regression to estimate various measures of a value and then combines them to estimate a statistical value.

Derived Statistics

The initial cardinality estimates based on interval histogram statistics, from which the Optimizer begins the process of query optimization, are adjusted dynamically throughout the optimization process to gain increasing accuracy by applying information derived from various database constraints, query predicates, and hash and join indexes.

Because of the way the system calculates these statistics, they are referred to as *derived statistics*. Derived statistics enable the Optimizer to achieve the same end result of dynamically reoptimizing a request in mid-query when it has been determined to have a suboptimal plan without having to reoptimize the entire query from the beginning.

Definition of Derived Statistics

Derived statistics are snapshots, derived from base table interval histogram statistics, that the Optimizer initializes and transforms while it optimizes a query. They are cardinality estimates that are transformed from various constraint sources, including query predicates, CHECK and referential integrity constraints, and hash and join indexes, and then adjusted dynamically at each stage of the query optimization process. In other words, derived statistics represent the demographics of column values after applying query predicates and demographic information derived from other sources.

Optimizing queries based on information derived from integrity constraints falls into the general category of semantic query optimization, which was initially developed in the field of deductive databases. Note that semantic query optimization methods can also add or subtract derived constraints from a request in order to better optimize it.

For example, the Join Planner needs to know the demographics of join columns after applying single-table predicates or after making a join. All this information is derived and propagated throughout query optimization using the derived statistics infrastructure, and it is used consistently in all cost and estimation formulas.

The derived statistics framework also employs several techniques for dealing with stale statistics, including comparing cardinality estimates obtained from a dynamic AMP sample with the statistics stored in the relevant interval histogram, and bidirectional inheritance of statistics between a base table and its supporting indexes, using whichever set has the more recent collection timestamp. See [Statistical Inheritance by Hash and Join Indexes](#).

For more information about stale statistics, see [Using Derived Statistics to Compensate for Stale Statistics](#).

Bidirectional inheritance is the term used to describe how base tables and their underlying indexes are able to inherit and use existing statistics from one another when either database object in an index-table pair has no existing interval histogram statistics.

If both database objects have existing interval histogram statistics, the Optimizer assumes that the more recently collected statistics are the more accurate, so it uses the set with the more recent collection timestamp.

Derived Statistics Flow

The following table and join index definitions are used for the query that illustrates the flow of derived statistics usage by the Optimizer to generate more accurate cardinality estimates:

```
CREATE TABLE t2 (
  a2 INTEGER PRIMARY KEY,
  b2 INTEGER,
  c2 CHARACTER(1) CHECK (c2 IN ('M', 'F')),
  d2 DATE);

CREATE TABLE t3 (
  a3 INTEGER,
  b3 INTEGER,
  c3 CHARACTER(5),
  d3 INTEGER);

CREATE JOIN INDEX ji_t1 AS
  SELECT a1, d1
  FROM t1
  WHERE b1 > 10
  AND   c1 = 'Teradata');

CREATE JOIN INDEX aji_t3 AS
  SELECT a3, d3, COUNT(*)
  FROM t3
```

```
WHERE b3 < 100
GROUP BY 1,2);
```

Given these table and join index definitions, the flow chart that follows later in this topic shows how the Optimizer uses derived statistics to more accurately estimate cardinalities for the following query:

```
SELECT *
FROM t1, t2, t3
WHERE b1 > 10
AND   c1 = 'Teradata'
AND   b3 < 50
AND   d1 = d2
AND   a2 = a3
AND   d2 = d3;
```

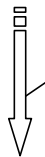
The stages of deriving various statistics for this query are as follows:

1. Refine and derive base table statistics using join index statistics.
 - Cap the number of unique values in t1.d1 at 1 500 using the join index statistics from ji_t1 on column d1 (see [Using Single-Table Sparse Join Indexes to Derive Column Demographics](#)).
 - No statistics have been collected on t2.(a2, d2), but statistics have been collected on a superset of those statistics, (a2, b2, d2), so the cardinality of that superset is stored in the derived statistics for t2.(a2, d2) and propagated to the next stage of the process (see [Using Subset and Superset Statistics to Derive Column Demographics](#)).
 - Inherit the statistics for table t3, for which no statistics have been collected, from the aggregate join index aji_t3 (see [Statistical Inheritance by Hash and Join Indexes](#) and [Using Aggregate Join Indexes to Derive Column Demographics](#)).
2. Join tables t1 and t2, consuming the term d1=d2 and producing the interim join relation R1.
 d1 and d2 are then merged into an *EquiSet* (see [Using Join Predicate Redundancy to Derive Column Demographics After Each Binary Join](#)), which takes the smaller of the 2 unique values cardinalities as MIN(200,1500), or 200.
 The entries for d1 and d2 are then removed from the derived statistics set.
3. Join table t3 with the interim join relation R1, consuming the terms a2=a3 and d2=d3 and producing the following set of derived statistics cardinalities for join relation R2.

R2	
Column Set	Number of Unique Values
(b1)	1000
(c1)	2000
(d1, d2, d3)	200

R2	
Column Set	Number of Unique Values
(a2, a3)	100
(a2, d2)	600
(a3, d3)	600

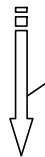
t1	
Col	NUV
(b1)	1000
(c1)	2000
(d1)	2500



Capping using
join index
statistics on d1

t1	
Col	NUV
(b1)	1000
(c1)	2000
(d1)	1500

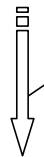
t2	
Col	NUV
(a2)	100
(d2)	200
(a2, b2, d2)	600



Adding using
Superset (a2, b2,
d2)

t2	
Col	NUV
(a2)	100
(d2)	200
(a2, d2)	600

t3	
Col	NUV
No base table statistics	



Derived from
aggregate join
index

t3	
Col	NUV
(a3, d3)	750

Consuming d1=d2
join term, d1 and d2
are merged into an
EquiSet, NUV
takes the smaller of
the two NUVs
MIN(200, 1500).
Entries for d1 and
d2 are deleted.

R1	
Col	NUV
(b1)	1000
(c1)	2000
(d1)	
(d2)	
(d1, d2) *	200
(a2)	100
(a2, d2)	600

Consume a2-a3 and
d2=d3

R1	
Col	NUV
(b1)	1000
(c1)	2000
(d1, d2, d3) *	200
(a2, a3) *	100
(a2, d2)	600
(a3, d3)	600

Column Heading	Represents
Col	the column set for which the number of unique values is derived by the derived statistics subsystem.
NUV	the number of unique values for a column set as derived by the derived statistics subsystem.

Statistical Inheritance by Hash and Join Indexes

Single-table non-sparse join and hash indexes inherit their statistics from their underlying base table if the statistics have not been collected on the index itself. This applies to single-column statistics, single-column-index statistics, multicolumn statistics, and multicolumn index statistics.

Available Forms of Statistical Inheritance

The following list summarizes the available forms of statistical inheritance:

- Underlying base tables can inherit the statistics from any single-table non-sparse join and hash indexes that reference them. In this case, it becomes possible to inherit statistics from multiple qualified join and hash indexes.
- Multicolumn statistics are inherited from an underlying base table by a single-table non-sparse join index or hash index defined on it.
- Partitioning columns can inherit the statistics from the system-derived PARTITION column, but only if the table is partitioned by a single-column partitioning expression.
- Bidirectional inheritance of statistics is implemented for single-table non-sparse join indexes and hash indexes so that statistics collected on the single-table non-sparse join or hash index are inherited by their underlying base table.

For this case, it is possible for there to be multiple qualified single-table non-sparse join or hash indexes from which a base table could inherit statistics.

- Bidirectional inheritance of statistics is implemented for PARTITION statistics, where those statistics are inherited by the partitioning column of a row-partitioned table if the table is partitioned by a single-column partitioning expression.

Bidirectional inheritance of statistics is not supported for PARTITION#L *n* statistics.

The inheritance of statistics occurs when the derived statistics are built for the base relations. All statistical inheritance is bidirectional.

Deriving Column Demographics

The Optimizer acquires derived statistics in several different ways. The next several topics introduce the more common acquisition methods of deriving statistics.

Using Single-Table Sparse Join Indexes to Derive Column Demographics

If there are single-table sparse join indexes that either fully or partially cover the single-table predicates of a relation, the Optimizer can use them to derive the demographics of the residual columns. This mechanism provides an automated way of deriving column correlations using single-table join indexes.

For example, suppose you have defined the following join index:

```
CREATE JOIN INDEX ji AS
  SELECT c1, d1
  FROM t1
  WHERE x1 > 10
  AND   y1 =10;
```

You then submit the following query:

```
SELECT t1.c1, COUNT (*)
FROM t1, t2
WHERE t1.d1 = t2.d2
AND   t1.x1 > 10
AND   t1.y1 = 10
GROUP BY 1;
```

Using the statistics on ji, the system can derive the demographics of the columns t1.c1 and t1.d1 after applying the single-table predicates t1.x1 > 10 and t1.y1 = 10. The newly derived demographics can then be used in join planning for the join column t1.d1 and in aggregate estimates to determine the number of groups for the grouping column t1.c1.

Note that partially covering join indexes can also be used to derive the demographics in all applicable cases. For example, suppose you define the following join index.

```
CREATE JOIN INDEX ji AS
  SELECT c1, d1
  FROM t1
  WHERE x1 > 10;
```

You then submit the following query:

```
SELECT t1.c1, COUNT (*)
FROM t1, t2
WHERE t1.d1 = t2.d2
AND   t1.x1 > 10
AND   t1.y1 = 10
GROUP BY 1;
```


In this case, the demographic information from the partially covering join index is used for columns c1 and d1. If the demographic information is available from both the base table and the join index, then the join index information is given higher precedence because it also captures the correlation information with respect to some single-table predicates.

Using Aggregate Join Indexes to Derive Column Demographics

The Optimizer can derive the unique values of grouping columns from aggregate join indexes. The derived values can then be used in join planning. This is another strategy to derive the column correlation information among the grouping column set automatically, and also after applying the single-table predicates.

For example, suppose you have defined the following aggregate join index:

```
CREATE JOIN INDEX ji AS
  SELECT a1, SUM (b1)
  FROM t1
  WHERE x1 > 10
  AND y1 = 10
  GROUP BY c1, d1;
```

You then submit the following query:

```
SELECT *
FROM t1, t2
WHERE t1.c1 = t2.c2
AND t1.d1 = t2.d2
AND x1 > 10
AND y1 = 10;
```

The cardinality of the aggregate join index is the number of unique values of the columns (t1.c1, t2.d1). If the unique values for these columns are already available from base table statistics, the system updates them with the information from the aggregate join index; otherwise, it creates new derived statistics entries. The unique values can then be used in join planning for estimating join cardinality, rows per value, skew detection, and so on.

Note that partially covered aggregate join indexes are also handled in the same way as non-aggregate single-table sparse join indexes, and that the statistics from aggregate join indexes are used automatically if that index replaces a derived table or view.

Using CHECK Constraints and Referential Integrity Constraints to Derive Column Demographics

CHECK and referential integrity constraints can also provide useful information to the Optimizer in the absence of statistics. For example, consider a column called patient_sex, which can have only 2 values:

M and F. Generally, users enforce the sanity of this column with a database CHECK constraint such as `patient_sex IN ("M","F")`.

If there are no statistics on this column and a query specifies the predicate `patient_sex = 'M'`, the system would normally assume that 10% of the rows qualify by default because of the equality condition, but because this column can have only 2 values, a more reasonable default is 50%.

$$\text{Total qualifying rows} = \text{Total rows} \times \frac{1}{\text{NumUniqueVals}} = 50 \text{ percent}$$

The Optimizer also considers CHECK constraints such as open or closed ranges, and IN lists when statistics are not available. Open ranges help in some scenarios when a query predicate can close the range. For example, if a CHECK constraint specifies the open range `x1 > 10`, and a query specifies the predicate `x1 < 20`, their combination produces a closed range that can then be used to derive the column demographics.

Similarly, if there are no statistics on the child table in a Primary Key-Foreign Key relationship, and statistics exist for the parent table PK column, many of its demographics, such as number of unique values, can be derived and used for the FK column in the child table. This information can be used for single-table estimates and in join planning for join cardinality and rows per value estimates. These kinds of estimates can also be useful for large child tables in those situations where collecting statistics is prohibitively expensive.

This mechanism can be effectively used in conjunction with non-enforced referential integrity constraints (also known as *soft referential integrity constraints*) to provide child table demographic information to the Optimizer without collecting statistics.

Note that this mechanism assumes that the child table has all the values of the parent table. If there is only a subset of values, or if the value set in this join column is significantly skewed, then you should collect statistics on the child column to avoid skewed redistributions and underestimates of cardinalities.

The derived statistics framework collects this kind of information from CHECK and referential constraints in the prejoin planning stage of query optimization and propagates them from there to the join planning stage.

Using Single-Table Predicates to Derive Column Demographics

Single-table estimation logic determines the probable number of qualified unique values and qualified high modal frequency, along with the number of qualified rows, for every predicate. The Optimizer can use this information for cases where a column specified in the single-table predicate is also specified in subsequent operations such as joins and aggregations.

The system can derive this information in the following ways:

- Using the base table statistics
- Using the statistics on a single-table join index

By using the base table statistics, Vantage can derive the column demographics using the single-table predicate on the column specified in the predicate. However, the derived information might not be adjusted later based on other single-table predicates. This is in line with the assumption of column independence in the absence of column correlation information.

For example, suppose you have the following query:

```

SELECT *
FROM t1, t2
WHERE   t1.d1 = t2.d2
AND     t1.d1 BETWEEN '1999-01-01' AND '2000-01-01'
AND     t1.c1 > 10;

```

Using base table statistics, the qualified number of unique values and qualified high modal frequency are derived and saved in the derived statistics entry for column t1.d1.

On the other hand, by using the available single-table join indexes, the Optimizer can get the demographics of the columns after applying all single-table predicates.

For example, suppose you have defined the following join index to support the previous query:

```

CREATE JOIN INDEX ji AS
  SELECT t1.d1
  FROM t1
  WHERE   t1.d1 BETWEEN '1999-01-01' AND '2000-01-01'
  AND     t1.c1 > 10;

```

If an interval histogram is available for the join index column t1.d1, then it is given higher precedence and captured in the derived statistics because it reflects the demographics of the column t1.d1 after both single-table predicates have been applied. See [Using Join Index Statistics to Estimate Single-Table Expression Cardinalities](#) for information about how the Optimizer can use single-table join index statistics to estimate the cardinalities of complex expressions that reference base table columns.

The derived information is then reused in join planning stages such as join cardinality estimation, rows per value estimation, and skew detection.

In general, the Optimizer considers any predicate that can be used for cardinality estimates for a query plan with the exception of complicated predicates. The following table provides some examples of predicates the Optimizer does consider and the statistics it uses to make cardinality estimates for those predicates.

Predicate	Definition
IS NULL	Self-defined.
	Self-defined.
IS NOT NULL	<ul style="list-style-type: none"> If nulls are present, the value is calculated as (TotalValues - 1). If no nulls are present, the value is just TotalValues.
	TotalRows - Number of Nulls
Single equality condition	Self-defined.
	<ul style="list-style-type: none"> If Loner/Mode is in any interval, the value is its frequency.

Predicate	Definition
	<ul style="list-style-type: none"> If Loner/Mode is not in any interval, the value is calculated as $\frac{\text{TotalRows}}{\text{TotalValues}}$.
Index with equality conditions on all columns	Self-defined.
	Self-defined.
NE	TotalValues - 1
	HighModeFreq of the distribution excluding the HighModeFreq for this value.
<ul style="list-style-type: none"> LT LE GT GE 	Values count derived from qualified intervals.
	HighModeFreq from the qualified intervals.
BETWEEN	LowValues + HighValues - TotalValues
	HighModeFreq from the qualified range intervals.
<ul style="list-style-type: none"> ORed term IN list 	The number of elements in the ORed list.
	The maximum frequency of all the ORed elements.
NOT IN term	TotalValues - Number of elements in NOT IN list
	The HighMode after excluding all IN list elements.

Examples of a term type that is not considered are ORed terms that reference multiple columns such as $x1=10 \text{ OR } y1>20$.

Using Single-Table Predicates to Derive Column Demographics in the Absence of Statistics

If single-table predicates such as EQ, IN lists, ORed predicates or closed ranges, are specified in a query, Vantage can derive some information from them, such as the number of unique values, and the derived information can then be used later in join planning if column statistics are not available.

An example of where such statistics would be derived is the following query:

```
SELECT *
FROM t1, t2
WHERE t1.d1 = t2.d2
AND t1.d1 in (10, 20);
```

Derived statistics captures and propagates the number of unique values information to the Join Planner based on certain single-table predicates even when no statistics have been collected on this column.

Using Single-Table Predicates and Multicolumn (PARTITION) Statistics to Derive Column Demographics

The demographics of the join columns are required after applying the single-table predicates to a join request. The Optimizer can use multicolumn statistics to derive the demographics of the join columns using some single-table predicates. This is another automated way of deriving column correlations using multicolumn statistics.

For example, suppose you submit the following query:

```
SELECT *
FROM t1, t2
WHERE t1.d1 = t2.d2
AND t1.x1 > 10;
```

The Join Planner needs to know the demographics of the join column t1.d1 after applying the single-table predicate t1.x1>10. Derived statistics derives the demographics of t1.d1 if there are multicolumn statistics (x1, d1) on t1. For PPI tables, if there is row partition elimination and the multicolumn PARTITION statistics are available, then the demographics of the join columns are derived based on the qualified partitions.

Note that the order of the columns in multicolumn statistics is important for these estimates. The columns are reordered internally based on their ascending field IDs irrespective of the order you specify for collecting multicolumn statistics. So if column d1 has smaller field ID than column x1, the multicolumn statistics are ordered internally as (d1, x1). In this case, it is not possible to derive the demographics of d1 for a given predicate on x1. Note that for multicolumn PARTITION statistics the PARTITION column is always the first column because it always has an internal field ID of 0.

Some of the single-table predicates the Optimizer considers for deriving the column correlations are equality conditions (x1=10), IN lists (x1 IN (10, 20, 30)), simple ORed lists on the same columns (x1=10 OR x1=20), and range predicates (x1 BETWEEN 10 AND 30, x1>10), and so on.

Also see [Using Join Index Statistics to Estimate Single-Table Expression Cardinalities](#) for information about how the Optimizer can use single-table join index statistics to estimate the cardinalities of complex expressions that reference base table columns.

Using Single-Column and Multicolumn Statistics to Discover Hierarchical Relationships in Denormalized Schemas

The derived statistics framework discovers hierarchical relationships for denormalized tables using statistics. The relationships are saved as value mappings in the derived statistics. The only mappings used to adjust demographics are those with 1 --> n relationships.

A combination of single and multicolumn statistics is required to detect these relationships.

A change to a single column cascades through the entire hierarchical chain. For example, if the value mapping for x --> y is 1 --> 5 and the value mapping for y --> z is 1 --> 10, then if one value is removed from x, 5 values are removed from y, and 10 values are removed from z.

More concretely, if the relationship between region --> nation is discovered to be 1 --> 5, then if one region is selected, the Optimizer detects that only 5 nations qualify.

In other words, if one region is disqualified by a single-table predicate, the Optimizer removes 5 nations from the nation column.

The following example illustrates this logic. Consider the denormalized dimension table `customer_nation_region`, which is defined as follows:

```
CREATE SET TABLE cust_nation_region (
  c_custkey      INTEGER,
  c_name         VARCHAR(25) CHARACTER SET LATIN CASESPECIFIC,
  c_address      VARCHAR(40) CHARACTER SET LATIN CASESPECIFIC,
  c_nationkey    INTEGER,
  c_phone        CHAR(15) CHARACTER SET LATIN CASESPECIFIC,
  c_acctbal      DECIMAL(15,2),
  c_mktsegment   CHAR(10) CHARACTER SET LATIN CASESPECIFIC,
  c_comment      VARCHAR(117) CHARACTER SET LATIN CASESPECIFIC,
  c_maritalstatus CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,
  n_name         CHAR(25) CHARACTER SET LATIN CASESPECIFIC,
  n_comment      VARCHAR(152) CHARACTER SET LATIN CASESPECIFIC,
  r_regionkey    INTEGER,
  r_name         CHAR(25) CHARACTER SET LATIN CASESPECIFIC,
  r_comment      VARCHAR(152) CHARACTER SET LATIN CASESPECIFIC)
PRIMARY INDEX (c_custkey);
```

In this example, the relationships are derived from a combination of single- and multicolumn statistics. The following statistics are required to discover the relationships:

- Single-column on `r_regionkey`
- Single-column on `r_regionkey`
- Single-column on `r_regionkey`
- Multicolumn on (`r_regionkey`, `n_nationkey`)
- Multicolumn on (`n_nationkey`, `c_custkey`)

The relationships discovered are.

- region --> nation is 1 --> 5
- nation --> customer is 1 --> 24,000

You can see an example of this in the following query, which specifies a predicate that selects only one regionkey value, and its EXPLAIN text:

```
EXPLAIN SELECT n_nationkey
FROM cust_nation_region
WHERE r_regionkey = 1
GROUP BY 1;
```

The following is a portion of the EXPLAIN output:

```
...
2) Next, we lock TPCD_OCES3.cust_nation_region for read.3) We do an all-AMPs SUM
step to aggregate from TPCD_OCES3.cust_nation_region by way of an all-rows
scan with a condition of ("TPCD_OCES3.cust_nation_region.R_REGIONKEY = 1"),
grouping by field1 ( TPCD_OCES3.cust_nation_region.N_NATIONKEY).
Aggregate Intermediate Results are computed globally, then placed
in Spool 3. The size of Spool 3 is estimated with low confidence
to be 5 rows (525 bytes). The estimated time for this step is
0.62 seconds.
-> The total estimated time is 0.63 seconds.
```

From the known hierarchical relationship between region and nation, which is 1 --> 5, the predicted number of nationkey rows is 5, which is verified by the EXPLAIN text highlighted in boldface type. By generalizing, if you were to write a request that selected 2 regionkey values, there would be 10 nationkey rows in the result set, and so on.

Using Subset and Superset Statistics to Derive Column Demographics

If there are multiple join predicates, and the join columns have some degree of correlation, multicolumn statistics are necessary to more accurately estimate join cardinality, rows per value, skew detection, and related statistics. Without having multicolumn statistics, and assuming column independence, the Optimizer multiplies the individual numbers of unique values to get an estimate of the combined number of unique values. However, if multicolumn statistics exist that are a superset of the join columns, they might provide more useful information about the column correlations than the column independence assumption would provide.

For example, consider the following query:

```
SELECT *
FROM t1, t2
WHERE t1.d1 = t2.d2
AND t1.x1 = t2.x2;
```

The system requires multicolumn statistics on (t1.d1, t1.x1) and (t2.d2, t2.x2) to be able to make accurate cardinality and costing estimates. If there are individual statistics on t1.d1, with 100 unique values, and t1.x1, with 50 unique values, and assuming column independence, the unique values are multiplied to calculate the combined maximum number of unique values, $100 * 50 = 5,000$, and having an upward bound of the total number of rows.

The derived statistics subsystem identifies such opportunities in the prejoin planning stage and derives dummy multicolumn statistics based on super sets. For example, if there is a multicolumn statistics collection that is a superset, such as (x1, y1, d1), with only 2,000 unique values, the Optimizer can cap the estimated unique value cardinality of 5,000 to a more accurate 2,000 by deriving a derived statistics entry (x1, y1) with 2,000 unique values, and then propagate the statistic to Join Planning.

Similarly, if there are individual entries and multicolumn statistics entries such as (x1), (d1) and (x1, d1) and single-table predicates such as $x1 \text{ IN } (1,2) \text{ AND } d1 < 20$, the Optimizer can update the demographics of the single-column entries while making its single-table estimates. Based on the updated information from the single-column entries, the multicolumn entry of derived statistics is updated and propagated.

Using Join Predicates to Derive Column Demographics

The Optimizer assumes join uniformity when it estimates join cardinalities and costs. In other words, it assumes that every value from the left relation in the join finds a match in the right relation if the number of values in the left relation is less than the number of right values in the right relation, and vice versa.

Based on this assumption, and given the join predicate $t1.d1 = t2.d2$, if $t1.d1$ has 100 unique values and $t2.d2$ has 50 unique values, then after the join, $t1.d1$ should have only 50 unique values. The Optimizer also considers multicolumn statistics for making adjustments. For example, given the following predicate and (x1, y1) with 100 unique values and (x2, y2) with 50 unique values, after the join, the number of (x1, y1) unique values is adjusted to 50.

```
t1.x1=t2.x2 AND t1.y1=t2.y2
```

The derived statistics subsystem performs this kind of analysis and adjusts the demographics of the join columns such as the number of unique values and the high modal frequency after each binary join and then propagates the adjusted demographic information to the next stages of the join planning process.

If column correlation information is available, the Optimizer uses it to adjust the other column demographics based on the adjustments to the join column. Otherwise, the join column is assumed to be independent of the other columns.

Assume that $t1.d1$ has 100 unique values, $t2.d2$ has 50 unique values, and $t3.d3$ has 200 unique values.

Suppose you submit the following query:

```
SELECT t3.d3, COUNT (*)
FROM t1, t2, t3
WHERE t1.d1 = t2.d2
AND t1.d1 = t3.d3
GROUP BY 1;
```

Assuming the join order is $(t1 \times t2) \times t3$, after the first join, the derived statistics subsystem adjusts the number of unique values of $t1.d1$ to the minimum number of unique values $(t1.d1, t2.d2) = 50$.

After the second join, the derived statistics subsystem adjusts the number of unique values of $t3.d3$ to 50, which is then used in the final aggregate estimate because $t3.d3$ is specified as a grouping column.

Using Join Predicate Redundancy to Derive Column Demographics After Each Binary Join

Some join predicates can become redundant after the system has made one or more binary joins. A predicate is redundant when it adds nothing to the overall selectivity for a query because its effect is equivalent to one or more other predicates that are also specified in that query.

The redundancy can result from either transitive closure or from user-defined query conditions. If such redundant predicates are not properly identified and handled during cardinality estimates and costing, they can have a negative effect on the optimization of the query, possibly leading to non-optimal query plans.

For example, suppose you submit the following query:

```
SELECT *
FROM t1, t2, t3
WHERE t1.d1 = t2.d2
AND   t1.d1 = t3.d3;
```

Transitive closure derives a new predicate $t2.d2 = t3.d3$ for this request. Assuming the join order $(t1 \times t2 \rightarrow j1) \times t3$, the join predicates for the second join are $j1.d1 = t3.d3$ and $j1.d2 = t3.d3$.

For more information about transitive closure, see [Predicate Simplification](#).

While estimating the number of unique values of $(d1, d2)$ for join $j1$, the Optimizer needs to be aware that these 2 columns had already been equated in the previous join, and it should not multiply their individual unique values to get the combined number of unique values. Instead, it should use $\text{MIN}(j1.d1, j1.d2)$ as the combined number of unique values.

The derived statistics infrastructure builds the appropriate entries by combining all the connected joined columns into EquiSets after every binary join. This way, the subsequent joins seamlessly handle the redundancy.

In join planning, an EquiSet is a set of columns that was equated in predicates from a previous join operation in the same query. Propagating EquiSets to subsequent join operations for reuse in a query is a fundamental component of the derived statistics framework.

For example, in the preceding example, an EquiSet derived statistics entry $(j1.d1, j1.d2)$ exists after join $j1$, with the minimum number of values of $d1$ and $d2$. When the subsequent join requests the unique values for the combination $(j1.d1, j1.d2)$ the Optimizer automatically uses the existing EquiSet entry.

Handling Multiple Sources of Information to Derive Column Demographics

As demonstrated in the preceding topics, information such as the number of unique values and the high modal frequency can be taken or derived from multiple sources such as single-table join indexes, multicolumn statistics, dynamic AMP samples, and interval histograms, among others.

It is not trivial to determine which of the available sources is likely to be the most useful or trustworthy. The following criteria all play an important role in making the determination:

- The information source captures the highest correlation.

- The information source covers the greatest number of single-table predicates.
- The information source is the most current (or, phrased another way, the least stale) source of information available.

For example, it is possible to have a join index producing non-correlated demographic information covering multiple single-table predicates, while at the same time there might be a multicolumn statistic covering one single-table predicate that produces highly correlated information for a join column.

Because of issues such as this, it is not possible to define precedence rules based on heuristics. To work around this restriction, the Optimizer quantifies the information and defines precedence dynamically based on the derived quantities.

- The first two criteria, highest correlation and greatest number of single-table predicates, can be quantified by using the number of unique values.

This translates to using the information source that provides the fewest unique values.

- The third criterion, least stale statistics, is also considered and explained in [Stale Statistics](#).

Propagating Column Demographics To All Temporary and Committed Joins

Column demographics are propagated to all temporary and committed joins using the derived statistics framework. As a result, demographics can be dynamically adjusted after each join, and the Join Planner does not need to reference base table interval histograms to retrieve the statistics.

Propagating Column Demographics For Materialized Instances Within a Query Block

If you specify complex predicates or outer joins in a query, they might become materialize into spool relations while the Optimizer is performing join planning within a query block.

Suppose you submit the following outer join query with a complex ON clause predicate:

```
SELECT *
FROM t1 INNER JOIN t2 ON (x1=x2)
      LEFT OUTER JOIN t3 ON (x2 NOT IN (SELECT x4
                                      FROM t4)
                          OR y3=10);
```

The ON clause predicate `x2 NOT IN (SELECT x4 FROM t4) OR y3=10` in this query makes this a complex outer join (if a subquery is specified in an ON clause, the predicate is classified as either semicomplex or complex).

One of the requirements for processing a complex outer join is to have only one left relation, so the left relations (t1, t2) are materialized into a single spool relation before the system processes the complex join. After materializing the left relations, derived statistics propagates the projected column demographic information for the materialized instance to the subsequent stages of join planning.

Propagating Column Demographics For Aggregate and Ordered-Analytic Function Estimates

To be able to do aggregate and ordered-analytic cardinality estimates after a series of joins, the demographic information of the grouping columns needs to be propagated through the join operations until final aggregations are done. The derived statistics subsystem adjusts the demographic information based on single-table and join predicates and then propagates it forward until final aggregation completes. For example, suppose you submit the following query:

```
SELECT t1.x1, t2.x2, COUNT (*)
FROM t1, t2
WHERE t1.x1 =10
AND    t1.y2 = t2.x2
GROUP BY 1, 2;
```

In the prejoin planning stage, derived statistics detects the single-table equality term on column x1 (t1.x1=10) and adjusts the number of unique values to 1. Also while doing join planning, the number of unique values of t2.x2 is adjusted to the minimum number of values of (t1.y2, t2.x2). This improves the aggregate cardinality estimates, which can benefit the overall plan if this query block is a spooled derived table or a view.

Propagating Column Demographics Across Query Blocks

Suppose you submit the following query:

```
SELECT *
FROM t1, (SELECT x2, x3, COUNT(*)
          FROM t2, t3
          WHERE x2=10
          AND    t2.x3=t3.x3
          GROUP BY 1, 2) AS dt
WHERE t1.x1=dt.x3;
```

The derived statistics framework carries the column demographic information of the derived table dt for columns x2 and x3 by making adjustments based on the derived table predicates. The adjusted demographics are then propagated to the join planning for the outer block.

Using Histograms to Estimate Cardinalities

An expression that can use histograms for cardinality estimation can contain either a unary or a binary operator. One of the operands must be either a column or a built-in functions or SQL operator from the following lists:

- UPPER
- LOWER
- NULLIFZERO

- ZEROIFNULL
- SUBSTR
- MOD
- CONCAT on columns of same table
- Implicit or explicit data type conversions on a column

The supported operators are the following:

- =
- ≥
- ≤
- >
- <
- <>
- IS NULL
- IS NOT NULL
- LIKE
- NOT LIKE
- BETWEEN
- IN

For binary operators, the other operand can be a simple expression involving any of the following:

- Constants whose value can be calculated by the Parser.
- System USING request modifier data or built-in function data such as a CURRENT_DATE value.
- Simple expressions involving another column from the same table.

For example, the selectivity of the single-table predicate $t1.x1 = t1.y1$ can be estimated more reasonably by considering the overlapping values of those 2 columns than by using a default selectivity formula.

Selectivity estimates for LIKE predicates is limited to "abc%" patterns and very conservative estimation formulas.

The Optimizer can use a multicolumn histogram for a group of predicates when the following statements are all true:

- The predicates are specified on the first n fields of the multicolumn histogram.
This rule exists because there is an ordering dependency of the fields of a multicolumn histogram.
- The predicates must specify an equality condition except for the first field of the multicolumn histogram.
- If the predicate on the first column of a multicolumn is a non-equality condition, then the Optimizer uses the multicolumn histogram for this predicate only.

For example, a histogram on (x, y, z) can be used to estimate the selectivity for predicate $x > 100$ as well as $x = 10$ AND $y = 20$.

The Optimizer can also use the data derived by date extrapolation to enhance its cardinality estimates for date-related predicates (see [Using Extrapolation to Replace Stale Statistics](#)).

Combining Selectivities

The Optimizer can detect independent columns and calculate their combined selectivity as the product of their individual selectivities. The following categories of independent columns are defined:

- Every value of one column maps to all values of the other column.

In this case, the number of combined values is the product of the number of values of individual column. Take column nation and column market-segment of the customer table as an example.

Each nation participates in the business of all market segments, while business in each market segment is provided in all countries.

- The value of one column is not constrained by the value of the other column, but given a value of the first column of 2 independent columns, the possible value of the first column is evenly distributed across all values of the second column.

Column market-segment and column account-balance of the customer table are an example of such independent columns. On the one hand, the account balance is not bounded by a specific market segment even though one account balance, say \$100.23, does not exist in all market segments.

There are three cases where the Optimizer cannot detect column independence.

- When the total size of all columns exceeds the histogram data row size limit of 16 bytes. In this case, multicolumn histogram does not have all the information necessary to make an accurate determination of column independence, so the test for independence could fail.
- When independence detection is activated only when the confidence of the individual selectivities is high. That is, for predicates whose selectivity is not high confidence because no statistics have been collected on the columns, or because an expression is complex, the Optimizer does not attempt to detect column independence.
- When independence detection is activated only when the selectivity estimation of a predicate on an individual column is based on the base table histogram.

Finding the Number of Unique Values in a Set of Join or Grouping Columns

Join cardinality estimates, rows per value estimates, skew detection, aggregate estimates, Partial Group By estimates, and several other computations all require estimating the number of unique values for a given set of join or grouping columns.

The Optimizer makes an exhaustive search of all possible combinations of statistics to determine the best set of non-overlapping statistics. Once it has been decided, that set is then used to find the Number of Unique Values, High Modal Frequency, and High AMP Frequency values at all stages of the optimization process.

The major goals of this algorithm are as follows:

- Find the set of non-overlapping combinations of statistics that has the least number of unique values.

If no complete coverage can be found, find the set of non-overlapping combinations of statistics that covers the largest number of columns and has the least number of unique values.

- Find the set of non-overlapping combinations of statistics that provides the highest High Modal Frequency and High AMP Frequency values by covering all the columns.

The set of combinations that provides the smallest Number of Unique Values might not be the same set that provides the best estimates for the High Modal Frequency and High AMP Frequency values because the High Modal Frequency and High AMP Frequency might not be available for some of the entries that are derived from sources other than base table interval histograms.

The number of unique values determined from partial statistics, where statistics do not cover all the hash or join columns, is considered for Rows Per Value and join cardinality estimates because it is a conservative approach, but the Optimizer does not consider this estimate for skew adjustment and Partial GROUP BY because it makes very aggressive cost estimates and, as a result, can cause performance problems when the costing estimates are grossly in error because of the overly aggressive method by which they were calculated. In other words, the Optimizer avoids Partial GROUP BY plans and does not attempt to make skew adjustments when the given hash or join columns are not fully covered by statistics.

The unique values discovery algorithm provides the following information (see [EXPLAIN Confidence Levels](#) for the definitions of the various Optimizer confidence levels for making cardinality estimates):

- MinVals and its confidence level

This estimate provides the absolute minimum number of values for the given collection of columns. The values are taken from a single derived statistics entry that covers the largest number of columns. If there are multiple entries that cover the same number of columns, the Optimizer selects the entry with the highest number of values.

The confidence levels for various entries are described in the following table.

IF a usable derived statistics entry is ...	The confidence level is ...
found	High confidence
not found	No confidence

- BestVals and its confidence level

This provides the best number of values estimate that can be derived from the set of derived statistics entries that meets both of the following criteria.

- Covers the greatest number of columns
- Produces the least number of values

Derived statistics entries that are either the same set, or a subset, of the given column collection are used to produce these values.

The values can even be taken from a set of derived statistics entries that covers only a portion of the columns in some cases. This can happen when, for example, there are insufficient statistics to cover all the columns.

The confidence levels for various entries are described in the following table:

FOR this derived statistics entry situation ...	The confidence level is ...
a single entry covers all the columns	High confidence
either of the following. <ul style="list-style-type: none"> multiple entries must be combined to cover all the columns only a subset of the columns is covered 	Low confidence
no usable derived statistics entries are found	No confidence

- MaxVals and its confidence level

This estimate provides the maximum number of possible values for the given collection of columns. The derived statistics entries that are either a subset, a superset, an EquiSet, or intersecting entries are all considered for producing these values.

If all columns are not covered by these entries, then default estimates based on the domain type are used for the non-covered columns, as described in the following table.

Derived Statistics Entry Situation	Confidence Level
a single entry covers all the columns	High confidence
multiple entries must be combined to cover all the columns	Low confidence
default estimates are used to compute the values	No confidence

If MinVals is found with a High or Low level of confidence, then the Optimizer can always determine a BestVals statistic. However, it is also possible to determine MaxVals with High or Low confidence, but not be able to determine a BestVals or MinVals.

The Optimizer uses BestVals for its join cardinality and RowsPerValue estimates if it has Low or High confidence.

Partial GROUP BY, aggregations, and skew detection always use MaxVals.

The values are combined in 2 levels using a stochastic model, as follows:

1. The values from the same source are combined.

The number of combined values is limited to the total rows of the source.

2. The combined values from different sources are themselves combined to get the final values.

These are limited based the total rows for the current set.

The system consumes the EquiSet entries and then derives additional combinations if any multicolumn statistics intersect with the EquiSets. For example, if the usable entries for the given hash columns (x1, x2, y2) are EquiSet [x1, x2] and (x2, y2)], then the Optimizer augments the multicolumn statistics entry (x2, y2) with other EquiSet columns, producing the new entry (x1, x2, y2) dynamically.

The following examples describe the different possibilities.

For the first example, consider the following derived statistics entries:

Column Set	Number of Unique Values
(a1, b1)	10
(b1, c1)	15
(a1, b1, c1)	20

If the hash columns are (a1, b1, c1), then the Optimizer derives the following values and confidence levels:

Statistic	Value	Confidence Level
MinVals	20	High confidence
BestVals	20	High confidence
MaxVals	20	High confidence

For the second example, consider the following derived statistics entries:

Column Set	Number of Unique Values
(a1, b1)	10
(c1)	5

If the hash columns are (a1, b1, c1, d1), then the Optimizer derives the following values and confidence levels:

Statistic	Value	Confidence Level
MinVals	10	High confidence
BestVals	50 Calculated from the product of the numbers of unique values for the column sets: $10 \times 5 = 50$.	Low confidence
MaxVals	Combination of 50 and the demographic estimate for d1.	No confidence

For the last example, consider the following derived statistics entry:

Column Set	Number of Unique Values
(a1, b1, c1, d1)	100

If the hash columns are (a1, b1, c1), then the Optimizer derives the following values and confidence levels:

Statistic	Value	Confidence Level
MinVals	TotalRows	No confidence
BestVals	TotalRows	No confidence
MaxVals	100	Low confidence

Using Join Index Statistics to Estimate Single-Table Expression Cardinalities

This topic describes how the Optimizer can use statistics collected on complex expressions specified in the select list of a single-table join index to more accurately estimate the cardinalities of complex expressions specified on base table columns in a query predicate. The use of the term *join index* anywhere in this topic refers only to single-table join indexes or an equivalent hash index.

You should consider hash indexes to be equivalent to non-sparse single-table join indexes for this topic. Anything written about a non-sparse single-table join index applies equally to an equivalent hash index, substituting the term *column list* for *select list*.

Although you cannot collect statistics on complex base table expressions, creating a single-table join index that specifies the same expression in its select list transforms the expression into a simple join index column, and you can then collect statistics on that column. The Optimizer can use those statistics to estimate the single-table cardinality of evaluations of the expression when it is specified in a predicate of a query on the base table.

This capability extends the application of join indexes in query optimization to functions beyond simple query rewrite to include using their statistics for single-table expression cardinality estimation when statistics have not been, or cannot be, collected on complex expressions coded in predicate expressions that refer to their base table columns.

To provide the enhanced single-table expression cardinality estimation capability, the following criteria must be met:

- The join index from which the statistics are collected must specify the relevant complex expression in its select list.
- You must collect statistics on the join index column set that specifies the relevant complex expression.

Definitions of Simple and Complex Expressions

You can collect statistics on a simple join index column created from a complex expression that is specified in its select list. The Optimizer can use statistics collected on the join index column to estimate more accurately the selectivity of complex expressions specified in a query predicate that specifies the matching base table expression.

A simple expression is one with only a simple column reference on the left hand side of the predicate, while a complex expression is one that specifies something other than a simple expression on its left hand side.

For example, the following predicate specifies a simple column reference:

DATE = '2010 - 06 - 10'

The following predicate specifies a complex expression because the term on its left hand side is not a simple column reference:

EXTRACT(YEAR FROM DATE) = 2010

Not all complex expressions can benefit from this optimization. The Optimizer can only use statistics collected on a join index column derived from a complex expression if that expression can be mapped completely to a simple join index expression. The following predicate, for example, cannot be mapped completely to a simple join index expression, so it cannot take advantage of this optimization:

$$\frac{\text{EXTRACT (YEAR FROM DATE)}}{2} = 1005$$

This expression cannot be mapped to a simple join index expression on which statistics can be collected.

Matching Predicates and Mapping Expressions

There are several specific cases where join index statistics can provide more accurate cardinality estimates than are otherwise available for base table predicates written using complex date expressions.

- The case where an EXTRACT expression specified in a query predicate can be matched with a join index predicate.
- The case where an EXTRACT DATE expression specified in a query predicate condition can be mapped to an expression specified in the select list of a join index.

The Optimizer uses expression mapping when it detects an identical query expression or a matching query expression subset within a non-matching predicate. When this occurs, the Optimizer maps the predicate to the identical column of the join index, which enables it to use the statistics collected on the join index column to estimate the cardinality of the expression result.

Using Predicate Matching to Estimate Single-Table Cardinalities for Complex Expressions

This topic examines the case where an EXTRACT/DATE expression specified in a query predicate condition can be matched with an expression specified in the select list of a join index. If you have collected statistics on the matching join index column, the Optimizer can then use those statistics to estimate the cardinality of the predicate result.

For example, consider the following join index created on base table *t100_a*:

```
CREATE JOIN INDEX ji_a3 AS
  SELECT i1, c1
  FROM t100_a
  WHERE EXTRACT(YEAR FROM d2) >= 1969;
```

You then collect statistics on the default nonunique primary index for *ji_a3*, *ji_a3.i1*.

```
COLLECT STATISTICS ON ji_a3 INDEX (i1);
```

Join index *ji_a3* is designed to support the following queries on base table *t100_a*:

Query 1

```
SELECT *
FROM t100_a
WHERE EXTRACT(YEAR FROM d2)>=1969;
```

Query 2

```
SELECT *
FROM t100_a
WHERE EXTRACT(YEAR FROM d2)>=1969
AND i1>2;
```

Because *ji_a3* specifies a predicate (`EXTRACT(YEAR FROM d2)>=1969`) that is identical to the predicates specified in both queries 1 and 2, and query 2 specifies a predicate condition (`i1>2`) on a column you have collected statistics on for *ji_a3*, both queries can use the statistics collected on *ji.i1* to enhance the ability of the Optimizer to estimate the cardinalities of their predicate expressions.

The true cardinality for this example is 100 rows.

```
EXPLAIN SELECT *
FROM t100a
WHERE EXTRACT(YEAR FROM d2)>=1970;
```

The following shows a portion of the EXPLAIN output:

```
...
3) We do an all-AMPs RETRIEVE step from df2.t100_a by way of
   an all-rows scan with a condition of ("(EXTRACT(YEAR FROM
   (df2.t100_a.D2 )))= 1970") into Spool 1 (all_amps), which
   is built locally on the AMPs. The size of Spool 1 is estimated
   with high confidence to be 100 rows (49,800 bytes). The estimated
   time for this step is 0.03 seconds.
```

The Optimizer estimates the cardinality of Spool 1 (highlighted in boldface type), which is derived from *t_100a*, to be 100 rows, an exact match with the true cardinality.

Using Expression Mapping to Estimate Single-Table Cardinalities for Complex Expressions

This topic examines the case where an EXTRACT/DATE expression specified in a query predicate condition can be mapped to an expression specified in the select list of a join index. Mapping converts the EXTRACT expression to an expression written on a DATE column.

When the Optimizer makes a cardinality estimate by mapping from the base table to a join index, the normalization between the base table DATE column that contains the desired year value and the EXTRACT function in the join index definition applies, where join index statistics can provide more accurate cardinality estimates for single-table predicates written using complex expressions specified on the base table than are otherwise available.

The following example demonstrates the case where the query specifies a predicate on DATE column *d2*, and join index *ji_a4* specifies an EXTRACT function on the same DATE column as a simple join index column.

For this case, the Optimizer attempts to convert the predicate DATE condition to its equivalent EXTRACT form before mapping to the join index column. The Optimizer first transforms the query predicate condition *d2* >= '1969-01-01' to the equivalent expression EXTRACT(YEAR FROM *d2*) >= '1969' and then further transforms that expression to *ji.yr* >= 1969. Because *ji.yr* >= 1969 is a simple expression on a join index column, the Optimizer can use the statistics collected on column *ji.yr* to estimate the cardinality of the expression result.

Consider the following join index:

```
CREATE JOIN INDEX ji_a4 AS
  SELECT i1, EXTRACT(YEAR FROM d2) AS yr, c1
  FROM t100_a
  WHERE i1 < 30;
```

You then collect the following statistics on the index:

- Column *i1* is the default nonunique primary index for *ji_a4*.

```
COLLECT STATISTICS ON ji_a4 INDEX(i1);
```

- The expression alias *yr* represents the value of the function EXTRACT(YEAR FROM *d2*).

```
COLLECT STATISTICS ON ji_a4 COLUMN(yr);
```

The true cardinality for this example is 30 rows.

The following query specifies a DATE expression in its predicate that can be mapped to the EXTRACT function specified in the select list of join index *ji_a4* because *d2* >= '1969-01-01' is a matching subset of the *ji_a4* select list expression EXTRACT(YEAR FROM *d2*) AS *yr* within a non-matching predicate:

```
EXPLAIN SELECT i1
  FROM t100a
```

```
WHERE i1<30
AND   d2>='1969-01-01';
```

The following shows a portion of the EXPLAIN output:

```
...
3) We do an all-AMPs RETRIEVE step from df2.JI_A4 by way of
   an all-rows scan with a condition of ("df2.JI_A4.yr >=
   1969") into Spool 1 (all_amps), which is built locally on the AMPs.
   The size of Spool 1 is estimated with high confidence to be 30
   rows (960 bytes). The estimated time for this step is 0.03
   seconds.
```

The Optimizer estimates the cardinality of Spool 1 (highlighted in boldface type), which is derived from *t_100a*, to be 30 rows, an exact match with the true cardinality.

Join index *ji_a4* is defined to alias the complex expression `EXTRACT(YEAR FROM DATE)` as *yr=2010*. This definition enables the Optimizer to map the complex predicate expression `EXTRACT(YEAR FROM DATE)=2010` to the simple expression *yr=2010*, allowing statistics collected on *yr* to be used to estimate the single-table cardinality of a predicate expression written using base table column references.

But what if you were to code a query predicate expression such as `EXTRACT(YEAR FROM DATE)/2 = 1005`? The Optimizer can map this expression to the somewhat simpler expression *ji_a4.yr/2 = 1005*, but this mapping does not enable the use of join index statistics to estimate the single-table cardinality of the expression.

The reason this mapping cannot facilitate more accurate cardinality estimation is that while the mapped expression has a less complicated appearance, it remains a complex expression. Because of the complexity of the expression, the Optimizer cannot use statistics collected on *yr* to estimate the single-table cardinality of the predicate expression `EXTRACT(YEAR FROM DATE)/2=1005` whether it can be mapped to *ji_a4.yr/2=1005* or not.

The next example is a slightly more complicated example of expression mapping. This case uses statistics from matching complex expressions specified in the select list of a join index to an expression specified in a query predicate. This example applies to both sparse and non-sparse join indexes, though the join index used for this example is sparse.

Consider the following SELECT request:

```
SELECT *
FROM   perfcurnew
WHERE  BEGIN(vt) <= CURRENT_DATE
AND    END(vt)   >  CURRENT_DATE
AND    END(tt)   IS UNTIL_CHANGED;
```

Vantage does not support collecting statistics from *perfcurnew* on the complex expressions specified in this query predicate that can be used to estimate the cardinality of the result set.

But suppose you create the following join index on *perfcurnew*. The select list of this join index specifies expressions that are components of the query predicate written against the base table *perfcurnew* in the example SELECT request.

```
CREATE JOIN INDEX ji, NO FALLBACK, CHECKSUM = DEFAULT AS
  SELECT i, j, BEGIN(vt)(AS bvt), END(vt)(AS evt), END(tt)(AS ett)
  FROM perfcurnew
  WHERE (evt>DATE)
  AND   END(tt) IS UNTIL_CHANGED
  PRIMARY INDEX (i);
```

After creating join index *ji*, you can collect the following statistics to support the SELECT request on *perfcurnew*:

- Column *i* is the nonunique primary index for *ji*.

```
COLLECT STATISTICS ON ji INDEX(i);
```

- Column *vt* is defined in base table *perfcurnew* with a Period type. The expressions aliased as *bvt* and *evt* represent the BEGIN and END bound functions, respectively, of the Period column *perfcurnew.vt*, and you can collect statistics on them as they are defined in the join index as *ji.bvt* and *ji.evt*.

```
COLLECT STATISTICS ON ji COLUMN(bvt);
COLLECT STATISTICS ON ji COLUMN(evt);
```

- Column *tt* is also defined in *perfcurnew* with a Period type, and you can collect statistics on the END bound function of the Period column *ji.tt*, aliased as *ett*.

```
COLLECT STATISTICS ON ji COLUMN(ett);
```

The true cardinality of the base table *perfcurnew* for this example is 691 rows.

The SELECT request in the following example specifies predicates that use the CURRENT_DATE function and the IS UNTIL_CHANGED predicate variable. IS UNTIL_CHANGED represents a “forever” or “until changed” date value for the END Period bound function specified for the Period column *perfcurnew.tt*.

Join index *ji* is defined using a subset of the predicates specified in the SELECT request, so the cardinality estimate for that predicate represents the number of rows selected by the predicates specified on END(vt) and END(tt).

The Optimizer maps the predicate BEGIN(vt) <= CURRENT_DATE - 2000 to the expression *ji.bvt* <= CURRENT_DATE - 2000 and uses the statistics collected on *ji.bvt*.

The combined cardinality estimate made from these statistics represents an evaluation of the cardinality of the result returned by all three of the predicates specified in the original SELECT request.

In this case, the predicates in the query are the same as the predicates specified in the select list of *ji*, and the cardinality of *ji* is the same as the cardinality of the result set of the query, so statistics collected on *ji* are not required to make the cardinality estimate.

```
EXPLAIN SELECT *
FROM perfcurnnew
WHERE BEGIN(vt) <= CURRENT_DATE - 2000
AND   END(vt)   > CURRENT_DATE
AND   END(tt)   IS UNTIL_CHANGED;
```

The following shows a portion of the EXPLAIN output:

```
...
3) We do an all-AMPs RETRIEVE step from a single partition of
   df2.perfcurnnew with a condition of (
   "((BEGIN(df2.perfcurnnew.vt ))<= DATE '2004-11-24') AND
   (((END(df2.perfcurnnew.tt ))= TIMESTAMP '9999-12-31
   23:59:59.999999+00:00') AND ((END(df2.perfcurnnew.vt ))> DATE
   '2010-05-17')))" into Spool 1 (all_amps), which is built locally
   on the AMPs. The size of Spool 1 is estimated with high
   confidence to be 723 rows (161,952 bytes). The estimated time for
   this step is 0.07 seconds.
```

The estimated cardinality of Spool 1 (highlighted in boldface type), which is derived from *perfcurnnew*, is 723 rows, which errs by only 32 rows.

Using a Unique Join Index in the Access Path for a Query

When multiple unique join indexes are defined on a single table, the Optimizer can use any of them for an access path. Any unique join indexes defined on a single table compete with one other for access path selection based on their costs, and the index that is determined to have the lowest cost when used as an access path is selected for the query plan by the Access Path Planner.

You should always collect statistics on the primary index of a unique join index, whether it is user-defined or system-defined, so the Optimizer can use those statistics to most accurately estimate the cost of using a particular unique join index.

The Optimizer can use the cardinality of a unique join index to match the base table predicates specified in a query, and cardinality estimates made using that method are reported with high confidence (see [EXPLAIN Confidence Levels](#) and [Using Join Index Statistics to Estimate Single-Table Expression Cardinalities](#)).

Unique Join Index Subsets and Access Path Selection

The Optimizer can use a unique join index such as a USI as an access path for single-row access in 2-AMP plans if the query specifies an equality condition on the columns that are defined as the primary index for the unique join index. The value in the equality condition forms the primary index key for accessing the unique join index.

For a unique join index whose primary index is defined explicitly as unique, as it is the case for user-defined unique join indexes, the coverage testing and the presence of an equality condition on the primary index

column set of the join index guarantees that the join index returns at most one row and thus is sufficient to qualify the join index to be used for single-row access.

For system-defined unique join index, coverage testing is based on the VALIDTIME and TRANSACTIONTIME conditions that the join index is defined with as well as the VALIDTIME and TRANSACTIONTIME conditions in the query. Vantage adds VALIDTIME and TRANSACTIONTIME conditions to a query that specifies one or more temporal qualifiers.

You can also specify additional VALIDTIME and TRANSACTIONTIME conditions in the query to further restrict the result set. All VALIDTIME and TRANSACTIONTIME conditions found in the query are used for coverage testing.

If one user-defined unique join index is a subset of another, and both are in contention for being selected for the access path for a request, the Optimizer selects the index that has fewer columns because it can hash on a smaller key.

For example, suppose you create the following 2 unique join indexes on *t1*:

```
CREATE JOIN INDEX t1_ji1 AS
SELECT a1, ROWID
FROM t1
UNIQUE PRIMARY INDEX (a1);
CREATE JOIN INDEX t1_ji2 AS
SELECT a1, b1, ROWID
FROM t1
UNIQUE PRIMARY INDEX (a1,b1);
```

You then submit the following SELECT request against *t1*.

```
SELECT c1
FROM t1
WHERE a1=10
AND b1= 5;
```

Both *t1_ji1* and *t1_ji2* can be used as access paths for this request, but the Optimizer selects *t1_ji1* because it has fewer columns.

However, for a system-defined join index, the Optimizer cannot use the fewest columns heuristic because for the same value on the primary index, there can be multiple rows in the join index, and the Optimizer must consider the overall costing, including the number of rows per value and the size of the table.

Unique Join Indexes as an Access Path in Two-AMP Query Plans

You can define both a non-compressed and a non-value-ordered single-table join index with a unique primary index.

The following are the valid forms of unique join index:

- A user-defined unique join index, which is a non-compressed single-table join index that is defined with a WHERE clause, contains base table ROWID in its select list and is defined with a unique primary index. Furthermore, the unique join index covers the query.

In this case, coverage means the WHERE clause of the unique join index qualifies a row set that is a super set of the row set qualified by the WHERE clause of the query against the same table.

- A system-defined unique join index, which is a system-defined non-compressed single-table join index used to enforce a temporal UNIQUE constraint, and the system-defined join index covers the query.

In this case, coverage also refers to the WHERE clause coverage with an additional requirement that the table is queried using a temporal qualifier. For information about temporal tables, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

A unique join index can be used like a USI as an access path for single-row access in 2-AMP plans if the query specifies an equality condition on the columns that are defined as the UPI for the unique join index. The value in the equality condition forms the primary index key for accessing the unique join index.

For a unique join index whose primary index is defined as unique explicitly, as it is the case for user-defined unique join index, the coverage testing and the presence of an equality condition on the join index primary index column set guarantees the join index to return at most one row, and thus is sufficient to qualify the join index to be used for single-row access.

For system-defined unique join indexes, coverage testing is based on the VALIDTIME and TRANSACTIONTIME conditions that define the join index as well as the VALIDTIME and TRANSACTIONTIME conditions in the query. See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for the definitions and use of these terms.

Vantage adds VALIDTIME and TRANSACTIONTIME conditions to a query that specifies temporal qualifies (see *Teradata Vantage™ - Temporal Table Support*, B035-1182 for details). You can also specify additional VALIDTIME and TRANSACTIONTIME conditions in the query to further restrict the result set.

The Optimizer uses all of the VALIDTIME and TRANSACTIONTIME conditions you specify in a query for coverage testing.

For example, for a CURRENT UNIQUE constraint defined on the TRANSACTIONTIME dimension of the Transaction Time table *part_types*, which is defined as follows:

```
CREATE MULTISET TABLE part_types (
  part_id      INTEGER NOT NULL,
  part_name    VARCHAR(20),
  part_type    CHARACTER(2)
  part_duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL AS
              TRANSACTIONTIME,
  CURRENT TRANSACTIONTIME UNIQUE (part_id))
PRIMARY INDEX (part_name);
```

Vantage defines a system-defined join index for *part_types* as follows:

```
CREATE SYSTEM_DEFINED JOIN INDEX part_types_TJI004 AS
  CURRENT TRANSACTIONTIME SELECT ROWID, part_id, part_duration
  FROM part_types
  UNIQUE PRIMARY INDEX (part_id);
```

The Optimizer adds the following condition to a request that queries the same table using an AS OF CURRENT_DATE qualifier:

```
BEGIN (part_id)<=CURRENT_DATE AND
END   (part_id)> CURRENT_DATE
```

Because the system-defined join index has a TRANSACTIONTIME condition that qualifies only Open rows to be stored in the join index, the index cannot cover the AS OF query by default. However, if the AS OF query explicitly specifies a TRANSACTIONTIME condition to qualify Open rows, then the join index can cover the query.

If a system-defined join index is used to enforce a unique constraint in the Transaction Time dimension only, an equality condition on the primary index column set of a system-defined join index is enough to guarantee that the join index returns at most one row.

However, if a system-defined join index is used to enforce a unique constraint in the Valid Time dimension, the index can contain multiple rows with the same value in *column_list* as long as they do not overlap the *validtime_column*. This means an equality condition on *column_list* with a unique constraint in the Valid Time dimension returns at most one row only if the table is queried for a single point in the Valid Time dimension. That is, the table is queried with a CURRENT or an AS OF temporal qualifier.

The conditions the Optimizer adds to a query for the CURRENT or AS OF qualifiers must also be evaluated as residual conditions on the system-defined join index to qualify the row that corresponds to that point in time. Furthermore, the system-defined join index must not have a partitioned primary index to avoid having to search multiple row partitions for the qualified row.

Therefore, in order for a system-defined join index that enforces a unique constraint in the Valid Time dimension to be usable for single-row access, the following things must be true:

- The system-defined join index must not be defined with a PPI.
- The system-defined join index must pass coverage testing.
- The query must specify either a CURRENT or an AS OF temporal qualifier on the table.
- The query must specify an equality condition on the columns that are defined as the primary index of the join index.

The following table summarizes when a system-defined join index qualifies for access path usage given that the coverage testing passes and the query specifies an equality condition on the primary index column set of the join index:

DML Qualifier	Temporal Constraint Time	Constraint Type	Does a System-Defined Unique Join Index Qualify for Use in the Query Access Path?
CURRENT	VALIDTIME	CURRENT UNIQUE	Yes
		SEQUENCED UNIQUE	Yes
	TRANSACTIONTIME	CURRENT UNIQUE	Yes
AS OF <i>date_expression</i> <i>timestamp_expression</i>	VALIDTIME	CURRENT UNIQUE	Yes if <i>date_expression</i> <i>timestamp_expression</i> >= resolved value of the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP for the single-table unique join index.
		SEQUENCED UNIQUE	Yes
	TRANSACTIONTIME	CURRENT UNIQUE	Yes if the TRANSACTIONTIME for the query is UNTIL_CLOSED
SEQUENCED [<i>period_of_applicability</i>]	VALIDTIME	CURRENT UNIQUE	No
		SEQUENCED UNIQUE	No
	TRANSACTIONTIME	CURRENT UNIQUE	Yes if the TRANSACTIONTIME for the query is UNTIL_CLOSED

Because the use of a unique join index as an access path is limited to queries with an equality condition on the primary index columns, and a 2-AMP step requires a read of the base table for the qualified RowId, it is possible to find a more optimal plan using the same join index as a covering join index. The Query Rewrite subsystem can also use a unique join index as a regular join index for query rewrites. This means that a unique join index can play a dual role in query optimization.

- It can be used as a regular join index and as either a partially covering or fully covering join index to rewrite the query.
- If the query is not rewritten to use the unique join index, the unique join index can be treated as a unique index by the Access Path Planner.

The Optimizer can use a unique join index as an access path if there is an equality condition on the base table columns that correspond to the primary index for the join index.

Consider the following example:

```
CREATE TABLE t1 (
  a1 INTEGER,
```

```

b1 INTEGER,
c1 INTEGER);
CREATE JOIN INDEX uji AS
  SELECT b1, c1, ROWID
  FROM t1
  WHERE c1 BETWEEN 200 AND 1000
  UNIQUE PRIMARY INDEX (b1);

```

Suppose you submit the following SELECT request against *t1*:

```

SELECT b1, c1
FROM t1
WHERE b1=10
AND   c1=500;

```

The Optimizer evaluates the following plans for this SELECT request:

- A plan that does a single-AMP retrieval from *uji*.
This plan results from the join index rewrite that replaces *t1* in the request with *uji*.
- A plan that does a 2-AMP retrieval using *uji*.
This plan results from using *uji* as an access path.

As is always true, the Optimizer selects the plan with the lower cost.

Estimating Join Cardinality With Single-Row Unique Index Access to One of the Tables

About Estimating Join Cardinality With Single-Row Unique Index Access to One of the Tables

If one of the source tables of a join has single-row access using a unique index, the Optimizer retrieves the row during the optimization phase, substitutes the values of the columns from the retrieved row, and makes its cardinality estimates based on the substitution.

For example, if a join condition has a form like `fact_table.yr_wk BETWEEN calendar.ytd_beg_wk AND calendar.ytd_end_wk`, and the calendar table has single-row access path using either a UPI or a USI, the Optimizer fetches the row during the optimization phase, substitutes the actual values from the retrieved row for `ytd_beg_wk` and `ytd_end_wk`, and estimates the cardinality of the join based on the substitution.

```

EXPLAIN
SELECT *
FROM ordertbl, calendar
WHERE yr_wk BETWEEN ytd_beg_wk AND ytd_end_wk  <-- join predicate
AND   calendar_date = 970101;    <-- UPI access to calendar table

```

...

- 3) We do a single-AMP RETRIEVE step from TPCD_OCES3.calendar by way of the unique primary index "TPCD_OCES3.calendar.calendar_date = DATE '1997-01-01'" with no residual conditions into Spool 2 (all_amps), which is duplicated on all AMPs. The size of Spool 2 is estimated with high confidence to be 14 rows (1,134 bytes). The estimated time for this step is 0.01 seconds.
- 4) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an all-rows scan, which is joined to TPCD_OCES3.order_75pct by way of an all-rows scan with no residual conditions. Spool 2 and TPCD_OCES3.order_75pct are joined using a product join, with a join condition of "(TPCD_OCES3.ordertbl.YR_WK >= Ytd_Beg_wk) AND (TPCD_OCES3.ordertbl.YR_WK <= Ytd_End_wk)". The input table TPCD_OCES3.ordertbl will not be cached in memory, but it is eligible for synchronized scanning. The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 365,114 rows (140,934,004 bytes). The estimated time for this step is 2.55 seconds.

As you can see in step 4 of the query plan, the cardinality estimate for the range join predicate is computed by substituting the values from the row of the calendar table that was retrieved in step 3.

Stale Statistics

Stale statistics are interval histogram statistics that no longer represent an accurate description of the column sets on which they were originally collected.

Interval histogram statistics are the initial source of column demographic information for the Optimizer to use in making its estimates of join cardinality, rows per value, skew adjustments, and several other pieces of information necessary for accurate query optimization.

The Optimizer also employs more dynamic methods of using statistics such as deriving column correlations using multicolumn statistics, adjusting and propagating the number of unique values after each binary join is made, and so on. The methods are developed and generalized in a consistent manner throughout the Optimizer.

Cost and cardinality estimation formulas and the strategies for deriving column correlations depend on the accuracy of the available statistics to produce optimal plans. The software might not produce reasonable plans if its statistical inputs are not accurate. The most common reason statistics are not accurate is because they are stale.

The process of detecting stale statistics and extrapolating data values (see [Using Extrapolation to Replace Stale Statistics](#)) when the Optimizer determines that the statistics it needs are stale is designed to reduce the frequency of collecting or recollecting statistics.

Detecting Stale Statistics

The estimated cardinality of a table is critical information that Vantage needs to detect stale histograms and to activate extrapolation.

Every histogram contains a cardinality value that is based on the number of rows in the table at the time the statistics were collected. The database compares the histogram row count with the newly calculated estimate of the table cardinality to determine whether a histogram is stale. If the Optimizer determines that a histogram is stale, it applies extrapolation techniques when doing selectivity estimations.

Using Derived Statistics to Compensate for Stale Statistics

You can use one or both of the following methods to enable the database to recollect statistics only when a limit that you specify with a COLLECT STATISTICS request is exceeded. For details, see [Optimal Times to Collect or Recollect Statistics](#) and [Optimizing the Recollection of Statistics](#).

For further details, see the information about COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- Set a recollection threshold

The THRESHOLD option enables you to instruct the database to skip recollecting statistics if either the change in the data from the last statistics collection or the age of the statistics is below the thresholds you specified when you initially collected the statistics.

For more information, see [Optimizing Statistics Recollection Using Threshold Options](#).

- Collect sampled, rather than full, statistics

The following methods exist for determining sampling defaults:

- You can specify the sampling percentage for the database to use.
- You can enable the database to determine the sampling percentage to use based on a set of criteria that it chooses.

When the database determines the sampling percentage, it uses a method of downgrading statistics collection from a 100% sampling to a sampling rate it selects to optimize recollections for a particular column or index set.

For more information, see [Optimizing Statistics Recollection Using Sampling](#).

You can specify the sampling percentage for the database to use to determine sampling defaults.

There are four principal ways the cardinality estimation subsystem handles the problem of stale statistics:

1. Adjusting the total cardinality

The Optimizer uses table-level SUMMARY statistics to determine the growth of a table. To do this, it compares the cardinality estimate from the current dynamic AMP sample against the cardinality from the residual table-level statistics. The Optimizer then adds the estimated growth to the cardinality estimate from the SUMMARY statistics to estimate the current cardinality of the table.

- If the primary index is not skewed and if the deviation between the sample and the interval histogram row count is more than the default percentage, the sample row count is taken as the

cardinality of the table. The skew can be identified from the high modal frequency value stored in the interval histogram.

- If no SUMMARY statistics are available, the sampled row count is taken to be the table cardinality as is done by the current Optimizer software.

2. Adjusting the number of unique values.

For indexes, the cardinality is estimated as the total number of unique values of the indexed columns. In this case, the number of unique values is adjusted in the same way as for adjusting total cardinality.

If the adjusted table row count deviates more than a defined percentage from the histogram row count on unindexed columns, then the number of unique values, which is the principal value input to the join cardinality and costing related formulas, is, assuming a uniform distribution, either scaled up or down for the following scenarios:

- The histogram is on a DATE column.
- The histogram is on multiple columns that have a DATE column component.
- The histogram is on a unique index.
- The histogram is on a *soft unique* column. Soft uniqueness is defined as the case where the interval histogram statistics indicate that the number of unique values is close to the total cardinality for the table.

The scaling operation assumes a constant number of rows per value. For example, assume that the table cardinality is estimated to be 100 rows from a dynamic AMP sample. If, for a column x1, the histogram indicates that the row count is 50 and the number of unique values is 25, meaning the number

of Rows Per Value is 2, then the number of unique values is scaled up to $\frac{\text{Cardinality}}{\text{Average rows per value}} = \frac{100}{2} = 50$.

Note that the value scaling, whether up or down, cannot be done unconditionally for all columns.

For example, columns like product_id, business_unit_id, and so on cannot have new adjusted values added to them because it makes no semantic sense. Their values are not only arbitrary, but also are not cardinal numbers, so it is meaningless to perform mathematical manipulations on them, particularly in this context.

3. Considering statistics collection timestamps

While inheriting the statistics from the single-table non-sparse join or hash index by the parent base table and vice versa, when statistics are available for a column in both the source and destination interval histograms, the system uses the most recent statistics of the 2 as determined by a timestamp comparison.

4. Adjusting the confidence level

If the available histogram was created from sampled statistics, then the system always lowers the their confidence level.

You should always keep the statistics for skewed columns and indexes current.

The handling of stale statistics for single-table cardinality estimates is done using various extrapolation methods.

Object Use and UDI Counts

About Object Use Counts

The purpose of collecting object use counts is to maximize the system management capabilities by providing an efficient way for DBAs to determine the usage frequencies of various database objects. Object use counts record information about how often database objects such as tables, views, macros, columns, indexes, and statistics are used at the system level. This includes the total access count for update, delete, and insert operations (collectively known as UDI counts) and the last access timestamp. This information is collectively referred to as object use data, or OUD.

Unlike the DBQL feature, object use count data measure system-level object use, while DBQL records similar object-related data at the request level.

Object use counts track the following types of information:

- Database accesses
- Table accesses
- Column accesses
- Index accesses

This includes all of the following index accesses:

- Primary
- Secondary
- Join
- Hash
- Delete, insert, and update accesses
- Types of usage associated with each database object access

This includes all of the following accesses:

- Macros
- Views
- User-defined procedures
- Triggers
- User-defined functions
- User-defined methods
- User-defined types
- Statistics usage

The Optimizer uses object use counts in `DBC.ObjectUsage` to track the access usage of various database objects. Vantage tracks two versions of use counts: user and system. The two counts are identical, with the only differentiating characteristic being the ability for Vantage to reset user-level use counts by users and to reset system-level use counts.

The following list defines the similarities and differences between user object use counts and system object use counts:

- You can reset user object use counts using the *DBC.ClearUserUseCount* and *DBC.ClearUserStatCount* system macros, and you can reset these use counts at self-determined intervals. See *Teradata Vantage™ - Data Dictionary*, B035-1092 for more information about these macros.

This type of information is useful within the context of system monitoring, analysis of system performance, and tuning of frequently used objects.

- Only Vantage can reset system object use counts. The reset interval depends on the optimizations for which these counts are used.

System object use counts are useful within the context of optimizing system database objects, such as statistics. The ability to accurately track the usage of statistics is critical for prioritizing the ability of the THRESHOLD option of the COLLECT STATISTICS statement to accurately recollect statistics.

About UDI Counts

UDI counts are an important level of object use counts tracked by Vantage. Effective interpretation of the THRESHOLD option for COLLECT STATISTICS requires a fine level of tracking granularity, so Vantage tracks the number of update, delete, and insert operations performed against database objects in DBC.ObjectUsage. These sums are referred to as UDI counts. Using the last reset timestamp, UDI counts provide an accurate measure of the changes in the cardinalities of tables.

This type of information is useful for internal database optimizations, for DBAs, and for application users. As is true for object use counts, Vantage tracks two versions of UDI counts: user and system. The two counts are identical, with the only differentiating characteristic being the ability to reset UDI counts by either users or by Vantage.

The collection of UDI counts is disabled by default, and you can only enable their collection by specifying the USECOUNT option for a database as part of a BEGIN QUERY LOGGING request. For details, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Vantage logs UDI counts for the following database objects:

- Tables

This does not include counts for the following table types:

- Data Dictionary
- Error
- Log
- Queue
- Join indexes
- Hash indexes

UDI counts are logged for the following SQL DML statements and accesses:

- DELETE

- INSERT
- MERGE (including merge, merge-delete, and merge-update operations)
- SELECT
- UPDATE (including the Upsert form of UPDATE)
- General

This category includes accesses by DML-invoked operations such as user-defined functions and user-defined procedures.

- Statistics

Only Vantage can reset system UDI counts.

Vantage uses system UDI counts to estimate table cardinalities, to extrapolate statistics, and to track thresholds for recollecting statistics.

Optimizer Use of UDI Counts

The Optimizer uses UDI counts to estimate table cardinalities and when it applies THRESHOLD options determined when statistics are initially collected.

To estimate table cardinalities, the Optimizer retrieves insert and delete counts from *DBC.ObjectUsage* and then applies them to cardinality estimation.

Whenever statistics are collected or recollecting, the Optimizer draws on UDI counts to apply the THRESHOLD options specified for the original COLLECT STATISTICS request. Vantage first retrieves the delete and insert UDI counts from the master record for the table and the update UDI counts from the column and then combines them with the latest UDI counts from *DBC.ObjectUsage* to make a final cardinality estimate.

Accuracy of Object Use Counts

The accuracy of object use counts must be acute to ensure that the Optimizer can make effective decisions about query optimization and the recollection of statistics. There are two common scenarios in which use count inaccuracies can occur.

Type of use count inaccuracy	Explanation
Underreporting	<p>Use counts can be underreported when they are not collected because of system restarts.</p> <p>When a restart occurs, use count information in the memory-resident OUC cache is lost and not written to disk. As a result, the Optimizer can make poor decisions such as not recollecting statistics when current statistics are not recognized as being stale.</p> <p>To lessen the risk of underreporting use counts, the OUC cache also uses a persistent memory segment that lives across restarts, making use count information that might otherwise have been lost kept intact, and the collection of use count data can resume without interruption.</p>
Overreporting	<p>Use counts can be overreported when they are collected and then retained after a transaction abort.</p>

Type of use count inaccuracy	Explanation
	<p>When an abort occurs, use count information in the memory-resident OUC cache does not get spoiled because the object use count collection operates independently of any query or transaction logic.</p> <p>This presents a risk of overreporting counts because information is committed to disk when it should not have been. The Optimizer can make unnecessary decisions such as recollecting statistics unnecessarily based on these overreported counts.</p>

Optimizing the Recollection of Statistics

About Optimizing the Recollection of Statistics

As was mentioned briefly in the topic [Using Derived Statistics to Compensate for Stale Statistics](#), the COLLECT STATISTICS SQL statement supports two methods of restricting the database from recollecting statistics when there is no reason to do so (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for details about these methods).

To accomplish this, you can specify to collect sampled, rather than full, statistics.

The following methods exist for determining sampling defaults:

- You can specify the sampling percentage for the database to use.
- You can enable the database to determine the sampling percentage to use based on a set of criteria that it chooses.

When the database determines the sampling percentage, it uses a method of downgrading statistics collection from a 100% sampling to a sampling rate it selects to optimize recollections for a particular column or index set.

- Set a recollection threshold
- Specify to collect sampled, rather than full, statistics

Two methods exist for determining sampling defaults.

- You can specify the sampling percentage for the database to use.
- You can enable the database to determine the sampling percentage to use based on a set of criteria that it chooses.

When the database determines the sampling percentage, it uses a method of downgrading statistics collection from a 100% sampling to a sampling rate it selects to optimize recollections for a particular column or index set.

Optimizing Statistics Recollection Using Threshold Options

The COLLECT STATISTICS (Optimizer Form) statement supports several options to specify thresholds for recollecting statistics. These options enable you to specify various criteria that the database can use to determine whether it should ignore a request you submit to recollect statistics on a column or index set.

The THRESHOLD options enable you to specify statistics recollection criteria based on the following measures:

- Percentage of change in the data, or change-based recollection.
The criterion used is based on how much the data has changed since statistics were last collected, using the percentage of change since the last recollection of statistics for the specified column or index set as the threshold.
- Age, or time-based recollection.
The criterion used is based on the age of the statistics, using the number of days that have passed since the last recollection of statistics for the specified column or index set as the threshold.

When you submit a COLLECT STATISTICS request that specifies a threshold not to recollect statistics if the data change from the last statistics collection or if the age of the statistics is below the specified thresholds for the recollection of those statistics.

You can specify that the statistics be collected using one or both of the following criteria:

- Enabling the database to determine appropriate thresholds.
When you specify a system-determined threshold option, the Optimizer automatically determine the appropriate thresholds to use based on previously collected statistics, column or index historical data, UDI counts, and other factors.
- Specifying your own user-determined thresholds.
When you specify a threshold, you do so by specifying a change percentage, an aging criterion based on a number of days, or both.

By specifying thresholds for recollecting statistics, you eliminate the responsibility and burden of determining when to refresh statistics. When you specify threshold options, you can then submit COLLECT STATISTICS requests at regular intervals, and the Optimizer can determine intelligently whether to skip recollecting specified statistics whose thresholds are not met, to recollect statistics for those columns or indexes that are over the threshold, or both.

For example, if either you or the database determines the threshold to be a 10% data change, and you submit a request to recollect statistics on a column or index set, the Optimizer does not recollect those statistics if the change in the data since the last collection of statistics is less than 10%.

If you decide that you want to recollect a set of statistics regardless of the specified thresholds, you can submit your COLLECT STATISTICS request and specify the keyword phrase FOR CURRENT with the option set you want to override. In this case, the database ignores the saved thresholds for the current recollection only and returns to the saved thresholds the next time you recollect statistics for the specified column or index set.

If you specify one or more thresholds the first time you collect statistics on a column or index set, the Optimizer collects the statistics and saves the threshold options that you specify. When you recollect statistics on a column or index set, the database uses the saved threshold options to determine whether the current statistics recollection request should be ignored or obeyed.

If you specify a new threshold for recollecting statistics, the new threshold overrides the previously existing threshold, and the database applies the specified threshold for the current request and stores it for future recollections of the specified statistics.

As was mentioned earlier, you can use the following general categories of thresholds to control the recollection of statistics:

- Change-based methods
- Time-based methods

For change-based thresholds, the Optimizer consults the system-maintained UDI counts, random AMP samples, and any available column or index history to determine the extent of data change from the last statistics collection.

If you specify either the `SYSTEM THRESHOLD` or `SYSTEM THRESHOLD PERCENT` options for your `COLLECT STATISTICS` requests, the Optimizer considers column and index histories and extrapolations methods to determine the appropriate change threshold.

See [Using Extrapolation to Replace Stale Statistics](#) for information about the various extrapolations the Optimizer can use.

The Optimizer also evaluates information such as small table NUSI statistics to determine whether to recollect the requested statistics or not. For such small database objects, the Optimizer tends to collect full statistics because collecting full statistics provides up-to-date statistics for a small cost.

Time-based thresholds cause the Optimizer to evaluate the age of current statistics to determine whether they need to be refreshed or not.

Of course, you can also specify various combinations of change-based and time-based thresholds in addition to various sampling options to reduce the cost of recollecting statistics for a column or index set.

Using the thresholds approach enables you to submit requests to recollect statistics regularly without needing to determine whether it is too soon to do so because if the specified recollection thresholds are not met, the database avoids recollecting the specified statistics. By not recollecting unnecessary statistics, the database can avoid wasting costly CPU and I/O resources to recollect statistics unnecessarily. And as mentioned earlier, by using system-determined thresholds, the Optimizer can determine automatically whether a recollection of statistics is required or whether extrapolation is an adequate substitute method of estimating the statistics values for the specified column or index set.

You can query the Data Dictionary table column `DBC.StatsTbl.LastCollectTimeStamp` to determine whether your `COLLECT STATISTICS` requests to recollect statistics have been recognized or skipped. See *Teradata Vantage™ - Data Dictionary*, B035-1092 for more information about `DBC.StatsTbl` and the system views you can use to access the table.

Optimizing Statistics Recollection Using Sampling

You can collect sampled statistics when you submit a `COLLECT STATISTICS` request using either a system-determined sampling percentage or a user-specified sampling percentage.

If you specify a system-determined sampling percentage, the database uses a downgrade approach to determine the appropriate time to switch from collecting full statistics to sampling. With this approach, the

Optimizer determines when to honor sampling and to what extent, up to collecting full statistics, without sacrificing the quality. The downgrade approach works as follows:

1. When you first submit a COLLECT STATISTICS request that specifies sampling, the Optimizer initially collects full statistics. Collecting full statistics provides the Optimizer with enough information to determine when it can downgrade to a smaller percentage.

This is unlike first collecting a small sample and trying to determine if a larger sample would be more appropriate.

2. On subsequent requests to recollect the same statistics, the Optimizer continues to collect full statistics until it has collected enough statistics to capture an adequate statistical history.
3. Once it has collected an adequate statistics history, the Optimizer can recognize the nature of the column or index, whether it is skewed, rolling, or static, and so on.

The Optimizer then considers the column usage from either detailed buckets or merely summary data that it maintains in the *DBC.StatsTbl.UsageType* column and the user-specified number of intervals to determine the appropriate time to downgrade from collecting full statistics to collecting sampled statistics.

Vantage is more aggressive in downgrading from collecting full statistics to collecting sampled statistics for the histograms whose summary data is used but not the detailed intervals.

4. The Optimizer then determines a sample percentage (2% to 100%) for recollecting statistics and automatically determines the appropriate formula to use for scaling based on the history and nature of the column.
5. The recollected statistics are compared to the history for quality. The percentage is lowered only if the Optimizer can determine safely that a smaller percentage provides quality results.

This approach removes your responsibility and burden to make a decision on which columns to collect sampled statistics. Moreover, by collecting full statistics initially, it provides the Optimizer with information needed for making an intelligent decision about how much to sample in subsequent recollections.

For example, for small tables, skewed columns, column that is member of the partitioning expression column set and columns that require detailed buckets, the Optimizer never switches to sample statistics at less than 100%.

As another example, the Optimizer can detect very nonunique columns in the full statistics and, in subsequent recollects, it can intelligently switch to sampled statistics at a much lower percentage than 100%, perhaps even as low as 2%, apply the appropriate scaling formula applicable for nonunique columns to obtain the extrapolated full statistics, and obtain quality statistics much faster than with full statistics.

For rolling, unique, and near-unique columns as detected in the full statistics, the Optimizer can subsequently collect sample statistics at a much lower percentage and apply a linear scaling formula to produce the extrapolated full statistics.

After only a few sample statistics collections (the number is based on the demographic pattern and UDI counts of the data), the Optimizer can decide to recollect statistics using a 100% statistics collection to verify and adjust as needed the sample percentage and the extrapolation method used.

Sampled statistics are not supported for single-table views.

You can query the dictionary table column `DBC.StatsTbl.SampleSizePct` to determine the sample percentage used by the database to collect the statistics. For a user-specified sample percentage, the actual sample size used to build the histogram might be the same, slightly higher, or slightly lower than the specified sampling percentage.

Using Extrapolation to Replace Stale Statistics

Extrapolating Statistics When Histogram Statistics Are Stale

Users often query their databases over date intervals for which one or both bounds on a predicate are dates for rows that have been inserted into a table since the last time statistics were collected for the date columns specified for that table in the predicate.

Extrapolating statistics enables you to submit, for example, a query specifying a date range predicate in which one or all of the specified dates is past the latest date stored in the statistical histograms for the DATE column set. To support such queries, the Optimizer applies an extrapolation technique to make a reasonable estimate for the rows that have been inserted after statistics were last collected without requiring them to be recollected. Extrapolation methods for statistics can be viewed as a form of derived statistics (see [Derived Statistics](#)).

In this context, the definition of a future date applies only to dates occurring between the time statistics were last collected on a DATE column and the current date. In other words, *future* is a narrowly defined, relative term, and extrapolation does not apply to data having true future dates that cannot exist at the time you submit your date-extrapolation-requiring queries. Rather, the term applies to dates that are otherwise in a statistical limbo.

The optimizer attempts to extrapolate the following demographics:

- Table cardinality

Vantage uses extrapolation from the following information to update table cardinalities.

- Histogram summary information
- System UDI counts if they are available, reliable, and not obsolete
- History data
- Single-AMP or all-AMP sampling

- Stale histograms, both for the number of distinct values and for the maximum value of the histogram

Extrapolation logic estimates the data growth from the last time statistics were collected or refreshed in order to detect stale statistics and apply extrapolations. The Optimizer combines cardinality estimates derived from both UDI counts and extrapolation logic to make accurate cardinality estimates, and making stale statistics detection more robust. UDI counts also enable the Optimizer to perform interpolations because of the improved accuracy of data changes provided by the tracking of UDI counts.

Vantage also uses extrapolation to estimate cardinalities based on dynamic single- or all-AMP samples.

The ability of the Optimizer to extrapolate statistics does not remove the need to recollect statistics.

Relative Accuracy of Residual Statistics Versus Dynamic AMP Sampled Statistics for Static Columns

The relative accuracy of residual (*residual* meaning existing statistics that were not collected recently) statistics for static columns with respect to a dynamic AMP sample (in this section, the term *dynamic AMP sample* refers to dynamic single-AMP sampling only.), and whether residual statistics should be considered to be stale, depends on several factors. Residual statistics are not necessarily stale statistics.

The term residual statistics implies the likelihood that those statistics no longer provide an accurate picture of current column data. This is not necessarily a correct assumption.

You can protect yourself from recollecting statistics unnecessarily by specifying various threshold options when you submit your initial COLLECT STATISTICS requests on an index or column set. See the information about COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information about using threshold options to recollect statistics.

If the relative demographics for a column set do not change, then residual statistics are normally a reliable representation of the current demographics of an index or column set. The comparison made by the derived statistics framework between residual statistics, if they exist, and a dynamic AMP sample makes the final determination of whether the Optimizer considers the residual statistics to be stale or not.

It is possible for the cardinality of a table to grow by more than 10%, but for the relative proportion of particular values in its rows not to change. This so-called *Ten Percent Rule* also applies at the partitioning level for row-partitioned tables. If the number of changed row partitions exceeds 10% of the total number of row partitions for the table (in other words, if more than 10% of the rows are added to or deleted from a row partition), then you should recollect statistics on the index. For row-partitioned tables and column-partitioned tables, any refreshment of statistics should include the system-derived PARTITION column.

As a result of these safeguards, 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 can still be more accurate than a newly collected dynamic AMP sample, though even a single-AMP dynamic AMP sample, depending to some degree on whether the column being sampled is indexed or not, typically provides excellent cardinality estimates (see, for example, [Dynamic AMP Sampling](#)). The accuracy of the statistics collected from a dynamic AMP sample also depends to a relatively small degree on the number of AMPs sampled, which is determined by the setting of an internal DBS Control 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.

Using Extrapolated Cardinality Estimates to Assess Table Growth

The Optimizer uses SUMMARY statistics to establish an estimate of the base cardinality of a table. Vantage then extrapolate the cardinality by comparing the saved dynamic sample from the SUMMARY statistics with the fresh sample. The system takes the difference between the new and old samples as the growth in cardinality of the table.

The Optimizer estimates table cardinalities from SUMMARY statistics, which you can collect or recollect explicitly by submitting an appropriate COLLECT SUMMARY STATISTICS request or by collecting or

recollecting statistics on any column set that indirectly updates the SUMMARY statistics, and dynamic AMP samples or collected PARTITION statistics, which can be gathered for all tables, whether they are partitioned or not. The collection of SUMMARY statistics takes advantage of several optimizations that make the process run very quickly. The ideal choice, then, is to provide accurate cardinality estimates to the Optimizer using the method with the least overhead.

The database compares the fresh dynamic AMP sample fetched at the time a request is compiled with the sample from the SUMMARY statistics, and the change in growth of the table is determined from that comparison. Even if the absolute estimate of the sample is not exact, the rate of growth is accurate as long as it is uniform across all of the AMPs on the system because the same AMP is used to collect a “dynamic” statistical sample each time a request is compiled (see [Dynamic AMP Sampling](#) for details).

Some Comparison Scenarios on the Usefulness of Derived Statistics

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.

Assume an average data block size of 50 KB 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 every 2,000 rows, or one row in every 2 data blocks. The Optimizer determines that using the NUSI will save reading some significant number of data blocks, so employing it in the query plan would result in fewer I/Os than doing a full-table scan.

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

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

Without the derived statistics framework being able to detect whether the residual statistics are stale or not, working with them could have meant that the Optimizer would have chosen a bad query plan instead of a new, faster plan. Or, worse still, the residual statistics could produce a new join plan that is much slower than the previous plan.

Cost-Based Optimization

About Cost-Based Optimization

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 longer a given request takes to complete, the more costly it is. (The unit for all Optimizer cost measures is time in milliseconds.)

The Optimizer uses the column and index demographics and statistics gathered by the COLLECT STATISTICS statement. This information permits the Optimizer to estimate the cardinalities of relations for single-table and join access, and identify skew in tables. With a more finely-tuned knowledge of these factors, the Optimizer can generate better plans for satisfying queries using Optimizer cost estimator functions. For more information on COLLECT STATISTICS (Optimizer form), see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

Consider the following fragment of a table, table_x:

state	serial_num
28	12345
28	23456
51	12345
51	23456

Assume the following query results in a full-table scan:

```
SELECT *
FROM table_x
WHERE state IN (28, 51)
AND serial_num IN (12345, 23456);
```

Now modify the request as follows:

```
SELECT *
FROM table_x
WHERE (state=28 AND serial_num=12345)
OR (state=51 AND serial_num=23456)
OR (state=28 AND serial_num=23456)
OR (state=51 AND serial_num=12345);
```

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

Path Selection

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

- (A 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 2 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 or single column partition, 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.

Consider the following expression:

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

In this case, CNF is more high-performing because the access path consists of 2 NUSI SELECT requests 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 SELECT requests would be required.

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.

Consider an expression that involves a single column comparison using IN, such as the following:

```
column IN (value_1, value_2, ...)
```

Internally, that expression is converted to CNF.

```
column=value_1 OR column=value_2 OR ...
```

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

Assume an expression involves a multiple-column comparison using an IN list, such as in the following example:

```
column_1 IN (value_1, value_2, ...)
AND column_2 IN (value_3, ...)
```

The Optimizer converts this syntax internally to conjunctive normal form.

```
(column_1=value_1 OR column_1=value_2 OR ...)
AND (column_2=value_3 OR ...)
```

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:

```
(column_1=value_1 AND column_2=value_3)
OR (column_1=value_2 AND column_2=value_4)
OR ...
```

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

About Predicting Costs for the Optimizer

The purpose of cost prediction is to provide the Optimizer with accurate estimates of the costs to perform various operations in various ways so the Optimizer can make cost-based decisions.

The cost of an operation is the service time required by subsystems to undertake an operation on an otherwise unloaded system. The Optimizer compares alternative ways of performing an operation based on estimates of CPU, I/O, and BYNET component service times and then chooses the alternative that has the lowest overall cost. There are practical situations where a particular approach should be excluded, or its likelihood greatly reduced, by increasing its predicted cost value, so a heuristic cost component is also included with the collection of various subsystem cost components.

Overall cost, which is a linear combination of the subsystem costs, is the value used by the Optimizer to compare the various alternatives available to it. The cost prediction formulas and methods maintain individual subsystem level values in order to make it possible to determine how much of a given overall cost is accounted for by each of its CPU, I/O, BYNET, and heuristic component values. This information makes it possible to distinguish between I/O-intensive and CPU-intensive approaches, enabling an analyst to have a clear view of error sources when a situation occurs in which a predicted cost value is grossly inaccurate.

You can capture subsystem cost values for individual steps in a query plan using either an SQL INSERT EXPLAIN request to funnel the appropriate information into the query capture database table QuerySteps, in the following column set:

- EstCPUCost
- EstIOCost
- EstNetworkCost
- EstHRCost

For more information, see [Query Capture Facility](#) and the information about INSERT EXPLAIN in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

If query logging is enabled, the system captures the same information in DBQLStepTbl in the following column set.

- EstCPUCost
- EstIOWCost
- EstNetCost
- EstHRCost

For more information, see *Teradata Vantage™ - Database Administration*, B035-1093 and the information about BEGIN QUERY LOGGING in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The corresponding overall cost values are captured in the Cost column of the QCD QuerySteps table and in the EstProcTime column of the DBQL DBQLStepTbl tables, respectively.

When predicting costs, there are 2 input data categories. First are values that characterize the runtime environment, and second are values which characterize the database entities involved in the request being optimized. The runtime environment includes hardware platform, operating system, I/O subsystem and database algorithms, essentially all the product set elements of a database system. I/O throughput rates, CPU speed, BYNET capacity, and code path lengths all are examples of runtime environment performance factors. Both CPU and I/O costing vary depending on the sizes of the data blocks containing the data accessed by a request that is being optimized.

During query planning, the Optimizer estimates database entity characteristics using various kinds of metadata, including statistics, dynamic AMP samples, and a variety of estimates derived from these base data sources.

Optimizer Cost Profiles

A *cost profile* is a suite of system-specific values (*cost profile constants*) that Vantage uses for Optimizer costing coefficients, enabling and disabling certain features, and other system settings. A cost profile can apply system-wide or be associated with a PROFILE and apply only to users assigned that PROFILE.

The following table lists some of the Optimizer cost profile constants and what they do:

Note:

Cost profiles should be set up and modified only by Teradata Support Center personnel.

Cost Profile Constant	Description
COLLECT STATISTICS Cost Profile Constant	
StatsDefaultTimeThreshold	Defines a default time threshold value in days for statistics recollection. The database only recollects statistics if this many days or more have passed since statistics were last collected. This setting is applicable only if a COLLECT STATISTICS request does not specify USING THRESHOLD DAYS.
StatsDefaultUserChangeThreshold	Defines a user-determined default data change percentage threshold value for statistics recollection. The database only recollects statistics if the data demographics have changed by this percentage since statistics were last collected.

Cost Profile Constant	Description
	This setting is applicable only if a COLLECT STATISTICS request does not specify USING THRESHOLD PERCENT.
StatsSysChangeThresholdOption	<p>Defines a system-determined default change threshold option for statistics recollection. The database only recollects statistics if the data demographics have changed by this percentage since statistics were last collected.</p> <p>This setting is only applicable if DefaultUserChangeThreshold is disabled and a COLLECT STATISTICS request does not specify USING THRESHOLD PERCENT.</p>
StatsSysSampleOption	<p>Defines a system-determined default sampling option for statistics recollection.</p> <p>This setting is only applicable if a COLLECT STATISTICS request does not specify the USING SAMPLE option.</p>
Partitioning Cost Profile Constants	
PPICacheThrP	Specifies the maximum amount of memory to be used for disk read operations that involve multiple index partitions.
Parameterized Value Peeking Cost Profile Constants	
TacticalResp1	<p>Runtime CPU cost per AMP that determines whether a query is treated as a tactical or decision support request.</p> <p>The value is derived from the CPU time used for query execution in the AMPs.</p> <p>By definition, a request whose per-AMP CPU time ≤ 1 second is considered to be tactical and a request whose per-AMP CPU time > 1 second is considered to be a decision support request.</p>
TacticalResp2	<p>Runtime CPU cost per AMP that determines whether a query is treated as a tactical or decision support query when it is submitted as a HiPriority request.</p> <p>The value is derived from the CPU time used for query execution in the AMPs.</p>
HighParsingPTThreshold	Threshold percentage of the parsing cost compared to the runtime CPU cost on which a determination is made on whether the request has a high parsing cost.
HighParsingRTThreshold	<p>Threshold multiplication factor for the determination of runtime benefits for a query that has a high parsing cost.</p> <p>If the request has a high parsing cost, then the runtime CPU times of specific and generic plans should differ by at least this value multiplied times the specific CPU parsing time.</p>
LowParsingPTThreshold	<p>Threshold multiplication factor for the determination of runtime benefits for a query that has a low parsing cost.</p> <p>If the request has a low parsing cost, then the runtime CPU times of specific and generic plans should differ by at least this value multiplied times the specific CPU parsing time.</p>

Cost Profile Constant	Description
UseHiPriority	A flag that enables or disables the HiPriority-based decisions in the caching algorithm.
ElapsedTimeThreshold	Threshold multiplication factor by which the elapsed time of a specific plan execution (the sum of its parsing and run times) should exceed the elapsed time of the equivalent generic plan.
EstimateCostFilter	Threshold factor by which the estimated cost of a specific plan should be better than the estimated cost of the equivalent generic plan. Used for comparison of estimated costs.
CompareEstimates	A Cost Profile flag that enables or disables the estimate-based comparisons for deciding between using the generic and specific plan for a request.

Statistics And Cost Estimation

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

- The Optimizer evaluates access selectivity based on demographic information.
If statistics have recently been collected on the primary index for a table, then it is likely, but not certain, that an accurate cardinality estimate is available to the Optimizer. If statistics are stale, then the derived statistics framework should minimize any problems with them.
- The Optimizer uses primary index statistics to estimate the cardinality of a table.
Therefore, you should keep those statistics current to ensure optimum access and join planning. Collect new statistics on primary indexes when more than 10% of the total number of rows are added or deleted.

When no statistics are available, or when the table has no primary index, the Optimizer estimates the total number of rows in a table based on information sampled from a dynamically selected AMP (see [Dynamic AMP Sampling](#)).

For the following reasons, this might not be an accurate estimate of the cardinality of the table:

- The table is relatively small and distributed unevenly across the AMPs.
- The NUPI 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.

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

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

Ensuring Indexed Access

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

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

If statistics are not available for a NUSI column, then the Optimizer assumes that indexed values are distributed evenly. The Optimizer estimates the number of rows per indexed value by selecting an AMP dynamically based on its table ID 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 to determine the plan the Optimizer will generate to perform a query.

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

- For NoPI tables, column access is done using secondary or join indexes. If no secondary or join indexes are defined for the column in question, then access is always done using a full-table scan.
- UPIs are used for fastest access to table data.
- USIs are used only to process requests that employ equality constraints.
- The best performance in joins is achieved when the following is possible.
 - Matching UPI values in one table with unique index values, either UPI or USI, in another table.
 - Using only a primary index when satisfying an equality or IN condition for a join.
- NUPIs are used for single-AMP row selection or join processing to avoid sorting and redistribution of rows.
- When statistics are not available for a table, the estimated cost of using an index is based on information from a single AMP. This estimate assumes an even distribution of index values. An uneven distribution impacts performance negatively.
- Composite indexes are used only to process requests that employ equality constraints for all columns that comprise the index.

Note that you can define an index on a single column that is also part of a multicolumn index on the same table.

- Bit mapping is used only when equality or range constraints involving multiple nonunique secondary indexes are applied to very large tables.
- Use either a nonuniquely indexed column or a USI column in a nested join to avoid redistributing and sorting a large table.

For example, consider the following condition.


```
table_1.column_1 = 1 AND table_1.column_2 = table_2.nusi
```

Without using a Nested Join, where table_1 is duplicated and joined with table_1.nusi, both table_1 and table_2 might have to be sorted and redistributed before a Merge Join. This join is not high-performing when table_2 is large.

Environmental Cost Factors

About Environmental Cost Factors

Vantage 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 or cloud environment. This environmental cost data is what enables the Optimizer to operate fully and natively in parallel mode.

Types of Cost Factors

The basic types of environmental cost factors are defined in the following table:

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

External Cost Parameters

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

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

External Cost Parameter Group	Definition
Network cost variables	Message distribution and duplication costs and overheads. Measured in milliseconds.
Optimizer environment variables	Hardware configuration information such as the number of CPUs and vprocs per node, maxima, minima, and average numbers of MIPS per CPU, nodes per clique, disk arrays per clique, and so on. Unitless decimal values.
Miscellaneous cost parameters	DBS Control information such as free space percentages, maximum number of parse tree segments, dictionary cache size, and so on. Measured in various units, depending on the individual cost parameter.
Performance constants	Various values used to assist the Optimizer to determine the best method of processing a given query. Measured in various units, depending on the individual performance constant.
PDE system control files	Lists various system control files used by PDE. Used to initialize the appropriate GDOs with the appropriate values at startup.
DBS control files	Lists various database control files found in the DBS Control record. Used to initialize the appropriate GDOs with their respective appropriate values at startup.
TPA subsystem startup initialization	Initializes or recalculates various Optimizer parameters, including the Optimizer target table GDO used by the target level emulation software. For more information, see <i>Teradata Vantage™ - Database Administration</i> , B035-1093.

Row Partitioning

About Row Partitioning

A table or join index uses row partitioning to assign rows to row partitions on AMPs based on the row partitioning expressions used to define the object. A row partition consists of 0 or more rows of a table or join index that have the same value for the partitioning expressions defined for the database object.

The primary advantage presented by row partitioning is using row partition elimination as an aid to optimizing queries made on row-partitioned database objects.

See the following topics for more information about row partitioning:

- [Row Partition Elimination](#)
- [Static Row Partition Elimination](#)
- [Delayed Row Partition Elimination](#)
- [Product Joins With Dynamic Row Partition Elimination](#)
- [Product Join With Dynamic Row Partition Elimination for Character Partitioning](#)

Consumption of Disk Space by Populated and Empty Partitions

With the large number of partitions that can be defined for a table or join index, it is very likely that a high percentage of those partitions are empty at any given time. For example, a table on a 200 AMP system that defines 100,000 combined partitions with 100 rows per data block and 100 data blocks per each combined partition per AMP has 200 billion rows. This is a relatively small number of combined partitions when you consider that the maximum for a table or join index is 9,223,372,036,854,775,807 combined partitions.

If each row were 100 bytes in length, the primary data alone consumes 20 petabytes of disk. That is 20×10^{15} bytes. It is highly unlikely that every combined partition would be populated. When you consider a multidimensional use of multilevel partitioning, you can easily deduce that not all combinations of dimension values actually occur.

In general, a populated combined partition should have either many data blocks or no data blocks per AMP. For the example proposed in the first paragraph, if there is actually only 200 gigabytes of data and each populated combined partition had 100 data blocks per AMP, about 99% of the combined partitions are empty.

Row Partition Elimination

About Row Partition Elimination

Row partition elimination is a family of methods that limit the number of row partitions that must be scanned to return a query result set for a row-partitioned table or join index. It does this by skipping row partitions that do not contain rows that meet the search conditions of a query. Row partition elimination is an automatic optimization in which the Optimizer determines, based on query conditions and a partitioning expression, that some row partitions for that partitioning expression cannot contain qualifying rows; therefore, those row partitions can be skipped during a file scan. Row partitions that are skipped for a particular query are called *eliminated row partitions*.

Vantage supports the following types of row partition elimination:

- Static
- Delayed
- Dynamic

Vantage also supports various forms of column partition elimination for column-partitioned tables and join indexes. See [Column Partition Elimination](#) for more information about column partition elimination.

Mixing Row Partition Elimination Methods Within a Request

Row partition elimination methods can be mixed within the same query and can also be mixed with column partition elimination methods when a table or join index has both column and row partitioning. For example, static row partition elimination can be used for some partitioning levels, while dynamic row partition elimination can be used for other levels, and some levels might not have any row partition elimination. Some individual partition levels might even benefit from a mix of multiple forms of partition elimination.

When there are multiple row partitioning expressions, Vantage combines row partition elimination at each of the levels to further reduce the number of data subsets that need to be scanned. For most applications, the greatest benefit of row partitioning is obtained from partition elimination.

Vantage can eliminate partitions from search consideration at any number of levels or at a combination of levels.

The full cylinder read optimization is not supported when there is row partition elimination. Therefore, there are trade-offs to consider in using partitioning. The Optimizer does use row partition elimination when possible, and does not cost a full-table scan with full cylinder reads against the cost of reading a subset of row partitions using block reads. In most cases, it is reasonable to assume that row partition elimination provides a greater benefit than a full-table scan with full cylinder reads.

Static Row Partition Elimination

When query conditions are such that they allow row partition elimination to be determined by the Optimizer during the early stages of query optimization, the form of row partition elimination used is referred to as *static row partition elimination*.

Any single-table constraints on partitioning columns can be used for static row partition elimination, including those on the system-derived column PARTITION or any of the members of the system-derived PARTITION#L n column set, where the value of n ranges from 1 through 62, inclusive.

See [Static Row Partition Elimination](#) for more information about this method of partition elimination.

Delayed Row Partition Elimination

When query conditions are based on a comparison derived in part from USING request modifier variables or from the result of a built-in function, it is not possible for the Optimizer to reuse a cached query plan as it would otherwise do because a cached plan needs to be general enough to handle changes in search condition values in subsequent executions.

In this case, the Optimizer applies row partition elimination at a later point in the optimization process, at the time it builds the finalized query plan from a cached plan using the values for this specific execution of the plan. This form of partition elimination is referred to as *delayed row partition elimination*.

See [Delayed Row Partition Elimination](#) for more information about this method of row partition elimination.

Dynamic Row Partition Elimination

When query conditions reference values in other tables that allow for row partition elimination to be done as a request is executing, the row partition elimination is referred to as *dynamic*.

When query conditions reference values in other tables that would allow row partition elimination as the query is executing, row partition elimination is performed dynamically by the AMP database software after a query has already been optimized and while it is executing. This form of partition elimination is referred to as *dynamic row partition elimination*.

Vantage can use a special form of dynamic row partition elimination known as *dynamic row partition elimination with range conditions* when it joins a given left table partition with a sequential range of row partitions from the right table when that range of row partitions is derived from a range join condition.

This form of dynamic row partition elimination is only supported for single-level partitioned tables that use RANGE_N partitioning.

Dynamic row partition elimination can also be used to simplify and enhance join performance by selecting the least costly method from a set of join methods especially designed to be used with row partition elimination.

See [Product Joins With Dynamic Row Partition Elimination](#) for more information about this method of partition elimination.

Character Partition Elimination

Eliminating character partitions is not a separate form of row partition elimination, but it has some unique characteristics.

Character partitioning typically defines a small number of row partitions compared to non-character row partitioning. The effective limit for character partitioning is roughly 2,000 partitions because of a 16,000 character limit on partitioning CHECK constraints (see *Teradata Vantage™ - Database Design*, B035-1094 or the information about CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for more information about partitioning CHECK constraints and their various forms). As a result, the most effective way to specify character partitioning for very large tables is often as part of a multilevel partitioning expression.

Combining partition elimination on multiple partitioning levels can then reduce the ratio of combined partitions that must be read from the rough limit of 1/2000 for a character-partitioned table up to the maximum number of combined partitions that can be defined.

For example, assuming a 2-byte internal partition number, the following table defines 65,475 partitions, which is close to the maximum number of 65,535 combined partitions that can be defined for 2-byte partitioning. In this case, 27 row partitions are defined at level 1, 485 row partitions at level 2, and 5 row partitions at level 3.

[illegible]

```
AND      DATE '2007-12-31'
EACH INTERVAL '7' DAY,
NO RANGE, UNKNOWN),
RANGE_N(revenue_code BETWEEN 1
AND      4
EACH 1,
NO RANGE OR UNKNOWN ));
```

Assuming an even distribution of rows to combined partitions, the following SELECT request accesses only one combined partition, or 1/65,475 of the rows in *markets*.

```
SELECT *
FROM markets
WHERE productname BETWEEN 'a' AND 'az'
      AND activity_date=DATE '2007-08-15'
      AND revenue_code=1;
```

Vantage uses row partition elimination on a character partitioning level together with partition elimination on other partitioning levels to access a smaller subset of the data than can be done using a character-partitioned table alone. This is because the maximum number of row partitions a character-partitioned table can define is limited to roughly 2,000.

Static and Delayed Row Partition Elimination for Row Partitioning That Specify BEGIN and END Period Bound Functions

Only static and delayed row partition elimination are supported for Period expressions that specify BEGIN and END Period bound functions.

The rules for row partition elimination of DateTime expressions apply in the same way for partitioning expressions based on the following functions that incorporate BEGIN and END Period bound functions:

- Direct partitioning by BEGIN and END Period bound functions.
- Partitioning by CASE_N functions using BEGIN and END Period bound functions.
- Partitioning by CASE functions using BEGIN and END Period bound functions.
- Partitioning by RANGE_N functions using BEGIN and END Period bound functions.

The database performs single-partition scanning only when the table accessed defines its partitioning based on BEGIN or END Period bound functions and when you specify equality constraints on BEGIN or END Period bound functions in a WHERE clause.

The rules for performing single-partition scans are as follows:

IF a table is partitioned on this Period bound function...	AND the WHERE clause specifies an equality constraint on ...	THEN the database does ...
BEGIN	a BEGIN Period bound function	single-partition access.

IF a table is partitioned on this Period bound function...	AND the WHERE clause specifies an equality constraint on ...	THEN the database does ...
	an END Period bound function	not do single-partition access.
	both a BEGIN Period bound and an END Period bound function	single-partition access.
END	a BEGIN Period bound function	not do single-partition access.
	an END Period bound function	single-partition access.
	both a BEGIN and an END Period bound functions	single-partition access.
BEGIN and END	a BEGIN Period bound function	not do single-partition access.
	an END Period bound function	not do single-partition access.
	both a BEGIN and an END Period bound functions	single-partition access.

Assume the following table definitions for the example set that illustrates these rules:

```
CREATE SET TABLE t11 (
  a INTEGER,
  b PERIOD(DATE))
PRIMARY INDEX(a)
PARTITION BY CAST((BEGIN(b)) AS INTEGER);

CREATE SET TABLE t12 (
  a INTEGER,
  b PERIOD(DATE))
PRIMARY INDEX(a)
PARTITION BY CAST((END(b)) AS INTEGER);
```

In the examples that follow, the relevant EXPLAIN text phrases are highlighted in boldface type.

The following SELECT request scans all the row partitions because the WHERE clause predicate and the partitioning expression are defined on different bounds:

```
EXPLAIN SELECT *
FROM t11
WHERE END(b)=DATE '2010-02-03';
```

The following shows a portion of the EXPLAIN output:

```
...
3) We do an all-AMPs RETRIEVE step from df2.t11 by way of an
```

all-rows scan with a condition of `("(END(df2.t11.b))= DATE '2010-02-03'")` 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 (56 bytes). The estimated time for this step is 0.03 seconds.

The following SELECT request performs a single-partition scan because the WHERE clause predicate is defined on an END Period bound function and the partitioning expression is defined using an equality condition on an END Period bound function.

```
EXPLAIN SELECT *
  FROM t12
  WHERE END(b)=DATE '2011-02-03';
```

...

3) We do an all-AMPs RETRIEVE step from **a single partition** of df2.t12 with a condition of `("df2.t12.b = PERIOD(DATE '2011-02-03' - INTERVAL '1' DAY, DATE '2011-02-03')")` with a residual condition of `("(END(df2.t12.b))= DATE '2011-02-03'")` 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 (56 bytes). The estimated time for this step is 0.03 seconds.

...

The following SELECT request performs a single-partition scan because the WHERE clause predicate is defined on an END Period bound function and the partitioning expression is defined using an equality condition on an END Period bound function:

```
EXPLAIN SELECT *
  FROM t12
  WHERE a=1
  AND   END(b)=DATE '2011-02-03';
```

...

1) First, we do a single-AMP RETRIEVE step from **a single partition** of df2.t12 by way of the primary index `"df2.t12.a = 1, df2.t12.b = PERIOD (DATE '2011-02-03'- INTERVAL '1' DAY, DATE '2011-02-03')"` with a residual condition of `("((END(df2.t12.b))= DATE '2011-02-03') AND (df2.t12.a = 1)")` into Spool 1 (one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with low confidence to be 1 row (56 bytes). The estimated time for this step is 0.02 seconds.

When a table is row partitioned directly using BEGIN and END Period bound functions resulting into a numeric value, and whenever the conditions in a request specify partitioning columns of that table that are specified in the partitioning expression, then the database performs row partition elimination using the same rules for BEGIN and END Period bound functions as apply for other DateTime expressions.

Assume the following table definition:

```
CREATE SET TABLE t1 (
  a INTEGER
  b PERIOD(DATE)
PRIMARY INDEX(a)
PARTITION BY CAST((END(b) AS INTEGER);
```

The following SELECT request scans 55,333 row partitions of *t1*:

```
EXPLAIN SELECT *
  FROM t1
  WHERE b>PERIOD(DATE '1901-02-02');
```

...

3) We do an all-AMPs RETRIEVE step from **55333 partitions of** df2.T1 with a condition of ("df2.T1.b > (PERIOD (DATE '1901-02-02', DATE '1901-02-02'))") 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 (56 bytes). The estimated time for this step is 0.03 seconds.

...

When a table is row partitioned using a CASE_N or CASE function that specifies BEGIN or END Period bound functions, the existing rules for row partition elimination with DateTime expressions also apply.

Assume the following table definition:

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderperiod   PERIOD (DATE) NOT NULL,
  o_orderpriority CHARACTER (21),
  o_comment       VARCHAR (79))
PRIMARY INDEX(o_orderkey)
PARTITION BY CASE_N(END(o_orderperiod) <= DATE '2010-03-31', /*Q1*/
                    END(o_orderperiod) <= DATE '2010-06-30', /*Q2*/
                    END(o_orderperiod) <= DATE '2010-09-30', /*Q3*/
```

```
END(o_orderperiod) <= DATE '2010-12-31' /*Q4*/
);
```

The following SELECT request scans 2 row partitions of *orders* and displays the details of the orders placed for the first 2 quarters:

```
EXPLAIN SELECT *
FROM orders
WHERE END(o_orderperiod) > DATE '2010-06-30';
```

...

3) We do an all-AMPs RETRIEVE step from **2 partitions of** df2.orders with a condition of ("(END (df2.orders.O_orderperiod))> DATE '2010-06-30'") 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 (167 bytes). The estimated time for this step is 0.03 seconds.

...

When a table is row-partitioned using a RANGE_N function that specifies BEGIN or END Period bound functions, and whenever the conditions in a request specify a column that is specified in the partitioning expression, the database performs row partition elimination using the same rules that apply for DateTime expressions.

Assume you define a sales history table with the following definition:

```
CREATE TABLE sales_history (
  product_code      CHARACTER(8),
  quantity_sold     INTEGER,
  transaction_period PERIOD(DATE))
PRIMARY INDEX (product_code)
PARTITION BY RANGE_N(END (transaction_period)
                     BETWEEN DATE '2006-01-01'
                     AND      DATE '2015-12-31'
                     EACH INTERVAL '1' YEAR);
```

The following SELECT request scans 5 row partitions of sales_history before 2010:

```
EXPLAIN SELECT *
FROM sales_history
WHERE transaction_period < PERIOD(DATE '2010-01-01');
```

...

3) We do an all-AMPs RETRIEVE step from **5 partitions of**

```
df2.sales_history with a condition of (
"df2.sales_history.transaction_period < (PERIOD (DATE '2010-01-01',
DATE '2010-01-01'))") into Spool 1 (group_amps), which is built
locally on the AMPs. The size of Spool 1 is estimated with no
confidence to be 1 row(64 bytes). The estimated time for this step
is 0.03 seconds.
```

```
...
```

When the RANGE_N partitioning expression specifies BEGIN or END bound functions and the query conditions contain the same partitioning expression, then the Optimizer uses row partition elimination.

The following SELECT request scans 4 row partitions of the *sales_history* table to display all the sales history before 2010:

```
EXPLAIN SELECT *
FROM sales_history
WHERE END (transaction_period) < DATE '2010-01-01';
```

```
...
```

```
3) We do an all-AMPs RETRIEVE step from 4 partitions of
df2.sales_history with a condition of (
"(END(df2.sales_history.transaction_period ))< DATE
'2010-01-01') into Spool 1 (group_amps), which is built locally
on the AMPs. The size of Spool 1 is estimated with no confidence
to be 1 row (64 bytes). The estimated time for this step is 0.03
seconds.
```

```
...
```

Static Row Partition Elimination

About Static Row Partition Elimination

Static row partition elimination, especially when applied at multiple levels, can significantly reduce the number of data blocks the database must read and process to respond to a query. Static row partition elimination is based on constant conditions on the partitioning columns and system-derived columns PARTITION and PARTITION#L *n*. The term *constant conditions* refers to literal values and predicate conditions such as equality, inequality, and BETWEEN and the substitution of any CURRENT_TIME, CURRENT_TIMESTAMP, or USING values for a specific plan and substitution of any DATE, CURRENT_DATE, TEMPORAL_DATE, or USER values.

The database applies static row partition elimination to each level independently, and combines the result into a single row partition elimination list. If the system cannot eliminate any row partitions at a level, then it must process all the row partitions for that level. The same algorithms as used for single-level row partitioning are used at each level. The exception to this is conditions on system-derived column PARTITION#L *n*, which the row partition elimination algorithms consider for level *n* instead of conditions on

system-derived column PARTITION. The system considers conditions on both the system-derived column PARTITION and the system-derived column PARTITION#L1 for single-level row partitioning.

Static partition elimination for a table or join index that has 8-byte partitioning is limited to 8,000 partitions, and no further static partition elimination occurs for that table or join index. The database applies static partition elimination to the first partitioning level, then the second partitioning level, and so on until either partition elimination completes or the limit of 8,000 non-eliminated partitions is reached. If the limit is reached, the remaining partitions in that level and any lower levels are not eliminated.

If there are constant equality constraints on the partitioning columns of a partitioning expression, static row partition elimination occurs. All but the specified row partition for that partitioning expression is eliminated.

The Optimizer might not always be able to eliminate as many row partitions as is theoretically possible because of limitations in its ability to detect such opportunities.

You should always run EXPLAIN request modifiers and measure actual system performance to verify that the desired partition elimination and performance occur for a candidate partitioning scheme and workload. If the results of your verification tests are not acceptable, or if they are not at the performance levels you expected, you should consider using a different partitioning scheme, or even using no partitioning, and then make other physical database design changes or query modifications such as adding constant conditions on the partitioning columns to obtain the level of performance you need and want.

Static Row Partition Elimination for RANGE_N Partitioning Expressions

If there are constant inequality constraints on the row partitioning columns of a row partitioning expression, static row partition elimination can occur when there is a single partitioning column for a RANGE_N partitioning expression and the test value expression is a recognized non-decreasing linear expression based on the partitioning column. The recognized linear expressions are the following:

- A row partitioning column.
- A CAST, Teradata conversion, or EXTRACT of a recognized linear expression.
- The sum of 2 recognized linear expressions.
- The sum of a recognized linear expression and a constant expression.
- The sum of a constant expression and a recognized linear expression.
- The difference of a recognized linear expression and a constant expression.
- The product of a row partitioning column and a constant expression, where the value of the constant expression is greater than 0.
- The product of a constant expression and a row partitioning column, where the value of the constant expression is greater than 0.
- The division of a row partitioning column by a constant expression, where the value of the constant expression is greater than 0.
- A CASE_N row partitioning expression if the Optimizer recognizes that conditions in the CASE_N expression are satisfiable in relation to the specified query conditions.

Note that this list does not include all possible non-decreasing linear expressions. Any such expressions that are not listed are not recognized for static row partition elimination by the database.

Static Row Partition Elimination for CASE_N Row Partitioning Expressions

Static row partition elimination can occur for a CASE_N row partitioning expression if the Optimizer recognizes that conditions in the CASE_N row partitioning expression are not satisfiable in relation to the specified query conditions.

Static Row Partition Elimination for Character Partitioning Expressions

There are special considerations for row partition elimination for RANGE_N character partitioning.

An example is the case where the RANGE_N condition specifies ranges that do not specify a range end, and there is a term of the form *part_col* > *range_end*, where *range_end* is the string equal to the next range beginning, except it substitutes the character with the next lowest code in the last character position as is done for the partitioning expression of *t1* in the following example:

```
CREATE TABLE t1(
  i INTEGER,
  j CHARACTER(4),
  k INTEGER)
PRIMARY INDEX(i)
PARTITION BY (RANGE_N(j BETWEEN 'aaaa', 'cccc', 'eeee', 'gggg',
                        'iiii', 'kkkk', 'mmmm', 'oooo',
                        'qqqq', 'ssss'
                        AND 'tttt',
                        NO RANGE));
```

The following SELECT request reads from 9 row partitions even though only 8 row partitions need to be read to return the response set. This happens because the range end of 'eeed' for row partition 2 is implied rather than explicitly specified in the RANGE_N expression.

```
SELECT *
FROM t1
WHERE j > 'eeed';
```

For INTEGER and DATE data, the Optimizer can subtract 1 (or 1 day) from the next range start to find the range end, but it cannot determine the next lowest or highest code point in a character set based on a given collation sequence. As a result, row partition elimination always finds one extra row partition that needs to be read in such cases.

Row partition elimination is supported on LIKE predicates of the form *part_col* LIKE 'abc%' for RANGE_N partitioning, where 'abc' represents any string literal. The '%' metacharacter, which matches any string of 0 or more characters if present in the search string, might only exist as the last character in the string (depending on the server character set, this might not be ASCII '%', but a different code point such as full width percent: `_unicode 'FF05'XC`). Use of the '_' metacharacter to match any single character results in no row partition elimination from the containing predicate.

Note that row partition elimination is not guaranteed for LIKE predicates with CASE_N character partitioning. This is because row partition elimination with CASE_N is dependent on SAT-TC support of LIKE predicates.

The best practice is to define a character partitioning using a RANGE_N partitioning expression if you expect the Optimizer to eliminate row partitions when LIKE predicates are specified in a query. Row partition elimination should occur with CASE_N character partitioning, similar to non-character partitioning, when other comparison operators such as =, <, >, <=, >=, <>, BETWEEN are specified in the CASE_N expression and WHERE clause predicate.

Row partition elimination requires that to be able to evaluate predicates, translation of partitioning column data is not required for the execution of that predicate.

If there is no valid translation from the character set of a constant expression in the predicate to the character set of the partitioning column in the comparison, then the database converts both to Unicode. When this occurs, no row partition elimination occurs.

To increase the likelihood of row partition elimination, you should ensure that the partitioning columns of a character-partitioned table be stored using the Unicode server character set.

A CAST of character data can truncate the string, resulting in a loss of information.

The database does not support row partition elimination for RANGE_N partitioning when the test value involves a CAST or other data conversion operation unless the predicate qualifying the level for row partition elimination is an equality condition.

Static row partition elimination for RANGE_N partitioning requires that none of the ranges be *unbounded*, meaning specified by an asterisk character for the start of the first range or the end of the last range. This restriction also applies to static row partition elimination for a character RANGE_N partitioning level. There is an exception to this restriction if the WHERE clause predicate qualifying the level for row partition elimination is an equality condition, and static row partition elimination should occur at that level.

For a character partitioning level, the following additional conditions must be met for static row partition elimination to be possible:

- There must be at most one character partitioning column at a row partitioning level for it to qualify for static row partition elimination.
- The case sensitivity of the WHERE clause condition qualifying a level for static row partition elimination must match the case sensitivity of all comparisons you specify in the partitioning expression for that level.

The case sensitivity of all string functions and column attribute qualifiers in any non-constant expression of the partitioning expression must be the same as the sensitivities for the functions and qualifiers specified in the WHERE clause condition.

The following are case-insensitive:

- LOWER
- SOUNDEX
- UPPER
- UPPERCASE qualifier

The following are case-sensitive:

- CHAR2HEXINT
- TRANSLATE
- TRANSLATE_CHK
- TRIM
- VARGRAPHIC

The following functions follow the same rules as comparison operators and the database examines the function inputs along with default case sensitivity for the session mode, either ANSI or Teradata, to determine case sensitivity:

- INDEX
- MINDEX
- POSITION

The presence of the SUBSTRING function does not affect case sensitivity.

A WHERE clause condition is considered to be case insensitive if any of the comparisons or string functions involving non-constant expressions in the condition is case insensitive.

- The presence of the concatenation operator with any non-constant expression in the partitioning expression disallows static row partition elimination at that level.
- The session collation of the DML request must be the same as the table collation.
- If the partitioning expression specifies the RANGE_N function, its test value must be a single column reference with no CAST or data conversion qualifier such as UPPERCASE, unless the predicate qualifying the level for row partition elimination is an equality condition.

The WHERE clause predicate that enables static row partition elimination must be a comparison between a single-column reference and a constant expression.

If you specify a CAST expression or data conversion qualifier such as UPPERCASE on a column reference in the WHERE clause predicate, then the Optimizer does not apply row partition elimination for that predicate.

Note that these rules do not apply to static row partition elimination for non-character partitioning levels, and the session collation need not match the table collation for row partition elimination to occur on non-character-based partitioning levels.

Row partition elimination can also occur when you apply the following string functions to constant string literals within the WHERE clause predicate:

- || (concatenation operator)
- LOWER
- SOUNDEX
- SUBSTRING/SUBSTR
- TRANSLATE
- TRIM
- UPPER
- VARGRAPHIC

For example, consider the following character partitioning definition:

```
CREATE MULTISET TABLE df2.t4, NO FALLBACK, NO BEFORE JOURNAL,
                                NO AFTER JOURNAL, CHECKSUM=DEFAULT (
  a INTEGER,
  b CHARACTER(4) CHARACTER SET UNICODE NOT CASESPECIFIC)
PRIMARY INDEX (a)
PARTITION BY RANGE_N(b BETWEEN 'A' AND 'F', 'FA' AND 'Z');
```

The following SELECT request reads from only one row partition because it specifies the concatenation operator and the SOUNDEX function in its WHERE clause:

```
SELECT *
FROM t4
WHERE b BETWEEN 'a' || 'a' AND SOUNDEX('d');
```

Specifying a string function on a partitioning column within a predicate, however, eliminates the possibility of its use for row partition elimination. For example, consider the following SELECT request against table *t4*. Because the request specifies the TRIM function on column *b*, which is the character partitioning column for *t4*, the Optimizer does not apply static row partition elimination to return the result set.

```
SELECT *
FROM t4
WHERE TRIM(trailing ' ' FROM b) BETWEEN 'a' AND 'd';
```

The database handles requests issued in ANSI session mode differently than requests submitted in Teradata session mode.

In ANSI session mode, the database handles any single-table conditions between a NOT CASESPECIFIC column and a constant with no case sensitivity qualifier by internally converting the column reference to be CASESPECIFIC. This disqualifies the term from being used in row partition elimination; therefore, if you want the Optimizer to apply row partition elimination to requests in ANSI session mode, you should create the character partitioning columns as CASESPECIFIC. Alternatively, you can cast all of the constants in the WHERE clause predicate of a query to be NOT CASESPECIFIC.

For example, consider the following table definition:

```
CREATE MULTISET TABLE MWS2.t4, NO FALLBACK, NO BEFORE JOURNAL,
                                NO AFTER JOURNAL, CHECKSUM=DEFAULT (
  a INTEGER,
  b CHARACTER(4) CHARACTER SET UNICODE NOT CASESPECIFIC)
PRIMARY INDEX (a)
PARTITION BY RANGE_N(b BETWEEN 'A' AND 'F',
                      'FA' AND 'Z');
```


In ANSI session mode, the database reads from all row partitions (does not apply static row partition elimination) because of the implicit CAST on column *b*.

```
SELECT *
FROM t4
WHERE b BETWEEN 'a' and 'd' ;
```

After you modify the same request slightly, the database reads from only one row partition because of the explicit CAST of column *b*.

```
SELECT *
FROM t4
WHERE b BETWEEN CAST ('a' AS CHARACTER(4) NOT CASESPECIFIC)
      AND      CAST ('d' AS CHARACTER(4) NOT CASESPECIFIC);
```

The following examples show the behavior of SELECT requests against character partitioning columns in Teradata session mode. First, create the following table:

```
CREATE TABLE markets (  
    productname      VARCHAR(50) NOT CASESPECIFIC,  
    region            BYTEINT NOT NULL,  
    activity_date     DATE FORMAT 'yyyy-mm-dd' NOT NULL,  
    revenue_code      BYTEINT NOT NULL,  
    business_sector   BYTEINT NOT NULL,  
    note              VARCHAR(256))  
  
PRIMARY INDEX (productname, region)  
  
PARTITION BY (RANGE_N(productname BETWEEN 'A','B','C','D','E','F',  
                                'G','H','I','J','K','L',  
                                'M','N','O','P','Q','R',  
                                'S','T','U','V','W','X',  
                                'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ',  
                                NO RANGE,  
                                UNKNOWN));
```

The comments associated with the following queries assume an even distribution of the rows among the row partitions and many data blocks per partition. Without being able to use static row partition elimination, these queries would be all-AMP, full-table scans.

The first SELECT request reads 2 row partitions on all AMPs. This means that the database reads roughly 2/28 of the data blocks in the *markets* table.

```
SELECT *
FROM markets
WHERE productname LIKE 'B%';
```

The next SELECT request reads only one row partition on all AMPs; therefore, the database reads roughly 1/28 of the data blocks in *markets*.

```
SELECT *
FROM markets
WHERE productname < 'B';
```

The next SELECT request reads 2 row partitions on all AMPs; therefore, the database reads roughly 2/28 of the data blocks in *markets*.

```
SELECT *
FROM markets
WHERE productname LIKE UPPER('B%');
```

The following example uses the KANJIEUC_0U session character set and JIS_COLL collation.

```
CREATE TABLE t12 (
  a INTEGER,
  b VARCHAR(15) CHARACTER SET GRAPHIC)
PRIMARY INDEX(a)
PARTITION BY (RANGE_N(b BETWEEN _GRAPHIC '9758'XC
                        AND _GRAPHIC '9759'XC,
                        NO RANGE, UNKNOWN));
```

The following SELECT request reads one row partition on all AMPs:

```
SELECT *
FROM t12
WHERE b BETWEEN _GRAPHIC '9758'XC AND _GRAPHIC '9759'XC;
```

Examples of Rewrites Using Row Partitioning

Table Definition for the Examples

The queries in the following example set all use the multilevel partitioned table defined by the following CREATE TABLE request:

```
CREATE TABLE markets (
  productid      INTEGER NOT NULL,
  region         BYTEINT NOT NULL,
  activity_date  DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  revenue_code   BYTEINT NOT NULL,
  business_sector BYTEINT NOT NULL,
```

```
note          VARCHAR(256))
PRIMARY INDEX (productid, region)
PARTITION BY (
RANGE_N(region          BETWEEN 1
                        AND      9
                        EACH      3),
RANGE_N(business_sector BETWEEN 0
                        AND      49
                        EACH     10),
RANGE_N(revenue_code    BETWEEN 1
                        AND      34
                        EACH      2),
RANGE_N(activity_date   BETWEEN DATE '1986-01-01'
                        AND      DATE '2007-05-31'
                        EACH INTERVAL '1' MONTH));
```

Row Partitioning Details

Note the following details about the partitioning of this table:

- The following four levels of row partitioning are defined with the indicated number of row partitions per level:

Number of Row Partitioning Level	Partitioning Column for This Level	Number of Row Partitions Defined at This Level
1	region	3
2	business_sector	5
3	revenue_code	17
4	activity_date	257

- The total number of combined partitions defined for the table is the maximum for a 2-byte internal partition number: 65,535, calculated as follows:

$$\begin{aligned} \sum \text{partitions} &= P_{L1} \times P_{L2} \times P_{L3} \times P_{L4} \\ &= 3 \times 5 \times 17 \times 257 \\ &= 65,535 \end{aligned}$$

where:

Equation element ...	Specifies ...
P_{Ln}	the number of partitions at level n .

For the sake of simplicity, the comments associated with the example queries make the following assumptions:

- Rows are evenly distributed among the row partitions.
- Many data blocks per combined partition can be combined.

If the example queries were not able to capitalize on static row partition elimination, they would all require all-AMP, full-table scans to be processed.

Row Partitioning Example: Reading Only Row Partition Two

For the following query, the database reads only row partition 2 at level 1 on all AMPs. As a result, it reads roughly 1/3 (33.3%) of the data blocks for the markets table because there are 3 row partitions at level 1, and Vantage needs to read only one of them to access all of the qualified rows.

Row partition 2 includes rows with a value of 5 or 6 for region. The database necessarily reads those rows as it reads row partition 2, but they do not qualify for the query predicate, so they are not returned in the results set.

```
SELECT *
FROM markets
WHERE region=4;
```

Row Partitioning Example: Reading Only Row Partitions Four and Five at Level Two on All AMPs

For the following query, the database reads only partitions 4 and 5 at level 2 on all AMPs. As a result, it reads roughly 2/5 (40%) of the data blocks for the markets table because there are 5 row partitions at level 2, and the database needs to read 2 of those 5 to access all of the qualified rows.

Row partition 4 includes rows whose value for business_sector is 30. The database necessarily reads those rows as it reads partition 4, but they do not qualify for the query predicate, so they are not returned in the results set.

```
SELECT *
FROM markets
WHERE business_sector>30;
```

Row Partitioning Example: Reading Only Row Partitions One, Two, and Three

For the following query, the database reads only partitions 1, 2, and 3 at level 3 on all AMPs. As a result, it reads roughly 3/17 (17.6%) of the data blocks for the markets table because there are 17 row partitions at level 3, and the system needs to read only 3 of them to access all of the qualified rows.

Partition 3 includes rows whose value for revenue_code is 6. The database necessarily reads those rows as it reads partition 5, but they do not qualify for the query predicate, so they are not returned in the results set.

```
SELECT *
FROM markets
WHERE revenue_code<5;
```

Row Partitioning Example: Reading Only Row Partitions 50 and 51

For the following query, the database reads only partitions 50 and 51 at level 4 on all AMPs. As a result, it reads roughly 2/257 (0.78%) of the data blocks for the markets table because there are 257 row partitions at level 4, and the system needs to read only 2 of them to access all of the qualified rows.

Both partition 50 and partition 51 include rows that do not meet the predicate for the query. The database necessarily reads those rows as it reads the row partitions, but they do not qualify for the query predicate, so they are not returned in the results set.

```
SELECT *
FROM markets
WHERE activity_date>=DATE '1990-02-12'
AND activity_date<=DATE '1990-03-28';
```

Row Partitioning Example: Reading Only Row Partitions Four and Five at Level Two on All AMPs and Partition Two at Level One on All AMPs

For the following query, the system reads partitions 4 and 5 at level 2 on all AMPs and partition 2 at level 1 on all AMPs. As a result, it reads roughly 2/15 (13.3%) of the data blocks for the markets table because there are 15 partitions at combined levels 1 and 2 (the combined total is 3 x 5 = 15), 3 at level 1 and 5 at level 2, and the system needs to read only 2 of them to access all of the qualified rows.

Note that all of these partitions contain some rows that do not qualify for the query predicate, so the system does not return them in the results set.

```
SELECT *
FROM markets
WHERE region=4
AND business_sector>30;
```

Row Partitioning Example 1: Reading Multiple Row Partitions at Multiple Levels on All AMPs

For the following query, the database reads the following set of row partitions at the indicated levels on all AMPs: 2 row partitions of level 4 (partitions 50 and 51), in 3 row partitions at level 3 (partitions 1, 2, and 3), and in 2 row partitions at level 2 (partitions 4 and 5).

Level Number	Number of Row Partitions Read at This Level	Partition Numbers Read Within the Level
2	2	<ul style="list-style-type: none">• 4• 5

Level Number	Number of Row Partitions Read at This Level	Partition Numbers Read Within the Level
3	3	<ul style="list-style-type: none"> • 1 • 2 • 3
4	2	<ul style="list-style-type: none"> • 50 • 51

As a result, the database reads roughly 12/21,845 (0.05%) of the data blocks for the markets table because there are 21,845 partitions (the combined total is $5 \times 17 \times 257 = 21,845$) at combined levels 2, 3, and 4, and the system needs to read only 12 of them (the combined total is $2 \times 3 \times 2 = 12$) to access all of the qualified rows.

All of these partitions contain some rows that do not qualify for the query predicate, so the database does not return them in the results set.

```
SELECT *
FROM markets
WHERE business_sector>30
AND revenue_code<5
AND activity_date>=DATE '1990-02-12'
AND activity_date<=DATE '1990-03-28';
```

Row Partitioning Example 2: Reading Multiple Row Partitions at Multiple Levels on All AMPs

For the following query, the database reads the following set of row partitions at the indicated levels on all AMPs: 2 row partitions of level 4 (partitions 50 and 51), in 3 row partitions at level 3 (partitions 1, 2, and 3), in 2 row partitions at level 2 (partitions 4 and 5), in 1 row partition of level 1 (partition 2).

Level Number	Number of Partitions Read at This Level	Partition Numbers Read Within the Level
1	1	2
2	2	<ul style="list-style-type: none"> • 4 • 5
3	3	<ul style="list-style-type: none"> • 1 • 2 • 3
4	2	<ul style="list-style-type: none"> • 50 • 51

As a result, the database reads roughly 12/65,535 (0.18%) of the data blocks for the markets table because there are 65,535 partitions (the combined total is $3 \times 5 \times 17 \times 257 = 65,535$) at combined levels 2, 3, and 4, and the database needs to read only 12 of them (the combined total is $1 \times 2 \times 3 \times 2 = 12$) to access all of the qualified rows.

All of these partitions contain some rows that do not qualify for the query predicate, so the database does not return them in the results set.

```
SELECT *
FROM markets
WHERE region=4
AND   business_sector>30
AND   revenue_code<5
AND   activity_date>=DATE '1990-02-12'
AND   activity_date<=DATE '1990-03-28';
```

Row Partitioning Example: Reading One Row Partition at Level Two on All AMPs

For the following query, the database reads one row partition at level 2 on all AMPs: partition 1. As a result, it reads roughly 1/5 (20.0%) of the data blocks for the markets table because there are 5 row partitions at level 2, and the database needs to read only one of them to access all of the qualified rows.

The row partition the database must read contains the rows having a value for business_sector between 0 and 9, inclusive.

This row partition contains some rows that do not qualify for the query predicate, so the system does not return them in the results set.

```
SELECT *
FROM markets
WHERE PARTITION#L2=1;
```

Row Partitioning Example: Reading One Combined Row Partition on All AMPs

For the following query, the database reads one combined row partition on all AMPs because the specified predicate is an equality condition on the combined partition that is equal to 32,531. As a result, it reads only 1/65,525 (0.15%) of the data blocks for the markets table because there is only one combined partition that has the value 32,531, and the system needs to read only the data block that contains combined partition number 32,531 to access all of the qualified rows.

The row partition the database must read contains the rows defined by the following value set:

PARTITION#Ln	PARTITION Value Where Qualified Rows Are Stored	Conditions Mapped To This PARTITION Value Range
1	2	region BETWEEN 4 AND 6
2	3	business_sector BETWEEN 20 AND 29

PARTITION#Ln	PARTITION Value Where Qualified Rows Are Stored	Conditions Mapped To This PARTITION Value Range
3	8	revenue_code BETWEEN 15 AND 16
4	149	activity_date BETWEEN 1998-05-01 AND 1998-05-31

```
SELECT *
FROM markets
WHERE PARTITION=32531;
```

Delayed Row Partition Elimination

Row partition elimination can occur with conditions comparing a partitioning column to a USING request modifier variable, including host variables, or with built-in function such as CURRENT_DATE or CURRENT_TIMESTAMP. The database might not cache such plans because cached plans must be suitably generalizable to handle changes in the cached expressions in subsequent executions of the plan (see [Parameterized Requests](#)). However, there are still optimization opportunities available for such conditions. For example, in certain cases, the system can delay row partition elimination until it builds the finalized plan from a cached plan using the values for this specific execution of the plan.

Equality Conditions That Define a Single Combined Partition

Delayed row partition elimination occurs for USING request modifier variables and built-in functions for equality over all row partitioning levels on all partitioning columns. In other words, constraints must exist such that a single combined partition of the combined partitioning expression is specified when the system applies the values of the USING request modifier variable or built-in function to build a finalized plan from a cacheable plan.

Some of the equality conditions might be constant conditions. No restrictions are defined on the form of the partitioning expressions; however, the following restriction does exist on the form of an equality constraint: it must be a simple equality condition between a partitioning column and any of the following:

- USING request modifier variable
- Built-in function
- Constant expression

There must be an ANDed equality condition on the partitioning columns such that a single combined row partition of the combined partitioning expression is specified.

For example, consider the following table definition:

```
CREATE TABLE markets (
  productid      INTEGER NOT NULL
  region         BYTEINT NOT NULL
  activity_date   DATE FORMAT 'yyyy-mm-dd' NOT NULL
```



```

revenue_code    BYTEINT NOT NULL
business_sector BYTEINT NOT NULL
note            VARCHAR(256)
PRIMARY INDEX (productid, region)
PARTITION BY RANGE_N(region          BETWEEN 1
                        AND           9
                        EACH          3)
                RANGE_N(business_sector BETWEEN 0
                        AND           49
                        EACH          10)
                RANGE_N(revenue_code   BETWEEN 1
                        AND           34
                        EACH           2)
                RANGE_N(activity_date   BETWEEN DATE '1986-01-01'
                        AND           DATE '2007-05-31'
                        EACH INTERVAL '1' MONTH))

```

This is the same table that was defined in [Examples of Rewrites Using Row Partitioning](#).

For the sake of simplicity, the comments associated with the example queries make the following assumptions:

- Rows are evenly distributed among the combined partitions.
- There are many data blocks per row partition.

If the example queries were not able to capitalize on delayed row partition elimination, they would all require all-AMP, full-table scans to be processed unless otherwise noted.

This example assumes a 2-byte internal partition number. For the following query, which specifies both a set of USING request modifier values and a built-in function, the database reads one combined row partition on all AMPs (combined because the conditions in the predicate are on all row partitioning levels of the table). As a result, it reads only 1/65,535 of the data blocks for the markets table because with all partitioning levels specified in the predicate, all 65,535 combined partitions are eligible to be scanned, but the database needs to read only the one combined row partition of those possible 65,535 combined row partitions to access all of the qualified rows.

The combined row partition might start and end in the middle of data blocks, and those data blocks might include rows whose values for the query predicate do not qualify. The database necessarily reads those rows as it reads the data blocks for the combined row partition, but because they do not qualify for the query predicate, the system does not return them in the result set.

```

USING (r BYTEINT, b BYTEINT, rc BYTEINT)
SELECT *
FROM markets
WHERE region=:r
AND    business_sector=:b

```

```
AND    revenue_code=:rc
AND    activity_date=CURRENT_DATE;
```

Delayed Row Partition Elimination and Character-Partitioned Tables

Partition row elimination can occur with conditions comparing a partitioning column to a USING variable, including host variables, or a built-in function such as CURRENT_DATE).

The following additional conditions must be met for the Optimizer to apply delayed row partition elimination for a character-partitioned table:

- There must be at most one character partitioning column at all row partitioning levels.
- If the session collation or table collation is MULTINATIONAL or CHARSET_COLL, and if any comparison or string function involving any non-constant expression in a partitioning expression at any level is case insensitive, then the session collation must match the table collation.

For delayed row partition elimination, the following are case-insensitive:

- LOWER
- SOUNDEX
- UPPER
- UPPERCASE qualifier

The following are case-sensitive:

- CHAR2HEXINT
- TRANSLATE
- TRANSLATE_CHK
- VARGRAPHIC

The concatenation operator is considered to be both case-sensitive and case-insensitive.

The following functions follow the same rules as comparison operators, and the database examines the function inputs along with default case sensitivity for the session mode, either ANSI or Teradata, to determine case sensitivity:

- INDEX
- MINDEX
- POSITION

The presence of the SUBSTRING function does not affect case sensitivity.

- If the WHERE clause condition qualifying a level for delayed row partition elimination is case-insensitive, then all comparisons in the partitioning expression for that level must be case-insensitive, and all string functions involving any non-constant expression of the partitioning expression must be case insensitive.

A WHERE clause condition is considered to be case-insensitive if any of the comparisons or string functions involving non-constant expressions in the condition is case-insensitive.

If the equality condition that qualifies a level for delayed row partition elimination is case-insensitive, then all comparisons at a given row partitioning level must also be case-insensitive. While this must be true for all row partitioning levels, case sensitivity does not need to match between different row partitioning levels.

If a character partitioning level involves any case specific comparisons of character data, then the corresponding ANDed equality conditions involving the partitioning columns for the level must also be case specific.

Assume you have created the following table, which is used in the SELECT request example that follows:

```
CREATE SET TABLE user_IDs, NO FALLBACK, NO BEFORE JOURNAL,  
                                NO AFTER JOURNAL, CHECKSUM = DEFAULT (  
    user_ID    INTEGER,  
    user_name CHARACTER(40) CHARACTER SET UNICODE CASESPECIFIC)  
PRIMARY INDEX (user_ID)  
PARTITION BY RANGE_N(user_name BETWEEN 'A','Z','a','y' AND  
                        'zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz',  
                        NO RANGE OR UNKNOWN);
```

For the sake of simplicity, assume an even distribution of the rows among the row partitions and many data blocks per row partition. Without this form of delayed row partition elimination, the request would require an all-AMP, full-table scan to return the requested result set.

Because it uses delayed row partition elimination, the database reads one row partition on all AMPs. The system substitutes the value of the built-in function `USER` for `user_name`; therefore, the database only needs to read roughly 1/5 of the data blocks for *user_IDs*.

Because the data blocks containing the row partition include some non-qualifying rows in the first and last of these data blocks, the database must also read them as it reads the row partition, but it does not return those rows as part of the result set.

```
SELECT *
FROM user_IDs
WHERE user_name=USER;
```

Dynamic Row Partition Elimination

Dynamic row partition elimination (DPE) is a form of partition elimination that occurs after data values in the tables are known in order to eliminate partitions: the database cannot determine the partitions to be eliminated until the plan is executed and the data is scanned. Because of this, dynamic partition elimination must be applied on the AMPs after query optimization has already taken place.

Several forms of dynamic row partition elimination exist such as DPE for merge, hash, or product joins, DPE that occurs with a rowkey-based merge join, and others. See [Product Joins With Dynamic Row Partition Elimination](#) and [Product Join With Dynamic Row Partition Elimination for Character Partitioning](#) for some applications of dynamic row partition elimination.

Single Partition Scans and BEGIN/END Bound Functions

An access plan can scan a single row partition when there are equality constraints specified on BEGIN or END bound functions in the WHERE clause of a SELECT request and a row partitioning level of the table is defined on those functions.

Rules for Performing Single-Partition Scans

The following table explains the rules for performing single-partition scans in this situation:

WHEN a primary index is row partitioned on ...	AND if the WHERE clause of a primary index retrieval from the table specifies ...	THEN the database ...
BEGIN bound function	an equality constraint on the BEGIN function	accesses only a single row partition.
	an equality constraint on the END function	does not access only a single row partition.
	an equality constraint on both the BEGIN and END functions	accesses only a single row partition.
END bound function	an equality constraint on the BEGIN function	does not access only a single row partition.
	an equality constraint on the END function	accesses only a single row partition.
	an equality constraint on both the BEGIN and END functions	accesses only a single row partition.
both a BEGIN bound function and an END bound function	an equality constraint on only the BEGIN function	does not access only a single row partition.
	an equality constraint on only the END function	does not access only a single row partition.
	both the BEGIN and END functions	accesses only a single row partition.

Examples of Single-Partition BEGIN or END Bound Function Scans on Partitioned Tables

Assume that you define the following two tables:

```
CREATE SET TABLE t11 (
  a INTEGER,
  b PERIOD(DATE))
PRIMARY INDEX(a)
PARTITION BY CAST ((BEGIN(b)) AS INTEGER);
CREATE SET TABLE t12 (
```

```

a INTEGER,
b PERIOD(DATE))
PRIMARY INDEX(a)
PARTITION BY CAST((END(b)) AS INTEGER);

```

The relevant EXPLAIN phrase text in the reports for the examples that follow is highlighted in boldface type.

The following SELECT request uses an single-partition scan to access the rows:

```

EXPLAIN
SELECT *
FROM t11
WHERE BEGIN(b)=DATE '2005-02-03';

```

The following shows a portion of the EXPLAIN output:

```

...
3) We do an all-AMPs RETRIEVE step from a single partition of
df2.t11 with a condition of ("df2.t11.b = PERIOD
(DATE '2005-02-03', DATE '2005-02-03')")
with a residual condition of ("BEGIN(df2.t11.b )
= DATE '2005-02-03'") 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 (56 bytes). The estimated time for this step
is 0.03seconds.
...

```

The following SELECT request uses an all-rows, all-partitions scan to access the rows because the predicate and partitioning expression are defined on different bounds:

```

EXPLAIN
SELECT *
FROM t11
WHERE END(b)=DATE '2005-02-03';

```

```

...
3) We do an all-AMPs RETRIEVE step from df2.t11 by way of an
all-rows scan with a condition of ("END(df2.t11.b )= DATE
'2005-02-03'") 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 (56 bytes). The estimated time for this step
is 0.03 seconds.
...

```

The following SELECT request uses a single-partition scan to access the rows:

```
EXPLAIN
SELECT *
FROM t12
WHERE END(b)=DATE '2006-02-03';
```

...

3) We do an all-AMPs RETRIEVE step **from a single partition of df2.t12** with a condition of `("df2.t12.b = PERIOD (DATE '2006-02-03'- INTERVAL '1' DAY, DATE '2006-02-03')")` with a residual condition of `("(END(df2.t12.b))= DATE '2006-02-03'")` 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 (56 bytes). The estimated time for this step is 0.03 seconds.

...

The following SELECT request uses a single-partition scan to access the rows:

```
EXPLAIN
SELECT *
FROM t12
WHERE a=1
AND   END (b)=DATE '2006-02-03';
```

1) First, we do a single-AMP RETRIEVE step **from a single partition of df2.t12** by way of the primary index `"df2.t12.a = 1, df2.t12.b = PERIOD (DATE '2006-02-03'-INTERVAL '1' DAY, DATE '2006-02-03')"` with a residual condition of `("((END(df2.t12.b)) = DATE '2006-02-03') AND (df2.t12.a = 1)")` into Spool 1 (one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with low confidence to be 1 row (56 bytes). The estimated time for this step is 0.02 seconds.

...

Column Partitioning

In some ways, a column store and vertical partitioning are similar. In practice, a column store usually means each column of a table is stored separately, but a user views the table as one object, while vertical partitioning typically means that multiple columns are partitioned into separate tables and a view is defined over those tables to cause a user to perceive the set of tables as a single database object. If you only put one column in a vertical partition or allow a column store to have multiple columns in a structure and put that structure in one column, the 2 concepts are very similar from a functionality perspective.

The manner in which a column store and a vertically partitioned object are stored physically, however, is distinctively different. Vertical partitioning typically means that each of the vertical partitions is stored in separate tables using a row store that could be organized with different primary and secondary indexes, different partitioning, and so on so there is sometimes more flexibility, but it comes with the potential cost of requiring more space or CPU overhead to link columns of a row together.

When a column store is used, each column value is appended to its column store so a corresponding value for one column has the same relative position as the corresponding values for the "row" in other column partitions. This can produce a lower space overhead and more efficient processing of predicates. Because many columns, particularly the important columns in the table, are in single-column partitions, there are many opportunities to compress a column store that do not occur if there are multiple columns in a partition as is typical for vertical partitioning.

The column partitioning used by Vantage is more closely related to a column store than to vertical partitioning, but the column partitioning used by Vantage also supports multiple columns in a column partition.

The Vantage implementation of a column store is referred to as column partitioning because it builds on the row partitioning and it is efficient for storing columns individually in partitions, which it does by using containers as the basis for storing column values in a manner that is similar to a column store, and by allowing a hybrid approach that also permits a row store for a column partition.

Column partitioning can be considered to be a hybrid of column store characteristics and vertical partitioning characteristics, with a word from each description taken to form the descriptive phrase *column partitioning*.

You can either use column partitioning by itself or together with row partitioning efficiently in SELECT, INSERT, UPDATE (excluding UPSERT form), and DELETE query execution plans. You cannot use either the Upsert form of UPDATE or MERGE requests, however, if the target table of the request is column-partitioned.

Column Partition Elimination

Column-partitioned tables are subject to *column partition elimination* during query processing. Column partition elimination enhances query performance by allowing the Optimizer to ignore column partitions that are not referenced by a query.

Search conditions applied to the remaining partitions can reduce how much of those partitions must be scanned, further enhancing performance.

If multiple columns are required for the response set for a query, the query plan includes putting projected column values from selected table rows together to form result rows. Vantage can combine this with row partition elimination to further reduce the data that must be accessed to satisfy a query.

Access Planning for Column-Partitioned Objects

Access planning for column partitions is driven by the single-table predicates and the projection list on a column-partitioned database object. The Optimizer groups the single-table predicates and the projection list for a table or join index by the column partitions that contain the referenced columns.

This process includes creating a list of conditions that can be evaluated on the same column partition or set of column partitions.

When there are both single-table predicates and projection columns, predicate ordering takes precedence over projection analysis because evaluating predicates eliminates rows that would project the required columns. When there are insufficient available column partition contexts for both, the Optimizer employs various heuristics to determine whether to apply the predicates and merge the projection columns into a column partition merge spool or only to apply the predicates and output row IDs of the qualified rows to the column partition merge spool.

Join Planning and Optimization

This section provides information about fundamentals of join planning and a cross section of join optimizations. Note that not all optimizations are described. Use an EXPLAIN to see a summary of the optimizations done for your queries.

The information provided is designed to help you to interpret EXPLAIN reports more accurately, and provides some recommendations for how you can improved your plans. For more information about how to optimize the performance of your database from other perspectives, see *Teradata Vantage™ - Database Design*, B035-1094.

Optimizer Join Plans

Selecting the optimum method to join relations is as critical to the performance of a query as the selection of table access methods.

There are many ways to join relations. The method the Optimizer ultimately chooses depends on multiple factors, including the comparison on which the join is made, the estimated cardinality of the relations being joined, the configuration of the system on which the join is to be made, and so on.

This section describes join processing at several levels.

Components of a Join Plan

Join planning involves the following core components:

- Selecting a join method.

There are often several possible methods that can be used to make the same join. For example, it is usually, but not always, less expensive to use a merge join rather than a product join. The choice of join method often has a major effect on the overall cost of processing a query.

- Determining an optimal join geography.

Different methods of relocating rows to be joined can have very different costs. For example, depending on the size of the relations in a join operation, it might be less costly to duplicate one of the relations rather than redistributing it.

- Determining an optimal join order.

The sequence in which relations are joined can have a powerful impact on join cost. In this context, a table could be joined with a spool rather than another table. The term *table* is used in the most generic sense of the word, and the logical term *relation* is often used as a substitute for any table, view, or spool.

- Determining the optimal spool format.

Teradata can select regular format or in-memory format, depending on the cost. The cost model uses various parameters, including the production of an in-memory spool. In some cases the optimizer converts a regular table to an in-memory spool format table.

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

Term Type	Definition	Example
Bind	An equality constraint between two columns of different relations. Normally used to determine whether there are conditions on an index.	(t1.a= t2.a)
Cross	A predicate with expressions on different relations. A cross term can be generated for a condition of the form column_name=expression, which is referred to as a <i>half cross term</i> . The Optimizer can use a half cross term as a constraint if it is specified on an index. The form conversion(column)=expression can be used for the same purpose if conversion(column) and column are hashed the same.	(t1.a= t2.a+1)
Exists	A predicate that specifies an EXISTS condition.	
Explicit	A predicate defined on a constant. 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. The set of miscellaneous terms includes the following: <ul style="list-style-type: none"> • Inequality terms. • Equality terms on either the same set of relations or on non-disjoint sets of relations. • ANDed terms. • ORed terms. • Other terms (for example, terms expressed over more than 2 relations). A miscellaneous term can be generated for a conversion(column) = constant condition. If conversion(column) and (column) hash the same, then the miscellaneous term points to an explicit term. For example, if the operation undertaken by the conversion is a simple renaming of the column. In this case, the miscellaneous term is used to determine whether a constraint is specified on an index column.	
Outer join	An outer join term.	

The first thing the Optimizer looks for when planning a join is connecting conditions, which are predicates, that connect an outer query and a subquery. The following are all examples of such 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)
```

Additionally, the Optimizer analyzes conditions to determine the connections between relations in a join operation:

- There is a direct connection between two relations if either of the following conditions is found:
 - An ANDed bind, miscellaneous, cross, outer join, or minus term that satisfies the dependent info between the two relations.
 - A spool of a noncorrelated subquery EXIST condition connects with any outer relation.
- An ANDed miscellaneous or explicit term on a single relation is pushed to the relation.
- A term on no relation is pushed to a relation.
- An ORed term that references some subqueries and a single relation is associated with that relation as a complex term.
- All relations that are referenced in an ORed term that specifies subqueries and more than one relation are put into complex set.
- All relations that are specified in some join condition are marked as connected.
- Assume selection and projection are done if a relation is spooled before join, for example:

```
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.
 - Its primary index, if it has one.
 - The set of connected relations such as the following:

```
SELECT t1.x1
FROM t1, t2
WHERE y1=1
AND    x1= x2;
```

- Find the following information about all base table relations in the set of input relations:
 - The relation row size.
 - The relation cardinality.
 - The selection condition list.
 - The projection list.
 - The best possible access paths (using calls to access planning functions).
- Find the following information about all spool relations in the set of input relations:

- The spool cardinality
- The selection condition list.
- Its assignment list.
- Its spool number.

Join Processing Methods

Depending on the indexes defined for the tables involved and whether statistics are available for the indexes, the Optimizer processes a join using a join method such as one of the following:

- Product join
- Hash join
- Merge join
- Nested join
- Exclusion join
- Inclusion join
- RowID join
- Correlated join
- Minus all join

See [Join Strategies and Methods](#) and the pages following for more information about the various join methods.

Limit on the Number of Tables or Single-Table Views That Can Be Joined

Excluding self-joins, as many as 128 tables or single-table views can be joined per query block. The maximum number of tables and single-table views that can be joined per query block is determined by the value of the MaxJoinTables performance field of the DBS Control record, which ranges from a minimum of 64 to a maximum of 128. See *Teradata Vantage™ - Database Utilities*, B035-1102 for details.

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:

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

Each reference to a relation, including those using correlation names, counts against the limit of 128 tables. This limit includes implicit joins such as the join of a hash or single-table join index to a base table to process a partial cover.

For example, consider a query with a single noncorrelated subquery. The subquery is limited to 128 tables and the outer query is limited to 127 tables, the 128th table for the outer query being the spooled result of the inner query that must be joined with it.

If the noncorrelated subquery were an outer query to an additional noncorrelated subquery, then the deepest subquery would be limited to referencing 128 tables, its outer query limited to 127 (127 plus the result of its inner query), and the parent outer query to 127 plus the result of its inner query.

In summary, while the number of tables, including intermediate spool relations, that can be joined is limited to 128 per query block, the cumulative number of tables referenced in the course of optimizing the query can be considerably greater than 128.

There is no way to determine a priori how many query blocks will be created and processed by the Optimizer in the course of producing a join plan, but the factors listed here are all candidates to evaluate if your queries terminate because they exceed the 128 table join limit.

Recommendation for Joins and Domains

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 *Teradata Vantage™ - Data Types and Literals*, B035-1143 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 *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for information about UDTs. See *Teradata Vantage™ - Database Design*, B035-1094 for information about using UDTs to define domains.

Amount of Parser Memory Required for Large Joins

If you plan to join more than 32 relations in a request, use the DBS Control Utility to increase the value for the MaxParseTreeSegs field (in the Performance group) to something greater than the default number of parse tree memory segments allocated to Parser memory for code generation. For more information about the DBS Control utility, see *Teradata Vantage™ - Database Utilities*, B035-1102

The MaxParseTreeSegs field defines the maximum number of parse tree memory segments the Parser allocates when parsing an SQL request. The database allocates the parse tree segments dynamically as they are needed, so this field does not unnecessarily preallocate memory that might not be used.

Evaluating Join Costing

Because that the Optimizer is cost-based, it evaluates the relative costs of the available join methods (see [Join Strategies and Methods](#)) to determine the least expensive method of joining two relations.

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

If the estimated cardinality of the small relation is 5 rows, then each row in the right relation is estimated to make 5 comparisons. If the estimate is 500 rows, the estimated number of comparisons is 500. This is a difference of two orders of magnitude. In the former case, the product may be the least costly method, while in the later case, another join method may be less costly.

Note that the ordering of joins is a separate process from the selection of methods to use for those joins. See [Determining the Order of Joins](#) for information about how the Optimizer evaluates join orders.

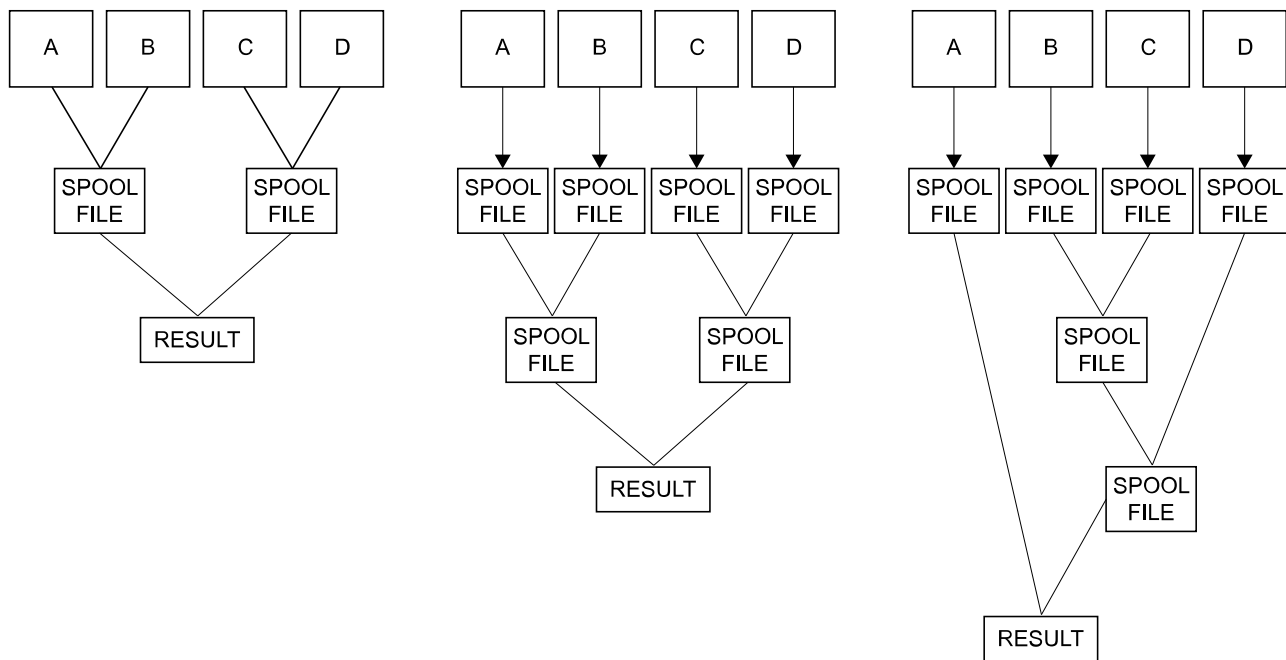
Planning *n*-way Joins

If more than two relations are specified to be joined, the Optimizer considers joining the relations as a series of binary joins and then attempts to determine the most cost-effective join order. the Optimizer, in some cases, also considers joining a subset of the relations using an *n*-way join in a single step.

For example, assume you submitted the following multitable request:

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

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



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

It is always advantageous to reduce the number of rows and columns that must be duplicated or redistributed. However, if a join can be done directly with the base table, it may be more efficient for column projection and row selection to be done in the join step after determining if two rows qualify to be joined.

Join Order Search Trees

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

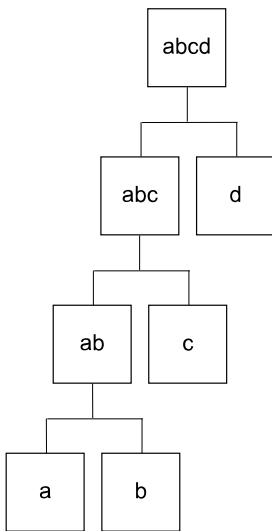
Left-Deep Search Trees

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

Number of join orders = $n!$

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

The following diagram illustrates a left-deep join tree for a 4-relation join, reading from the bottom up:



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, so the Optimizer uses several heuristics to prune the search space. For example, with the exception of star joins, bushy search trees are not considered for unconstrained joins.

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

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

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

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

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

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

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


•
•
•

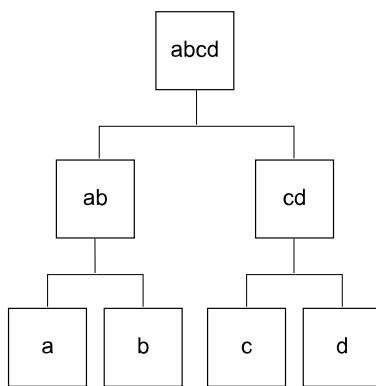
and so on.

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

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

The illustrated process can be described as follows:

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



Depending on table demographics and environmental costs, the Optimizer could just as likely produce the following join sequence:

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

The Optimizer 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.

Join Geography

Join geography is a term that describes the relocation, if any, of rows required to perform a join. Remember that for two rows to be joined, they must be on the same AMP, so the Optimizer must determine the least costly row relocation strategy, and the relative cardinalities of the two relations in any join is the principal cost

that must be considered. This is one of the main factors that compels having fresh statistics available for the Optimizer.

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:

- Hash-based 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 relation.

The EXPLAIN phrase to watch for is *Locally Build*.

- Randomly redistributing one of the relations.

After relocating a relation, the relation may need to be sorted. 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 [AMP] indexes of the two relations are such that equal-valued 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.

Redistributing the Rows of One Table for a Merge Join

This example illustrates a merge join strategy, which includes hash 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.

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 tables below show the columns and data of the two tables to be joined.

Employee Table

enum (PK, UPI)	name	dept (FK)
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 Table

dept (PK, UPI)	name
150	Payroll
200	Finance
310	Manufacturing
400	Engineering

The following graphic shows how individual rows would be relocated to make this join for a 4-AMP system:

Employee table rows distributed and sorted based on the row hash of e.enum, the UPI.

AMP1	AMP2	AMP3	AMP4
6 Kaczmarek 400 8 Challis 310	4 Korlapati 400 3 Chiang 310	1 Higa 200 7 Eggers 310	5 Sinclair 150 2 Kostamaa 310

Spool file after redistribution and sorting based on the row hash of e.dept.

AMP1	AMP2	AMP3	AMP4
5 Sinclair 150	7 Eggers 310 3 Chiang 310 8 Challis 310 2 Kostamaa 310	1 Higa 200	6 Kaczmarek 400 4 Korlapati 400

Department table rows distributed and sorted based on the new hash of d.dept.

AMP1	AMP2	AMP3	AMP4
150 Payroll	310 Manufacturing	200 Finance	400 Engineering

The example is run on a 4-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 (dept), but not both, of the tables.

Duplicating and Sorting the Rows of the Smaller Table on all AMPs, Building and Sorting 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.

This query and tables used for this example are the same as those used for [Redistributing the Rows of One Table for a Merge Join](#), but the Optimizer pursues a different join geography strategy because the statistics for the tables are different. If the Optimizer determines from the available statistics that it would be less expensive to duplicate and sort the smaller table than to hash redistribute the larger table, it chooses the strategy followed by this scenario.

The following graphic shows how individual rows would be relocated to make this join for a 4-AMP system

Spool file after duplicating the department table and sorting on d.dept row hash.

AMP1	AMP2	AMP3	AMP4
150 Payroll 200 Finance 310 Manufacturing 400 Engineering	150 Payroll 200 Finance 310 Manufacturing 400 Engineering	150 Payroll 200 Finance 310 Manufacturing 400 Engineering	150 Payroll 200 Finance 310 Manufacturing 400 Engineering

Spool file after locally building and sorting on the e.dept row hash.

AMP1	AMP2	AMP3	AMP4
8 Chiallis 310 6 Kaczmarek 400	3 Chiang 310 4 Korlapati 410	1 Higa 200 7 Eggers 310	5 Sinclair 150 2 Eggers 310

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.

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. 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 Table

enum (PK, FK, NUPI)	name (PK)	dept (PK)
1	Higa	200
2	Kostamaa	310
3	Chiang	310
4	Korlapati	400
5	Sinclair	150

enum (PK, FK, NUPI)	name (PK)	dept (PK)
6	Kaczmarek	400
7	Eggers	310
8	Challis	310

Employee_Phone Table

enum	area_code	phone
1	213	5551576
2	213	5550703
3	408	5558822
4	415	5557180
5	312	5553513
6	203	5557461
7	301	5555885
8	301	5551616

The following graphic shows how individual rows would be relocated to make this join for a 4-AMP system:

Employee table rows distributed and sorted based on the row hash of e.enum, the UPI

AMP1	AMP2	AMP3	AMP4
6 Kaczmarek 400 8 Chiallis 310	4 Korlapati 400 3 Chiang 310	1 Higa 200 7 Eggers 310	5 Sinclair 150 2 Kostamaa 310

Employee_Phone table rows distributed and sorted based on the row has of enum, the NUPI.

AMP1	AMP2	AMP3	AMP4
6 203 8337461 8 301 2641616 8 301 6675885	4 415 6347180 3 408 3628822	1 213 3241576 1 213 4950703	5 312 7463513

Duplicating the Smaller Table on Every AMP for a Product Join

This example illustrates a product join strategy, which includes duplicating the smaller table rows on every AMP of a 4-AMP system. The tables used for this example are the same as those used for [Redistributing the Rows of One Table for a Merge Join](#).

The following query generates a product join:

```
SELECT *
FROM employee AS e INNER JOIN department AS d
ON e.dept > d.dept;
```

The product join is caused by the non-equi join condition `e.dept > d.dept` (see [Product Join](#)). Based on the available statistics, the Optimizer determines that the department table is the smaller table in the join, so it devises a join plan that distributes copies of all those rows to each AMP. employee table rows, which are hash distributed on their UPI enum values, are not relocated. The product join operation returns only the rows that satisfy the specified join condition for the query after comparing every row in the employee table with every row in the duplicated department table.

The following graphic shows how individual rows are relocated to make this join for the 4-AMP system.

Spool file after duplicating Department table rows.

AMP1	AMP2	AMP3	AMP4
310 Manufacturing 400 Engineering 200 Finance 150 Payroll	310 Manufacturing 400 Engineering 200 Finance 150 Payroll	310 Manufacturing 400 Engineering 200 Finance 150 Payroll	310 Manufacturing 400 Engineering 200 Finance 150 Payroll

Relocation Scenarios

The primary [AMP] index is the major consideration used by the Optimizer in determining how to join two tables and deciding which rows to relocate, though for tables that have no primary index, other criteria must be considered.

Three general scenarios can occur when two primary-indexed relations are joined using the merge join method (see [Merge Join](#)), as illustrated by the following table.

The following 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 [AMP] index	both.
2	2	the primary [AMP] index	one.
3	3	not the primary [AMP] index	neither.

Determining the Order of Joins

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

Join order evaluation is a critical aspect of query optimization that is highly dependent on accurate statistics. Using out of date statistical and demographic data can result in inaccurate assumptions about the cardinalities of intermediate results. Such inaccurate assumptions cause errors in join plan estimation that propagate exponentially as a function of the number of joins a query makes.

Join Order Evaluation Process Overview

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

1. Generate a 1-join lookahead plan.
- 2.

IF the following conditions are found ...	THEN ...
<ul style="list-style-type: none"> • The cost of the 1-join lookahead plan exceeds a threshold value. • There exists one relation larger than a threshold value. • The query has no outer join requirement. 	Generate a 5-join lookahead plan and evaluate its cost. Keep the less-costly plan from the 1- and 5-join lookahead plans.

- 3.

IF the following conditions are found ...	THEN ...
<ul style="list-style-type: none"> • The cost of the current plan exceeds a threshold value. • A star join might be required. 	Generate a plan that uses star/snowflake join optimization, if such a plan can be found. If it is, compare the star/snowflake and lookahead join plans, and keep the less-costly plan.

Lookahead Join Planning

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 forward one or more plans to examine the ultimate cost of joining relations in various orders.

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

The following is a simplified explanation of lookahead join planning:

1. Find the best 3-way or n -way join plans among the combinations considered.

Follow these rules to find them:

- Use a depth-first search.

- Skip a join plan if any one of the following conditions is detected:
 - A less costly plan exists that delivers a result with similar or better attributes.
 - The accumulated cost exceeds that of the current candidate join plan.
- Only consider joins between connected relations.
- Join an unconnected relation only after all connected relations have been joined.

The Optimizer does not evaluate all possible combinations of relations because the effort would exceed any optimizations realized from it. For example, a 10-way join has 17.6×10^9 possible combinations, and a 64-way join has 1.2×10^{124} possible combinations.

In its pursuit of combinations that are driven mainly by join conditions, the Optimizer might overlook the best possible join plan, but in practice, that generally does not happen.

2. Evaluate the best current join plan.

IF the ...	THEN ...
following conditions are found for the first 2 joins of the best join plan. <ul style="list-style-type: none"> • The joins involve one relation connected with 2 relations not connected to each other • Neither join is a product join 	try a compromise join where the 2 unconnected relations are product-joined in the first join and that result is then joined with a third relation.
compromise join plan is less costly than the best current join plan	replace the first 2 joins on the best join plan with the compromise plan.

The compromise plan is designed to enable a star join plan.

3. Materialize the first join of the best n -way join 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 relation with the result of the first join via the primary index for the newly joined relation.

4. Generate connection information for new relations based on the predicates.
5. Iterate Stages 1 through 4 until only one active relation remains.

Example: Compromise Join Plan

Consider the following request.

```

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

```

The first join diagram in the following graphic illustrates the connections among the relations in this query.

The explicit connections are these.

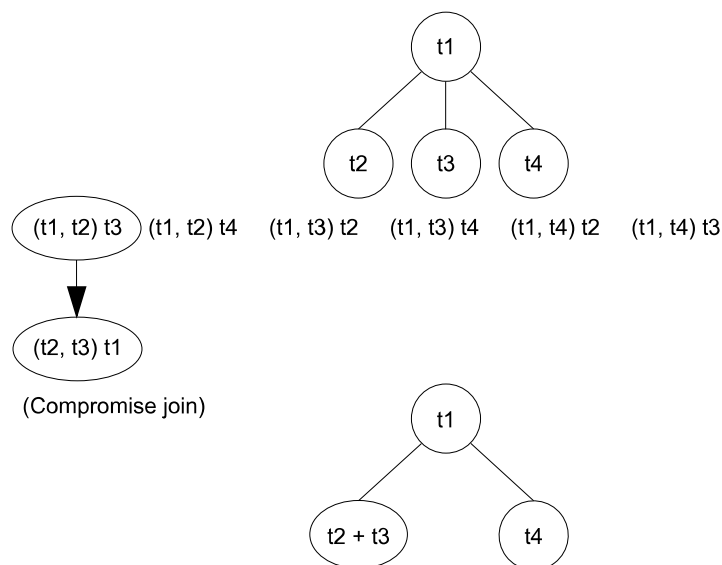
- t1 is connected to t2 because of the condition $x1=x2$
- t1 is connected to t3 because of the condition $y1=y3$
- t1 is connected to t4 because of the condition $z1=y4$

We know from step 2 that when the following conditions are true, the Optimizer tries to make a compromise join where the two unconnected relations are product-joined and the result is joined with the third relation:

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

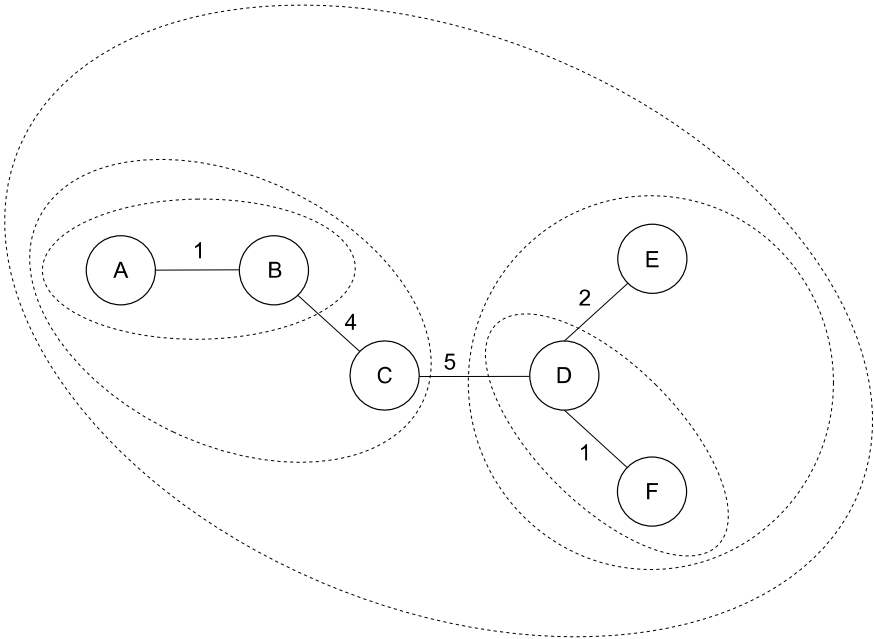
The second join diagram in the graphic indicates that the join of t1 with t2 followed by a join of the resulting relation t1t2 with t3 is not as good as the compromise join of the unconnected relations t2 with t3 followed by a join of the resulting relation t2t3 with t1.

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



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

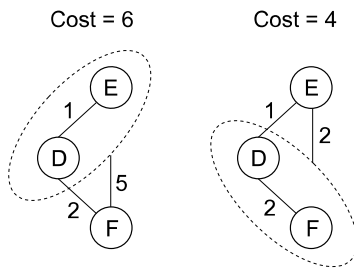
The following graphic illustrates the result of an Optimizer analysis of the join space for a 6-way join. The relative cost of each join is given by an integer within the selected join sequence.



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:



The cost relationships among the relations 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 relations D and F first following by joining the resulting relation DF with relation E, is selected for the join plan because it is the less costly join.

Partial GROUP BY Block Optimization

About Partial GROUP BY Block Optimization by the Optimizer

Where appropriate, aggregations over joins are split, with part being done before the join, and part being done after the join. This means less work for the join as well as less aggregation work after the join.

In a join query plan for a query with a GROUP BY (or DISTINCT) clause, Partial GROUP BY (PGB) operations are used to reduce the cardinalities of base tables, join indexes, and intermediate spools, thereby reducing processing required in performing the join query plan. To further enhance efficiency, certain of the partial group operations, including an intermediate Partial GROUP BY operation, a final GROUP BY operation, or both can be eliminated in certain predefined conditions.

The Partial GROUP BY optimization can be done almost at any stage of the join plan. An early GROUP BY reduces the rows of the base relations or the intermediate join relations during the optimization process.

However, GROUP BY and merge join operators have a tendency to compete with one other for use in the lowest cost plan because both apply significant row reduction. Therefore, it is extremely important for the Optimizer to cost the alternatives.

The Partial GROUP BY optimization can reduce the processing and I/O times for typical requests. The more duplicate grouping values a table has, the greater the reduction in I/O processing which, in some cases, can be as much as a factor of 100 times.

Examples of Partial GROUP BY Optimization

Consider the following typical query:

```
SELECT b1, c1, SUM(t1.float_1), SUM(t2.float_2), SUM(t3.float_3)
FROM t1,t2,t3
WHERE b1=b2 AND b2=b3
GROUP BY b1,c1;
```

Without Partial GROUP BY enabled, the Optimizer selects one of the following join plans:

- [(t1 X t2) X t3]'
- [(t2 X t3) X t1]'
- [(t1 X t3) X t2]'

where X represents the join operator and []' denotes the GROUP BY operator applied to the relation.

With Partial GROUP BY enabled, some additional join plans are considered, such as the following:

- [([t2]' X t3) X t1]'
- [(t1 X [t3]') X t2]'
- [([t2]' X [t3]') X t1]]'
- [([t1 X t3]' X [t2]'
- ...

This provides the Optimizer with an opportunity to find a lower cost join plan among a larger set of possible join plans. Even though there are more join plans to be considered, the additional search effort consumes negligible CPU time and returns a significant reduction time in the execution of the request.

Consider the following example:

```
SELECT ps_partkey, SUM(l_quantity), SUM (ps_supplycost)
FROM partsupp, lineitem
WHERE l_partkey = ps_partkey
GROUP BY 1;
```

Without Partial GROUP BY enabled, the Optimizer redistributes lineitem on the join column l_partkey, performs the join with partsupp, and then aggregates the result of the join. With a one terabyte database, this redistributes 6 billion rows and performs the join with partsupp to produce a result set of 6 billion rows on which the aggregation is performed.

With Partial GROUP BY enabled, the Optimizer first separately aggregates the relations partsupp and lineitem and then joins the results of those aggregations.

Without the transformation, approximately 85 billion rows must be read and written. This is largely because the join of the 6 billion row lineitem relation is joined to the 800 million row partsupp relation to produce a 24 billion row join result spool for aggregation.

With the transformation, the lineitem relation is aggregated as part of the sort and redistribution operation to produce a 200 million row spool. The partsupp relation is locally aggregated to produce another 200 million row spool. The two spools are then joined to produce a 200 million row result, and there is an overall reduction of about 3 times in the number of rows read and written.

Identifying Partial GROUP BY Operations in EXPLAIN Report Text

There are two ways to coalesce rows using Partial GROUP BY:

- Partial SUM steps
- Sort/Group steps

Early GROUP BY With a Partial SUM Step

This section demonstrates the EXPLAIN phrase *partial SUM*.

Consider the following query:

```
SELECT l_partkey,SUM(l_quantity),SUM(ps_supplycost)
FROM partsupp, lineitem
WHERE l_partkey = ps_partkey
GROUP BY 1;
```

Without Partial GROUP BY enabled, the join plan would be [partsupp X lineitem]'. With Partial GROUP BY enabled, the join plan is [partsupp]' X [lineitem]'.

The following EXPLAIN text compares two explanations of this query: the first with Partial GROUP BY disabled and the second with Partial GROUP BY enabled.

This is partial EXPLAIN text with Partial GROUP BY disabled.

```
...
4) We do an all-AMPS RETRIEVE step from TPCD50G.lineitem by way of an all-rows scan
   with no residual conditions into Spool 4 (all_amps), which is redistributed by hash
   code to all AMPS. Then we do a SORT to order Spool 4 by row hash. The result spool
   file will not be cached in memory. The size of Spool 4 is estimated with high
   confidence to be 300,005,811 rows. The estimated time for this step is 2 hours and
   51 minutes.
5) We do an all-AMPS JOIN step from TPCD50G.parsupp by way of a RowHash match
   scan with
   no residual conditions, which is joined to Spool 4 (Last Use). TPCD50G.parsupp and
   Spool 4 are joined using a merge join, with a join condition of
   ("L_PARTKEY = TPCD50G.parsupp.PS_PARTKEY").
   The input table TPCD50G.parsupp will not be cached in memory, but it is
   eligible for
   synchronized scanning. The result goes into Spool 3 (all_amps), which is built-
   locally on the AMPS. The result spool file will not be cached in memory.
   The size of Spool 3 is estimated with low confidence to be 1,200,023,244 rows.
```

The estimated time for this step is 2 hours and 46 minutes.

- 6) We do an all-AMPs SUM step to aggregate from Spool 3 (Last Use) by way of an all-rows scan, and the grouping identifier in field 2. Aggregate Intermediate Results are computed locally, then placed in Spool 5. The aggregate spool file will not be cached in memory. The size of Spool 5 is estimated with low confidence to be 1,200,023,244 rows. The estimated time for this step is 10 hours and 6 minutes.

This is partial EXPLAIN text with Partial GROUP BY enabled.

```

...
4) We do an all-AMPs partial SUM step to aggregate from TPCD50G.partsupp by way of
an all-rows scan with no residual conditions, and the grouping identifier in
field 1.
Aggregate Intermediate Results are computed locally, then placed in Spool 6.
The input table will not be cached in memory, but it is eligible for synchronized
scanning. The aggregate spool file will not be cached in memory. The size of
Spool 6
is estimated with low confidence to be 10,000,000 rows. The estimated time for
this step is 12 minutes and 36 seconds.
5) We execute the following steps in parallel.
1) We do an all-AMPs RETRIEVE step from Spool 6 (Last Use) by way of an all-
rows scan
into Spool 5 (all_amps), which is built locally on the AMPs. Then we do a SORT
to order Spool 5 by row hash. The result spool file will not be cached in
memory. The size of Spool 5 is estimated with low confidence to be
10,000,000 rows.
2) We do an all-AMPs partial SUM step to aggregate from TPCD50G.lineitem by
way of an all-rows scan with no residual conditions, and the
grouping identifier
in field 1. Aggregate Intermediate Results are computed globally, then placed
in Spool 10. The input table will not be cached in memory, but it is eligible
for synchronized scanning. The size of Spool 10 is estimated with
low confidence
to be 10,000,000 rows. The estimated time for this step is 3 hours and
39 minutes.
6) We do an all-AMPs RETRIEVE step from Spool 10 (Last Use) by way an all-rows scan
into Spool 9 (all_amps), which is redistributed by hash code to all AMPs. Then
we do
a SORT to order SPOOL 9 by row hash. The result spool file will not be cached in
memory. The size of Spool 9 is estimated with no confidence to be
10,000,000 rows.
7) We do an all-AMPs JOIN step from Spool 5 (Last Use) by way of a RowHash
match scan,
which is joined to Spool 9 (Last Use). Spool 5 and Spool 9 are joined using a
merge join, with a join condition of ("L_PARTKEY = PS_PARTKEY").
The result goes into Spool 3, (all_amps), which is built locally on the AMPs.
The result spool file will not be cached in memory. The size of Spool 3 is
estimated with low confidence to be 993,739,248 rows. The estimated time for this
step is 2 hours and 16 minutes.

```

With Partial GROUP BY disabled, the SUM step is performed in step 6 where the cardinalities are reduced. In contrast, when Partial GROUP BY is enabled, the partial SUM step is performed in steps 4 and 5.1.

Because the Partial GROUP BY optimization is performed before the join, the EXPLAIN text adds the phrase **partial** before the SUM step for easy identification of the optimization.

Note that it is possible for the Optimizer to select a join plan that does not use this optimization, even though Partial GROUP BY is enabled. This might happen, for example, if the cost of aggregation makes it not worth doing because of the number of small groups.

SORT/GROUP Step and Partial GROUP BY Optimization

This section introduces the phrase *SORT/GROUP* as an identification of GROUP BY. The following request has a nested subquery, with the inner query specifying an OUTER JOIN. Both the inner and outer queries have GROUP BY clauses.

```
SELECT c_count, COUNT(*) AS custdist
FROM (SELECT c_custkey, COUNT(o_orderkey)
      FROM customer LEFT OUTER JOIN ordertbl
        ON c_custkey = o_custkey
        AND o_comment NOT LIKE '%special%requests%'
      GROUP BY c_custkey) AS c_orders (c_custkey, c_count)
GROUP BY c_count
ORDER BY custdist desc, c_count DESC;
```

This is partial EXPLAIN text with Partial GROUP BY enabled.

```
..
4) We do an all-AMPS RETRIEVE step from TPCD50G.ORDERTBL by way of an all-rows scan
   with a condition of ("NOT(TPCD50G.ORDERTBL.O_COMMENT LIKE '%special%requests%')")
   into Spool 5 (all amps), which is redistributed by hash code to all AMPS.
   Then we do a SORT/GROUP to order Spool 5 by row hash and non-aggregate
   fields grouping duplicate rows. The result spool file will not be cached in memory.
   The size of Spool 5 is estimated with no confidence to be 67,500,000 rows.
5) We do an all-AMPS JOIN step from TPCD50G.CUSTOMER by way of a RowHash match scan
   with no residual conditions, which is joined to Spool 5 (Last Use).
   TPCD50G.CUSTOMER and Spool 5 are left outer joined using a merge join,
   with a join condition of ("TPCD50G.CUSTOMER.C_CUSTKEY = O_CUSTKEY").
   The input table TPCD50G.CUSTOMER will not be cached in memory. The result goes
   into Spool 3 (all amps), which is built locally on the AMPS. The result spool file
   will not be cached in memory. The size of Spool 3 is estimated with low confidence
   to be 74,954,952 rows. The estimated time for this step is 15 minutes and
   32 seconds.
6) We do an all-AMPS RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan
   into Spool 1 (all amps), which is built locally on the AMPS. The result spool file
   will not be cached in memory. The size of Spool 1 is estimated with low confidence
   to be 74,954,952 rows. The estimated time for this step is 11 minutes and 6 seconds.
7) We do a SUM step to aggregate from Spool 1 (Last Use) by way of an all-rows scan,
   and the grouping identifier in field 3. Aggregate Intermediate Results are computed
   globally, then placed in Spool 10. The size of Spool 10 is estimated with no
   confidence to be 8,658 rows. The estimated time for this step is 1 minute and
   26 seconds.
```

In this explanation, the phrase *SORT/GROUP* appears in Step 4. That means the Partial GROUP BY is used early to reduce cardinalities.

Eliminating the Last Partial GROUP BY Aggregation for Better Performance

The following are examples of join plans that do not require a final GROUP BY operation.

Assume you submit the following query for the TPC-H benchmark.

```
SELECT l_suppkey,SUM(l_quantity),SUM(ps_availqty)
FROM lineitem,partsupp
```



```
WHERE l_suppkey=ps_suppkey
GROUP BY 1;
```

This query uses the join plan [lineitem]' X [partsupp]'.
Consider the following request:

```
SELECT b1, c1, SUM(t1.float_1), SUM(t2.float_2), SUM(t3.float_3)
FROM t1, t2, t3
WHERE b1=b2
AND b2=b3
GROUP BY 1,2;
```

Without Partial GROUP BY, this query would use the join plan [[t2xt3]'X t1']', with a final GROUP BY operation. Because the columns cover join columns, the last GROUP BY can be skipped, and the join plan can be optimized to [t2xt3]'xt1'.

The last GROUP BY operation need not be performed for the following request:

```
SELECT c_custkey, c_name,
       SUM(l_extendedprice*(1-l_discount)(FLOAT))(DECIMAL(18,2))
       AS revenue, c_acctbal, n_name,
       c_address,c_phone,c_comment
FROM customer, ordertbl, lineitem, nation
WHERE c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate >= '1993-10-01'
AND o_orderdate < DATE '1993-10-01' + INTERVAL '3' MONTH
AND l_returnflag = 'R'
AND c_nationkey = n_nationkey
GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address,
       c_comment
ORDER BY revenue DESC;
```

This query might have two possible join plans depending on the system configuration. Neither plan requires the last GROUP BY operation. The two join plans are as follows:

- ((lineitem X ordertbl)' customer) X nation
- ((ordertbl X customer)' lineitem)' X nation

Suppose you submit the following request against the TPC-H performance benchmark database:

```
SELECT c_custkey,c_name,SUM(l_extendedprice*(1-l_discount)(FLOAT))
       (DECIMAL(18,2)) AS revenue, c_acctbal, n_name, c_address,
       c_phone,c_comment
FROM customer, ordertbl, lineitem, nation
```

```

WHERE c_custkey = o_custkey
AND   l_orderkey = o_orderkey
AND   o_orderdate >= '1993-10-01'
AND   o_orderdate < DATE '1993-10-01' + INTERVAL '3' MONTH
AND   l_returnflag = 'R'
AND   c_nationkey = n_nationkey
GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address,
         c_comment
ORDER BY revenue DESC;

```

When the definition of the nation table is changed to define the primary index on n_key rather than n_name, the join plan without Partial GROUP BY would include the last GROUP BY as follows: (((lineitem X ordertbl) X customer) X nation). Note the 'GROUP BY operator at the end of the join plan specification.

The Partial GROUP BY optimization compensates for this change in the primary index definition by removing the last GROUP BY, resulting in the plan ((lineitem X ordertbl) X customer) X nation.

Collecting Statistics to Enable Partial GROUP BY Optimization

For the Optimizer to know when to apply GROUP BY operators effectively, you should collect statistics on all join and GROUP BY columns in your typical requests.

For the following query, for example, you should collect statistics on l_orderkey, o_orderkey, o_orderdate, c_custkey, c_nationkey, and n_nationkey:

```

SELECT c_custkey, c_name,
       SUM(l_extendedprice*(1-l_discount))(FLOAT))(DECIMAL(18,2))
       AS revenue, c_acctbal, n_name,
       c_address,c_phone,c_comment
FROM customer, ordertbl, lineitem, nation
WHERE c_custkey=o_custkey
AND   l_orderkey=o_orderkey
AND   o_orderdate>='1993-10-01'
AND   o_orderdate<DATE '1993-10-01' + INTERVAL '3' MONTH
AND   l_returnflag='R'
AND   c_nationkey=n_nationkey
GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address,
         c_comment
ORDER BY revenue DESC;

```

Using another example request, you should collect statistics on l_partkey and ps_partkey for the following query:

```

SELECT l_partkey, SUM(l_quantity), SUM(ps_supplycost)
FROM partsupp, lineitem

```

```
WHERE l_partkey=ps_partkey
GROUP BY 1;
```

Join Strategies and Methods

The Optimizer has several strategies for joining tables, including those based on factors such as join geography and join order. The Optimizer also has many join methods, or modes, to choose from to ensure that any join operation is fully optimized. A cross section of the join methods available to the Teradata Optimizer is described in the sections that follow. Examples of join methods include product joins, merge joins, and hash joins.

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

Guidelines for Helping to Optimize Join Operations

The following 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 more information on the Optimizer form of the COLLECT STATISTICS statement, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- To obtain an accurate description of the processing that will be performed for a particular join, always submit an EXPLAIN request modifier 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 more information on the EXPLAIN request modifier, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

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	<ul style="list-style-type: none"> • Always selected by the Optimizer for WHERE clause inequality conditions. • High cost because of the number of comparisons required.
<ul style="list-style-type: none"> • Merge • Hash 	<ul style="list-style-type: none"> • When done on matching primary [AMP] indexes, does not require any data to be redistributed. • Hash joins are often better performers and are used whenever possible. They can be used for equijoins only.
Nested	<ul style="list-style-type: none"> • Only join expression that generally does not require all AMPs. • Preferred join expression for OLTP applications.

Strategies for Joining Skewed Tables on an Equality Join Condition

When one or both tables in a join contain skewed values, the performance of the join operation is frequently degraded. Skew on certain values refers to the variation in the number of rows that contain those values among the various AMPs. If the variation is high, meaning that some AMPs have a high number of rows with those values and some AMPs have far fewer rows with the values, the values are described as being skewed and the base table is said to be skewed on these values.

For example, consider the 2 tables *product* and *sales*, which are joined on the condition `product.product_id = sales.product_id`, and column *sales.product_id* is skewed on the value 1.

The classic join strategies pursue various geographic methods such as those listed below, where the relative sizes of the individual tables play a major role in selecting the optimal strategy. For example, if the table being duplicated in the second or third plan is large or if both the tables are large, any plan chosen from the list can degrade performance.

- Redistribute *product* on *product.product_id* and redistribute *sales* on *sales.product_id*.
- Duplicate *product* and access *sales* locally/directly.
- Access *product* locally/directly and duplicate *sales*.

A partial redistribution partial duplication (PRPD) join strategy helps to minimize the impact of skew on join performance by, for example, making the join between *sales* and *product* as two separate joins.

- For the first join, PRPD keeps the rows with the skewed value 1 of *sales.product_id* locally on each AMP and duplicates rows from *product.product_id* that match the skewed value of *sales.product_id* to all AMPs and then joins only those rows.
- For the second join, PRPD joins the non-skewed rows from the *sales* table and the rest of the rows from the *product* table using whatever join method the Optimizer determines to be best.

The database combines the results of these two joins as the final join result.

The process of dividing the rows in a single source into several subparts is referred to as a *split*. Using the PRPD join strategy, the database splits both the *sales* and *product* tables into two relations that participate in two regular joins. For the subparts of the two tables with skewed values, the Optimizer also selects the best join plan based on the costs of the different join plans, so the geographies are set according to the most optimal join plan.

The preceding case is just one possible example of PRPD when there is a single skewed value in a single column of one table. The Optimizer can also use PRPD when there are multiple skewed values and multiple join columns with skew in one or both of the tables. When the set of join conditions is on expressions of base table columns and statistics have been collected on the expressions, the Optimizer can also use PRPD to join skewed tables.

For the preceding example, the Optimizer selects a local geography for the skewed rows in *sales* and a duplication geography for the rows with skewed values in *product*. These geographies are not fixed in PRPD.

PRPD requires the same demographic support as any other join operation, such as accurate statistics, which it uses to determine the list of skewed values. PRPD uses skew detection logic to update existing statistics if the Optimizer determines that is necessary. The Optimizer selects a join plan using PRPD only if

it is cheaper than other join methods. When both relations being joined are skewed on the same value, the selection of which relation is the skewed relation depends on the number of rows with skewed values and the row size of both relations. The Optimizer selects the relation with a higher cost based on those factors as the skewed relation.

The main challenge the Optimizer faces when it determines whether to use PRPD for a join is to detect the skewed values and their frequencies because the single-table conditions and previous joins, if any, might filter out some skewed values and alter the frequency of the surviving skewed values. The Optimizer does not always have accurate skewed value information for PRPD planning.

PRPD is designed to operate in one of 2 modes, depending on an internal parameter setting.

Mode	PRPD plan is considered by the Optimizer
normal	only when the surviving skewed values and their frequencies can be determined with some confidence. This is the default setting.
aggressive	even when there is no proper information about the surviving skewed values. The skewed values are derived based on heuristics.

The following examples clarify when the Optimizer can try a PRPD plan with join operations.

Consider the following 3 tables for the example set:

- *t1*(*x1*, *y1*, *z1*), primary index defined on (*x1*)
- *t2*(*x2*, *y2*, *z2*), primary index defined on (*x2*)
- *t3*(*x3*, *y3*, *z3*), primary index defined on (*x3*)

In the following examples, when it is said that a relation qualifies for PRPD, it means that the following items are true:

- Statistics are collected on the hashed join column set.
- The demographics of the join column set satisfy a set of conditions that are determined by an internal parameter setting.

Normal PRPD mode is assumed by default. If an example applies to aggressive mode only, that is explicitly stated.

Example: PRPD for Multiple Skewed Tables

In this example, if either *t1* or *t2* qualifies for PRPD, the Optimizer tries a PRPD plan by considering *t1* or *t2* as the skewed relation. In this case, there are two partial joins.

If both tables *t1* and *t2* are skewed, the Optimizer tries a PRPD plan using 3 joins.

```
SELECT *
FROM t1, t2
WHERE t1.y1 = t2.y2;
```

Example: PRPD When Derived Statistics Can Identify Surviving Skewed Values

In this example, the derived statistics logic can find the surviving skewed values from the histogram with some confidence so *t1* qualifies for PRPD if statistics have been collected on *y1*.

```
SELECT *
FROM t1,t2
WHERE t1.y1 = t2.y2
AND t1.y1 > 5;
```

Example: PRPD for Mixed Single-Column and Multicolumn Statistics on Skewed Values

In this example, the derived statistics logic can find the range of surviving values for *t1.y1* after applying the single-table condition, so *t1* qualifies for PRPD if statistics have been collected on (*t1.y1*) and (*t1.z1*, *t1.y1*).

If multicolumn statistics on (*t1.z1*, *t1.y1*) have not been collected, the Optimizer cannot find the surviving skewed values on column *t1.y1* after it applies the single-table condition so *t1* does not qualify for PRPD in normal mode. For this case, the Optimizer considers PRPD only if the internal parameters for PRPD are set for aggressive mode.

```
SELECT *
FROM t1,t2
WHERE t1.y1 = t2.y2
AND t1.z1 > 5;
```

There are other scenarios as well, where the Optimizer can find the surviving skewed values on the join column with some confidence when there are single-table conditions. An example is when there is a sparse join index that covers the table and statistics have been collected on the join column in the join index. For this example, the join can qualify for PRPD if there is a join index with the following definition and statistics are collected on *j1.y1*, *t1* can qualify for PRPD.

```
CREATE JOIN INDEX j1 AS
  SELECT *
  FROM t1
  WHERE z1 > 5;
```

Example: PRPD for Skewed Expression Statistics

The Optimizer evaluates *t1* for PRPD only when statistics on the expression (*t1.y1* + *t1.z1*) are available. If the Optimizer determines that *t1* qualifies for PRPD based on (*t1.y1* + *t1.z1*) expression statistics, it tries a PRPD plan similar to [Example: PRPD for Multiple Skewed Tables](#).

```
SELECT *
FROM t1,t2
WHERE t1.y1 + t1.z1 = t2.y2;
```

Example: PRPD for Skewed Single Column Statistics

Suppose $t1.y1$ is skewed, statistics have been collected on $t1.y1$, and the first join (call it $R4$) is between tables $t1$ and $t3$. After the first join, the Optimizer cannot determine which skewed values from $y1$ survived the join condition $t1.x1 = t3.x3$, so for the $R4 \times R2$ join, $R4$ does not qualify for PRPD.

For this case, the Optimizer considers PRPD only if the internal parameters for PRPD are set for aggressive mode.

```
SELECT *
FROM t1,t2, t3
WHERE t1.y1 = t2.y2
AND t1.x1 = t3.x3;
```

Example: PRPD for Multicolumn Sets Skewed on Some Values

Suppose $(t1.y1, t1.z1)$ and $(t2.y2, t2.z2)$ are skewed on some values and you have collected multicolumn statistics on those column sets.

To process this request, the Optimizer normally picks an inclusion join with a pre-join sort that removes duplicates on $(t1.y1, t1.z1)$. If there is skew on $(t1.y1, t1.z1)$, the pre-join sort removes the duplicates and removes the skew as well. Because of this, the Optimizer does not evaluate a PRPD plan by considering $t1$ to be a skewed table. However, if there is skew on $(t2.y2, t2.z2)$, the Optimizer does consider a PRPD plan.

```
SELECT *
FROM t2
WHERE t2.y2 IN (SELECT t1.y1
                FROM t1
                WHERE t2.z2 = t1.z1);
```

Related Information

Product joins, see [Product Join](#).

Hash joins, see [Hash Join](#).

rowID joins, see [RowID Join](#).

Column-partitioned tables and join indexes, see the information about CREATE TABLE and CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 and the information about primary indexes and hash and join indexes in *Teradata Vantage™ - Database Design*, B035-1094.

Product Join

Definition of the Product Join

The product join compares every qualifying row from one relation to every qualifying row from the other relation and saves the rows that match the WHERE predicate filter. Because all rows of the left relation in the join must be compared with all rows of the right relation, the system always duplicates the smaller relation on all AMPs, and if the entire spool does not fit into available memory, the system is required to read the same data blocks more than one time. Reading the same data block multiple times is a costly operation.

This operation is called a product join because the number of comparisons needed is the algebraic product of the number of qualifying rows in the two relations.

Any of the following conditions can cause the Optimizer to apply 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 relation is not specified in any join condition.
- The product join is the least costly join method available in the situation.

Product Join Types

The following are the families of product join methods:

- Inner product join
- Left outer product join
- Right outer product join
- Full outer product join
- Piggybacked product join
- Inclusion product join

Used only for a product join made on an IN term.

- Inner inclusion product join
- Outer inclusion product join

- Exclusion product join

Used only for a product join made on a NOT IN term.

Exclusion product joins with dynamic column partition elimination are not supported.

- Inner exclusion product join
- Outer exclusion product join

Processing a Product Join

The following list outlines the product join process:

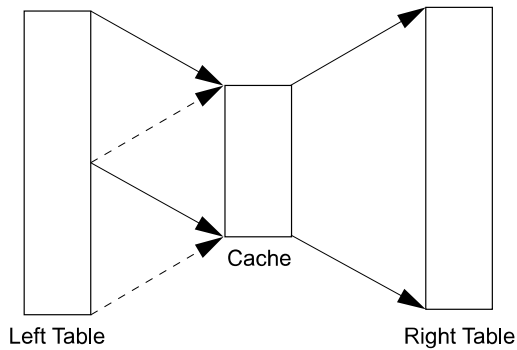
1. Cache the left relation rows.

- Join each row of the right relation with each row from the cached left relation.

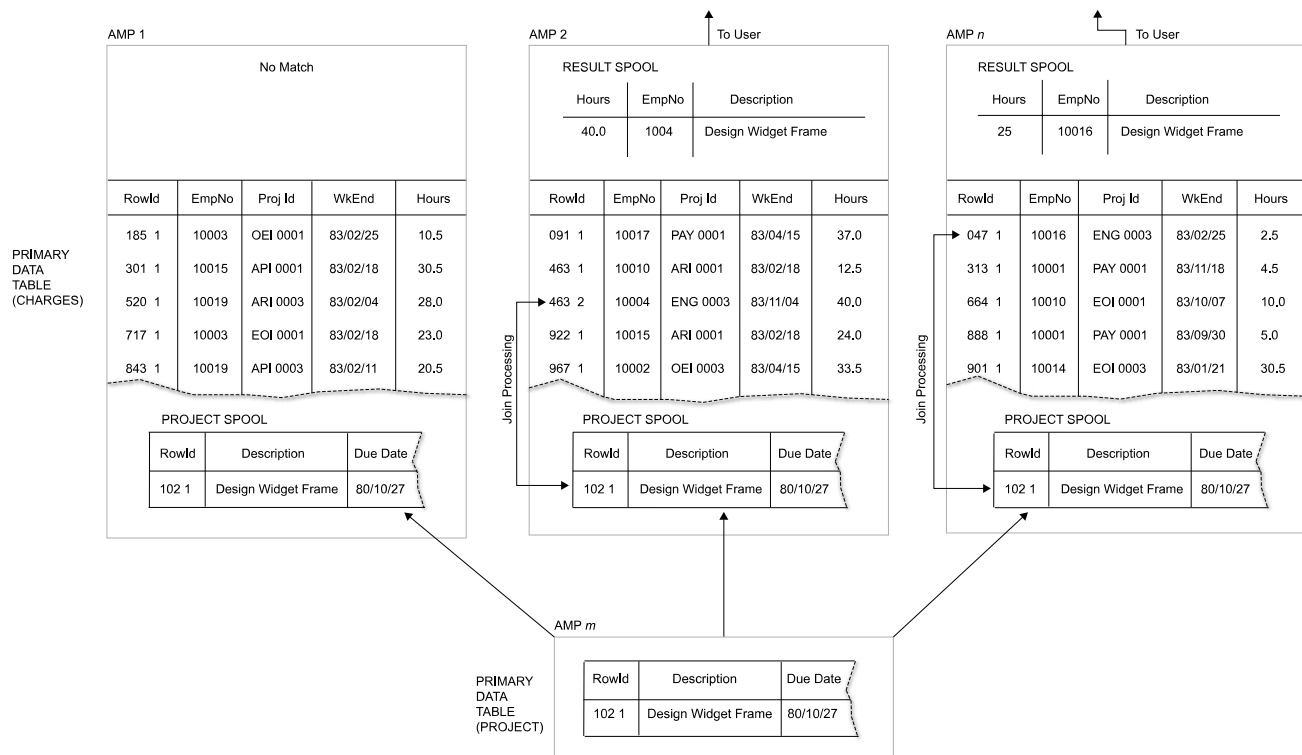
An overall join condition is a **WHERE** constraint that links relations to be joined on a column that is common to each, as shown in the following example:

```
WHERE employee.deptno=department.deptno
```

The following graphic illustrates the generic product join process.



The following graphic illustrates a product join where rows from the Project table are duplicated to all AMPs.



Product Join Costs

Product joins are relatively more time consuming than other types of joins because of the number of comparisons that must be made.

The product join is usually the most costly join method available in terms of system resources, and is used only when there is not a more efficient method, such as a merge join, hash join, or a nested join. However, a product join is useful because it can resolve any combination of join conditions.

Product Joins With Dynamic Row Partition Elimination

The database performs dynamic row partition elimination, or DPE, for product joins in the AMPs after a query has already been optimized, so in this case, the row partition elimination undertaken is dependent on the actual data on disk, not a statistical estimate of that data.

The term *direct product join* describes a join in which the table or join index of interest is not spooled in preparation for a product join, but is instead joined directly. The Optimizer might choose a direct product join when all the partitioning columns of one or more partitioning expressions are specified in equality join terms.

As is the case for static partition row elimination (see [Static Row Partition Elimination](#)), the database applies the product join DPE optimization for single-level partitioning independently for each partitioning expression.

Row partition elimination methods can be mixed within the same query. For example, static row partition elimination can be used for some partitioning levels, while DPE can be used for other levels, and some levels might not evoke any partition elimination. Some partition levels might even benefit from multiple forms of partition elimination.

The system combines the result of the partition elimination process to determine which internal partitions need to be accessed. Combining can occur for Static Partition Elimination as part of generating the query plan.

DPE occurs within the AMPs as row sets are processed, and is combined with any static row partition elimination from the query plan at that point.

When performing join steps, the qualifying row partitions of the row-partitioned table or join index are determined dynamically based on the values of rows selected from the other relation in the join. Instead of a product join against all the rows in the row-partitioned table, the database does a product join only for each set of rows of the other relation of a binary join that match with a single row partition. The database does not necessarily choose to use this optimization for this type of query because other join plans might have a better estimated cost depending on the demographics of the data and the form of partitioning that is used.

DPE for product joins can occur when there is an equality constraint between a partitioning column of one table and a column of another relation in the join. This is useful when the system uses a product join and must look up rows in one table and match those rows to rows in corresponding row partitions instead of performing a product join to the entire table. In this case, only those row partitions that are required to answer the request take part in the join.

For a product join with DPE, the left relation is sorted by RowKey using the same partitioning expression as the right relation.

Rows from the left relation are loaded into the cache one row partition at a time. This partition is then set as the partition for reading the right row-partitioned table. When the end of the partition for the right table is reached, the system reloads the left cache with the next row partition, and the process is repeated.

For example, if there are 100 row partitions in the row-partitioned table and only 5 of them are needed to answer the join request, the system does not join the other 95 row partitions, providing a 95% resource saving for the operation.

Be sure to collect statistics on the following column sets from each table being joined:

- The primary indexes
- The system-derived PARTITION columns
- The partitioning columns of the row-partitioned table
- The column in the other table being equated with the partitioning column of the row-partitioned table

Product Join With Dynamic Row Partition Elimination for Character Partitioning

For the Optimizer to apply dynamic row partition elimination for a character partitioning level with an equality join term qualifying that level, the following conditions must all be met:

- The estimated cost of the join is less than the estimated cost of all other join types.
- There must no more than one character partitioning column at the partitioning level being considered.
- If the session collation or the collation of the row-partitioned table is either MULTINATIONAL or CHARSET_COLL, and if any comparison or string function involving any non-constant expressions in the partitioning expression at the partitioning level being considered is case insensitive, the session collation must match the collation of the row-partitioned table.

The following functions and attribute are case-insensitive.

- LOWER function
- SOUNDEX function
- UPPER function
- UPPERCASE attribute

The following functions are case-sensitive.

- CHAR2HEXINT
- TRANSLATE
- TRANSLATE_CHK
- TRIM
- VARGRAPHIC

Presence of the concatenation operator (||) marks the expression as having the property of being both case sensitive and case insensitive.

The following functions follow the same rules as comparison operators, function input arguments, and the default case sensitivity for the session mode that was in effect when the CPPI was created or modified, and the system examines them all to determine case sensitivity:

- INDEX
- MINDEX
- POSITION

Specifying the SUBSTRING function does not affect case sensitivity.

- If the WHERE clause predicate that qualifies a row partitioning level for a product join with DPE is case insensitive, then all comparisons and all string functions in any non-constant expressions in the partitioning expression for that level must also be case-insensitive.

The WHERE clause predicate is considered to be case-insensitive if any of the comparisons or string functions involving non-constant expressions in the condition is case-insensitive.

For the Optimizer to specify a product join with DPE for a given row partitioning level, the collation for the current session need not match the collation for the row-partitioned table when all equality join terms on character partitioning columns are case-sensitive.

Merge Join

A merge join retrieves rows from two tables by processing equality joins that probe one of the tables using either row key or row hash values from the qualified rows in the other table being joined. A merge join assumes that the probed table is sorted either by row key or row hash. The operation then puts the joined rows onto a common AMP based on the row key or row hash of the join columns. The database sorts the rows into join column row key or row hash sequence, then joins those rows that have matching join column row key or 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](#)) because it requires fewer comparisons and because blocks from both tables are read only once.

Merge Join Methods

Each of these general merge join algorithms, described in more detail in the sections that follow, can also be applied to the various merge join methods.

- Fast path inner merge join
Fast path inner inclusion merge join
- Slow path inner merge join
Slow path inner inclusion merge join
- Exclusion merge join
- Fast path left outer merge join
Fast path left outer inclusion merge join
- Slow path left outer join
Slow path left outer inclusion merge join

- Fast path right outer merge join
- Slow path right outer merge join
- Full outer merge join

Another class of merge join methods is used only with row-partitioned tables.

- Direct merge join
- Rowkey-based merge join
- Single window merge join
- Sliding window merge join

Dynamic row partition elimination is not supported for merge joins of multilevel partitioned tables. For more information, see [Product Joins With Dynamic Row Partition Elimination](#) and *Teradata Vantage™ - Database Design*, B035-1094.

Approaches to Merge Joining Row-partitioned PI Table to Unmatched Relation

The Optimizer has the following general approaches when joining a row-partitioned PI table to a nonpartitioned table or when joining two row-partitioned PI tables with different partitioning expressions:

- Spool the row-partitioned PI table (or both row-partitioned PI tables) into a nonpartitioned spool in preparation for a traditional merge join.
- Spool the nonpartitioned table (or one of the two row-partitioned PI tables) into a row-partitioned PI spool, with identical partitioning to the remaining table, in preparation for a rowkey-based merge join (see [Rowkey-Based Merge Join](#)).

This option is not always available.

- Use the sliding window merge join of the tables without spooling either one (see [Sliding-Window Merge Join](#)).

In this case, both tables must have the same primary index and the join must be an equality join between the primary index columns and neither can be column-partitioned.

In all cases, the Optimizer considers all reasonable join strategies and selects the one with the least estimated cost.

Generic Merge Join Strategy

The following is the high level process applied by the merge join algorithm:

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.
 - a. Place the qualifying rows from one or both relations into separate spools.
As the qualified rows are placed into spool, relocate them to their target AMPs based on the rowkey or hash of the join column set.
 - b. Sort the qualified spool rows on their join column rowkey or row hash values.
3. Compare the relocated row set with matching join column row hash values in the other relation.

Dynamic Row Partition Elimination for Merge Join with Equality Join Terms

When there are equality join terms on all primary index columns of a single-level row-partitioned PI table plus certain forms of range join terms on the partitioning column, then the Optimizer might use a merge join with dynamic row partition elimination to make the join.

You can think of this as a form of sliding window merge join (see [Sliding-Window Merge Join](#)) with the row-partitioned PI table in which the other table being joined is partitioned the same way as the row-partitioned PI table, and a subset of the partitions in one table is joined to a subset of the row partitions in the other table as determined by the specified range terms.

Dynamic row partition elimination-applicable join terms must be specified in one the following forms:

```
{ ppi_t1.partitioning_column_1 + C1 operator ppi_t2.partitioning_column_2 |
  ppi_t2.partitioning_column_2 operator ppi_t1.partitioning_column_1 + C2 |
  ppi_t1.partitioning_column_1 + C1 operator ppi_t2.partitioning_column_2 |
  ppi_t2.non_partitioning_column_2 operator ppi_t1.ppi_column_1 + C2
}
```

ppi_t1.partitioning_column_1

A partitioning column for table *ppi_t1*.

C1

C2

A constant.

operator

One of the following join operators:

- <
- <=
- =>
- >

ppi_t2.partitioning_column_2

A partitioning column for table *ppi_t2*.

ppi_t2.non_partitioning_column_2

A nonpartitioning column for table *ppi_t2*.

ppi_t1.ppi_column_1

A column for table *ppi_t1*.

Merge Join with Dynamic Row Partition Elimination Not Supported for Character-Partitioned Tables

Because you cannot arithmetically add strings together, and because string intervals are not defined, merge join dynamic row partition elimination is not supported for character-partitioned tables.

Slow Path Inner Merge Join

The process applied by the slow path merge join algorithm is as follows:

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

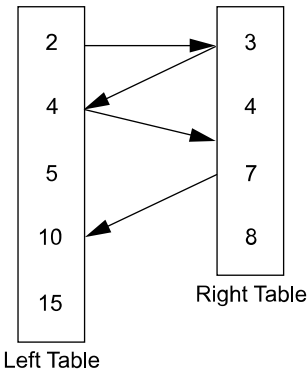
Fast Path Inner Merge Join

The process applied by the fast path merge join algorithm is as follows:

- 1. Read a row from the left table and record its hash value.
- 2. Read the next row from the right table that has a row hash \geq to that of the left table row.

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

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



Merge Join Distribution Strategies

Merge joins use one of the following distribution strategies:

Strategy Number	Strategy Name	Stage	Process
1	Hash Redistribution	1	Hash redistribute one or both sides to few or all AMPs (depending on the primary indexes used in the join).

Strategy Number	Strategy Name	Stage	Process
			If a table has a Primary AMP on the join columns, the database can spool locally instead of using a hash redistribution, but stage 2 for the table is still needed.
		2	Sort the rows into join column row hash sequence.
2	Duplication	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	Index Matching	1	No redistribution is required if the primary indexes are the join columns and if they match.

Merge Join Examples

The following SELECT request determines who works in what location by merge joining the *employee* and *department* tables on the condition that the *dept_no* values in both tables are equal:

```
SELECT name, dept_name, loc
FROM employee INNER JOIN department
WHERE employee.dept_no = department.dept_no;
```

The following list presents the stages of processing this merge join.

1. Because department rows are distributed according to the hash code for *dept_no* (the unique primary index of the department table), employee rows are themselves redistributed by the hash code of their own *dept_no* values.
2. The redistributed employee rows are stored in a spool on each AMP and sorted by the hash value for *dept_no*.

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 *dept_no* 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.

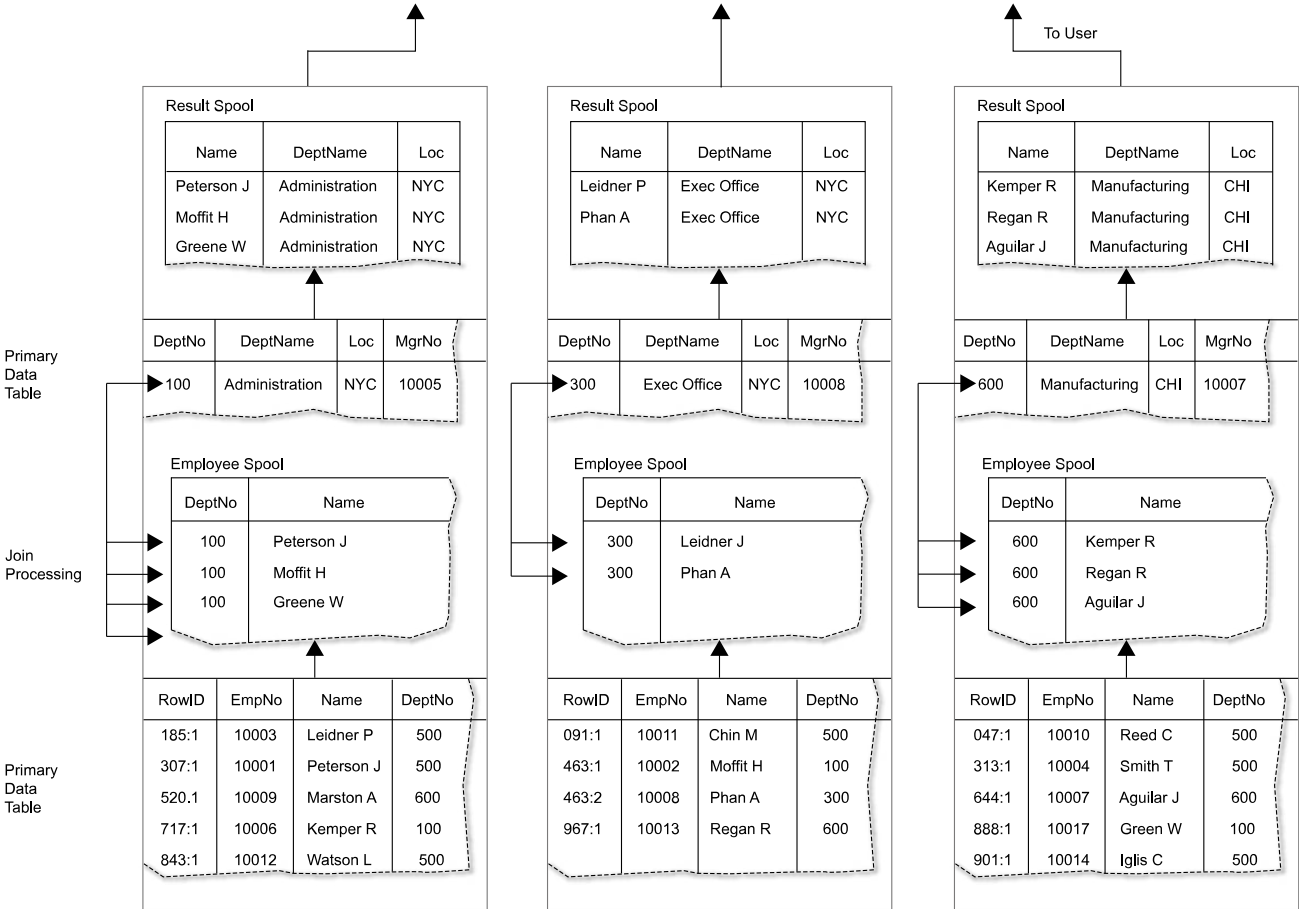
The hash code and value comparisons continue until the end of one of the tables is reached.

6. On each AMP the name, *dept_no*, and loc values from each of the qualifying rows are placed in a result spool.

7. When the last AMP has completed its merge join, the contents of all result spools are merged and returned to the user.

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

The following graphic illustrates this merge join process:



The next merge join example uses the following table definitions:

Employee Table

e_num	e_name	dept
1	Brown	200
2	Smith	310
3	Jones	310
4	Clay	400
5	Peters	150
6	Foster	400

e_num	e_name	dept
7	Gray	310
8	Baker	310

Column e_num is the UPI and PK of the table, and dept is a FK.

Department Table

dept	dept_name
400	Education
150	Payroll
200	Finance
310	Mfg

Column dept is the UPI and PK of the table.

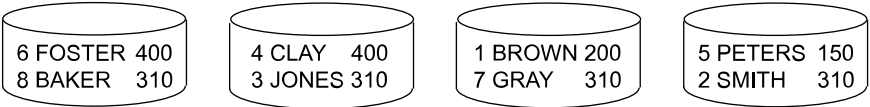
One of the merge join row redistribution methods is used if you perform the following SELECT request against these tables:

```
SELECT *
FROM employee INNER JOIN department
WHERE employee.dept = department.dept;
```

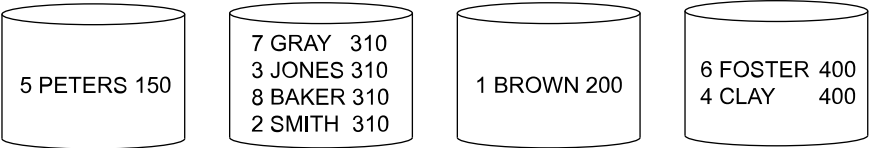
Merge Join Row Distribution (Hash Redistribution)

The following graphic shows a merge join row distribution using hash redistribution:

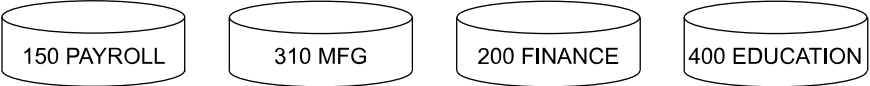
EMPLOYEE ROWS HASH-DISTRIBUTED ON EMPLOYEE.ENUM (UPI):



SPOOL FILE AFTER REDISTRIBUTION ON EMPLOYEE.DEPT ROW HASH:



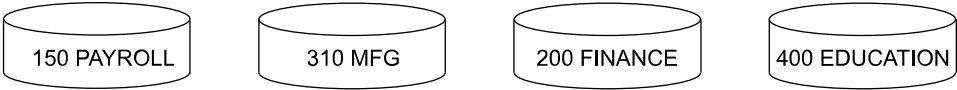
DEPARTMENT ROWS HASH-DISTRIBUTED ON DEPARTMENT.DEPT (UPI):



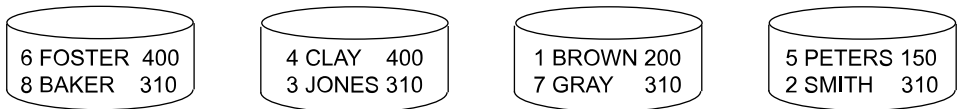
Merge Join Row Distribution (Duplicate Table)

The following graphic shows a merge join row distribution by duplicating a table:

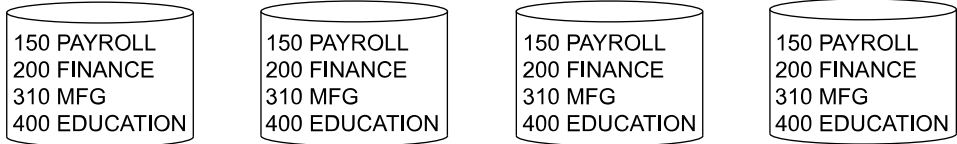
DEPARTMENT ROWS HASH-DISTRIBUTED ON DEPARTMENT.DEPT (UPI):



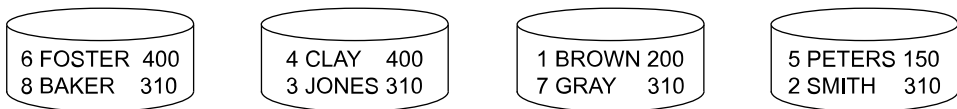
EMPLOYEE ROWS HASH-DISTRIBUTED ON EMPLOYEE.ENUM (UPI):



SPOOL FILE AFTER DUPLICATING AND SORTING ON DEPARTMENT.DEPT ROW HASH:



SPOOL FILE AFTER LOCALLY COPYING AND SORTING ON EMPLOYEE.DEPT ROW HASH:



This example uses the following employee and employee_phone table definitions:

Employee Table

e_num	e_name
1	Brown
2	Smith
3	Jones
4	Clay
5	Peters
6	Foster
7	Gray
8	Baker

Column e_num is the UPI and PK of the table.

Employee_phone Table

e_num	area_code	phone
1	213	4950703
1	408	3628822
3	415	6347180
4	312	7463513
5	203	8337461
6	301	6675885
8	301	2641616
8	213	4950703

Column e_num is the NUPI and a FK of the table, and all three columns constitute the PK.

The matching indexes row distribution strategy is used if you perform the following SELECT request against these tables.

```
SELECT *
FROM employee INNER JOIN employee_phone
WHERE employee.e_num = employee_phone.e_num;
```

Direct Row-partitioned PI Merge Join

About the Direct Row-partitioned PI Merge Join

The term *direct merge join* describes a join method in which the table or join index of interest is not spooled in preparation for a merge join, but instead is done directly. The Optimizer might choose a direct merge join when at minimum all columns of the primary index are specified in equality join terms.

To qualify for a direct row-partitioned PI merge join, there must be equality conditions on all the primary index columns of the two relations. This applies equally to character and non-character row-partitioned PIs. There are several forms of this optimization. The particular form selected by the Optimizer depends on factors such as the following:

- Any additional conditions in the query
- The total number of row partitions
- The number of populated row partitions

In the following example, the Optimizer can choose to do a direct merge join of markets and market_penetration instead of redistributing both tables to spool, sorting the spool in hash order of the primary index, and then doing a row hash merge join.

The example uses the following table definitions:

```

CREATE TABLE markets (
  productid      INTEGER NOT NULL,
  region         BYTEINT NOT NULL,
  activity_date  DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  revenue_code   BYTEINT NOT NULL,
  business_sector BYTEINT NOT NULL,
  note           VARCHAR(256))
PRIMARY INDEX (productid, region)
PARTITION BY (
  RANGE_N(region      BETWEEN 1
            AND        9
            EACH       3),
  RANGE_N(business_sector BETWEEN 0
            AND        49
            EACH       10),
  RANGE_N(revenue_code BETWEEN 1
            AND        34
            EACH       2),
  RANGE_N(activity_date BETWEEN DATE '1986-01-01'
            AND        DATE '2007-05-31'
            EACH INTERVAL '1' MONTH));

CREATE TABLE market_penetration (
  productid      INTEGER NOT NULL,
  region         BYTEINT NOT NULL,
  activity_date  DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  revenue_code   BYTEINT NOT NULL,
  business_sector BYTEINT NOT NULL,
  saturation     FLOAT)
PRIMARY INDEX (productid, region)
PARTITION BY (
  RANGE_N(region      BETWEEN 1
            AND        9
            EACH       3),
  RANGE_N(business_sector BETWEEN 0
            AND        49
            EACH       10),
  RANGE_N(revenue_code BETWEEN 1
            AND        34
            EACH       2),
  RANGE_N(activity_date BETWEEN DATE '1986-01-01'
            AND        DATE '2007-05-31'
            EACH INTERVAL '1' MONTH));

```

The example request joins `markets` and `market_penetration`. Because of the specified conditions, the Optimizer is able to select a direct row-partitioned PI-to-row-partitioned PI merge join to join the relations.

```
SELECT a.*, b.saturation
FROM   markets AS a INNER JOIN market_penetration AS b
WHERE  a.productid      = b.productid
AND    a.region         = b.region
AND    a.business_sector = b.business_sector
AND    a.revenue_code   = b.revenue_code
AND    a.activity_code  = b.activity_code;
```

Rowkey-Based Merge Join

A rowkey-based merge join for single-level and multilevel partitioning requires equality conditions on all the primary index columns and partitioning columns of the two relations.

To be eligible for a rowkey-based merge join, both relations must also have the same partitioning. Otherwise, one of the relations must be spooled and partitioned to impose equal partitioning between the two.

Vantage does not support full outer rowkey-based merge joins with partition remapping for row-partitioned PI tables having 8-byte partitioning.

Also see *Necessary Conditions for a Rowkey-Based Merge Join With a Character-Partitioned Relation* below for additional restrictions that apply for character PPIs.

Supported Join Methods for Rowkey-Based Merge Join

The following join types are supported for a rowkey-based merge join, where *fast path* designates row hash match scan within left and right matching partitions and *slow path* designates index access on one relation followed by a lookup in the other table for all rows with matching row hashes within the corresponding partition:

- Fast path inner, left and full outer merge join.
- Slow path inner and left outer merge join.
- Fast path inner and outer inclusion/exclusion merge join.
- Slow path inner and outer inclusion/exclusion merge join.
- Fast path correlated inclusion/exclusion merge join.
- Slow path correlated inclusion/exclusion merge join.

Note that this type of join can only occur between a table and a spool, and not directly between two tables.

Right outer merge join is not supported for a rowkey-based merge join. Note that the Optimizer can switch the relations so that the join type is a left outer merge join.

If the internal partition mapping is not the same for both relations, a rowkey-based merge join is not eligible for synchronized scanning. If the internal partition mapping is the same for both relations, a rowkey-based merge join is eligible for synchronized scanning only for the following join types:

- Slow path left outer merge join.
- Slow path inner merge join.

A rowkey-based merge join can be considered for a join between a primary-indexed table and another table whose primary index is not a join condition. To make such a join, the system builds the rowkey for the other table in spool based on the partitioning expressions of the row-partitioned PI table on the other side of the join.

Unsupported Rowkey-Based Merge Join Partition Elimination Feature

Vantage does not support dynamic row partition elimination for full outer rowkey-based merge joins with partition remapping for 8-byte partitioning.

Making a Row-partitioned PI-to-Spool Join

In this example, the database can select rows from the dimension tables and join the intermediate results set to form a spool that is sorted by a rowkey matching that of the row-partitioned PI table. Then the system can do a rowkey-based merge join from the PPI table to the spool.

This method replaces having to redistribute the spool and then sorting it on the join columns, with the other tables joined and spool also sorted on the join columns, and then joining the two spools using a rowhash merge join or some other join plan. Ultimately, the Optimizer selects the plan that it estimates to be the least costly of its available options.

Definition DDL request text for the markets table.

```
CREATE TABLE markets (
  productid      INTEGER NOT NULL,
  region         BYTEINT NOT NULL,
  activity_date  DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  revenue_code   BYTEINT NOT NULL,
  business_sector BYTEINT NOT NULL,
  note           VARCHAR(256))
PRIMARY INDEX (productid, region)
PARTITION BY (
  RANGE_N(region      BETWEEN 1
              AND      9
              EACH      3),
  RANGE_N(business_sector BETWEEN 0
              AND      49
              EACH      10),
  RANGE_N(revenue_code BETWEEN 1
              AND      34
              EACH      2),
  RANGE_N(activity_date BETWEEN DATE '1986-01-01'
              AND      DATE '2007-05-31'
              EACH INTERVAL '1' MONTH));
```


Definition DDL request text for the products table is as follows:

```
CREATE TABLE products (
  productid    INTEGER NOT NULL,
  product_name CHARACTER(30),
  description   VARCHAR(256))
PRIMARY INDEX (productid);
```

Definition DDL request text for the regions table is as follows:

```
CREATE TABLE regions (
  region       INTEGER NOT NULL,
  region_name  CHARACTER(30),
  description  VARCHAR(256))
PRIMARY INDEX (region_name);
```

Definition DDL statement text for the business_sectors table is as follows:

```
CREATE TABLE business_sectors (
  productid          INTEGER NOT NULL,
  business_sector_name CHARACTER(30),
  description         VARCHAR(256))
PRIMARY INDEX (business_sector_name);
```

Definition DDL statement text for the revenue_codes table is as follows:

```
CREATE TABLE revenue_codes (
  revenue_code      INTEGER NOT NULL,
  revenue_code_name CHARACTER(30),
  description       VARCHAR(256))
PRIMARY INDEX (revenue_code_name);
```

Definition DDL statement text for the activity_calendar table is as follows:

```
CREATE TABLE activity_calendar (
  quarter        CHARACTER(6),
  activity_date  DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  description    VARCHAR(256))
PRIMARY INDEX (quarter);
```

Example query that joins all these tables is as follows:

```

SELECT p.product_name, r.region_name, b.business_sector_name,
       rc.revenue_code_name, a.quarter, m.activity_date, m.note
FROM   markets AS m, products AS p, regions AS r,
       business_sectors AS b, revenue_codes AS rc,
       activity_calendar AS a
WHERE  p.productid BETWEEN 4000
        AND 4999
       AND r.region_name = 'West'
       AND b.business_sector_name IN ('Cosmetics','Snack Food','Hats')
       AND rc.revenue_code_name IN ('Web', 'Catalog')
       AND a.quarter IN ('2006Q1', '2005Q1')
       AND m.productid      = p.productid
       AND m.region         = r.region
       AND m.business_sector = b.business_sector
       AND m.revenue_code   = rc.revenue_code
       AND m.activity_code  = a.activity_code;

```

Necessary Conditions for a Rowkey-Based Merge Join With a Character-Partitioned Relation

The following rules apply to *direct* rowkey-based merge joins for character-partitioned relations. In each case, the attributes must be the same for both relations in the join for it to be eligible for the rowkey-based merge join method.

Note:

A *Direct Join* is a binary join operation for which the relation of interest is not spooled in preparation for the join. For example, a direct merge join is a join in which the relation of interest is not spooled in preparation for a merge join. Similarly, a direct product join is a product join in which the relation of interest is not spooled in preparation for a product join.

- The relations must have the same collation.
- The relations must have the same server character set for their character partitioning columns.
- The character partitioning columns of the relations must have the same data type, including column length and case specific attribute.

Unless all of these conditions are true for both of the relations to be joined, one of them must be spooled and then repartitioned to match the partitioning of the other partitioned relation in the join.

Whether the Optimizer selects a direct rowkey-based merge join, or whether one relation is spooled and then partitioned to match the partitioning of the character-partitioned relation, the following rules must be met for the Optimizer to specify a rowkey-based merge join:

- The estimated cost of the join must be less than the estimated cost of all other join types.
- There must be no more than one character partitioning column at each partitioning level.

- If the session collation or table collation is MULTINATIONAL or CHARSET_COLL, and if any comparison or string function involving any non-constant expression in the partitioning expression at any partitioning level is case-insensitive, the session collation must match the table collation.

The following functions and attribute are case-insensitive:

- LOWER function
- SOUNDEX function
- UPPER function
- UPPERCASE attribute

The following functions are case-sensitive:

- CHAR2HEXINT
- TRANSLATE
- TRANSLATE_CHK
- TRIM
- VARGRAPHIC

Presence of the concatenation operator (||) marks the expression as having the property of being both case sensitive and case-insensitive.

The following functions follow the same rules as comparison operators, function input arguments, and the default case sensitivity for the session mode that was in effect when the character partitioning was created or modified, and the system examines them all to determine case sensitivity.

- INDEX
- MINDEX
- POSITION

Specifying the SUBSTRING function does not affect case sensitivity.

- If the WHERE clause predicate that qualifies a partitioning level for a rowkey-based merge join is case insensitive, then all comparisons and all string functions in any non-constant expressions in the partitioning expression for that level must also be case-insensitive.

The WHERE clause predicate is considered to be case insensitive if any of the comparisons or string functions involving non-constant expressions in the condition is case-insensitive.

- If the character set of the character partitioning column in an equality join condition is different than character set of the expression involving that column in the other relation in the join condition, then you must store the partitioning column data using the Unicode server character set.

This is because the system handles comparisons between non-constant character expressions by implicitly converting both expressions to Unicode before making the comparison.

For the Optimizer to specify a rowkey-based merge join, the collation for the current session need not match the collation for the table when all equality join terms on character partitioning columns are case sensitive.

If you create a character-partitioned table in an ANSI mode session, the system implicitly casts the partitioning columns compared in CASE_N-based partitioning expressions or generalized partitioning

expressions to CASESPECIFIC unless the constant expression in the comparison is explicitly CAST as NOT CASESPECIFIC.

This has an effect on the types of join terms that can be evaluated using rowkey-based merge join with such a partitioned primary index (see the example of a case where a rowkey-based merge join cannot be used later in this topic).

In this example, 2 character-partitioned tables are joined on their primary indexes and partitioning expressions. Assume the following table definitions, both of which were created under the same session collation:

```
CREATE SET TABLE test1, NO FALLBACK, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL,CHECKSUM = DEFAULT (
    i INTEGER,
    j CHARACTER(4) CHARACTER SET UNICODE CASESPECIFIC)
PRIMARY INDEX (i)
PARTITION BY RANGE_N(j BETWEEN 'AAAA','ZZZZ','aaaa','yyyy'
                      AND   'zzzz');

CREATE SET TABLE test11, NO FALLBACK, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL, CHECKSUM = DEFAULT (
    i INTEGER,
    j CHARACTER(4) CHARACTER SET UNICODE CASESPECIFIC)
PRIMARY INDEX (i)
PARTITION BY RANGE_N(j BETWEEN 'AAAA','ZZZZ','aaaa','yyyy'
                      AND   'zzzz');
```

Assume you submit the following SELECT request that joins these tables:

```
SELECT *
FROM test1 INNER JOIN test11
WHERE test1.i = test11.i
AND    test1.j = test11.j;
```

The database can join these tables using a direct rowkey-based merge join instead of having to redistribute them both to spool, sorting the spool by the hash of the primary index, and then making a row hash merge join because the relations being joined conform to the following requirements for a direct rowkey-based merge join.

- The relations have the same session collation for their character partitioning columns.
- The relations have the same server character set for their character partitioning columns.

In this case, the matching server character set is Unicode for character partitioning column *j* in both relations.

- The relations have the same data type, column length, and case specific attribute for character partitioning column *j*.

In this case, the data types (CHARACTER), their lengths (4 characters), and their case specificity (CASESPECIFIC) are the same for character partitioning column j in both relations.

In this example, 2 tables, only one of which has character partitioning, are joined on their primary indexes. Assume the following table definitions, both of which were created under the same session collation:

[illegible]

Note that markets is a character-partitioned relation, while products is not.

Assume you submit the following SELECT request that joins these tables:

```
SELECT *
FROM products INNER JOIN markets
WHERE products.product_name = markets.product_name;
```

The database cannot join these tables using a direct merge join, so it must instead redistribute the products table to spool, sort the spool by the hash of the primary index, and then make a rowkey-based merge join because the relations being joined do not conform to the requirements for a direct rowkey-based merge join in the following ways:

- Only one of the relations in the join, markets, has partitioning.
- The relations do not have the same case-specific attribute for the join column product_name.

In this case, the case specificity is CASESPECIFIC for column product_name in relation markets, but NOT CASESPECIFIC for character partitioning column product_name in relation products.

Instead of using a rowkey-based merge join, the system selects rows from the products table to form a spool that it then sorts by a rowkey that is the same as that for the character-partitioned relation markets, and it then makes a rowkey-based merge join from the character-partitioned relation markets to the spooled relation products.

The session collation need not match the character-partitioned table collation because all comparisons in the partitioned table are case specific, but if the table collation definition changes after the table was created (this is applicable to the MULTINATIONAL and CHARSET_COLL collations only), then the Optimizer does not specify a rowkey-based merge join.

The following CREATE TABLE requests are submitted in ANSI session mode:

```
CREATE MULTISET TABLE tt1(
  a INTEGER,
  b CHARACTER(30) NOT CASESPECIFIC)
PRIMARY INDEX (a)
PARTITION BY CASE_N(b BETWEEN 'A' AND 'Z',
                    b BETWEEN 'a' AND 'z',
                    NO CASE OR UNKNOWN);
CREATE MULTISET TABLE tt2(
  a INTEGER,
  b CHARACTER(30) NOT CASESPECIFIC)
PRIMARY INDEX (a);
```

The following SELECT request joining these tables is submitted in Teradata session mode:

```
SELECT *
FROM tt1 INNER JOIN tt2
WHERE tt1.a = tt2.a
AND    tt1.b = tt2.b;
```

The system cannot use a rowkey merge join for this request because the comparisons in the partitioning expression are case-sensitive, but the join conditions in the SELECT request are case-insensitive.

The following CREATE TABLE requests are submitted in Teradata session mode:

```
CREATE MULTISET TABLE tt3(
  a INTEGER,
  b CHARACTER(30) NOT CASESPECIFIC)
PRIMARY INDEX (a)
PARTITION BY CASE_N(TRIM(LEADING 'a' FROM b) BETWEEN 'A'
                    AND      'Z',
                    TRIM(LEADING 'a' FROM b) BETWEEN 'a'
                    AND      'z',
                    NO CASE OR UNKNOWN);
```

```
CREATE MULTISET TABLE tt2(
  a INTEGER,
  b CHARACTER(30) NOT CASESPECIFIC)
PRIMARY INDEX (a);
```

The following SELECT request joining these tables is submitted in Teradata session mode:

```
SELECT *
FROM tt3 INNER JOIN tt2
WHERE tt3.a = tt2.a
AND    tt3.b = tt2.b;
```

The Optimizer cannot select a rowkey merge join for this case because the comparisons in the partitioning expression for tt3 are case-sensitive because of the TRIM function, but the join conditions in the SELECT request are case insensitive.

Single-Window Merge Join

About the Single-Window Merge Join

A single-window merge join is very similar to a sliding-window merge join (see [Sliding-Window Merge Join](#)) except that the number of populated partitions after static or delayed row partition elimination has been applied for each table in the join is small enough to be able to handle all the partitions at once. This could be either a join of a PPI table to a PPI table, a PPI table to a nonpartitioned primary-indexed table, a nonpartitioned primary-indexed table to a PPI table, a PPI table to a spool, or a spool to a PPI table.

The two relations must be joined with equality conditions on the primary index. There must be a limited number of participating row partitions. The number of participating row partitions is estimated based on the estimated number of populated row partitions after any static partition elimination. The maximum number of combined partitions processed in as a single set is determined in the same way as for single-level row partitioning.

This calculation is based on the setting of the DBS Control field PPICacheThrP.

The Optimizer estimates whether the set will be small enough; if the estimate differs from the actual data, a sliding-window merge join might or might not be used. For example, the estimate might indicate a single-window merge join is the best plan but a sliding-window merge join may be done if many populated row partitions exist after static or delayed row partition elimination has been applied. The Optimizer might also estimate that the cost of a single-window merge join would exceed that of some other join method, and so would choose to use that method instead.

A single-window merge join must first spool and then sort the column-partitioned relation when one of the relations is column-partitioned.

Single-window merge joins can be used for both non-character-partitioned tables and for character-partitioned tables.

In the following example, the system joins 2 tables on their primary indexes. The WHERE clause conditions eliminate all but two partitions for level 1, and 3 partitions for level 2 of orders. The WHERE clause conditions also eliminate all but 1 partition for level 1, and 7 partitions for level 2 of lineitem. After the system applies these conditions, it must join 6 partitions of the combined expression for orders to 7 partitions of the combined partitioning expression of lineitem, making a total of 13 partitions.

Because of this situation, the system is able to join the 2 tables using a direct merge join, assuming the Optimizer estimates it to be the most cost effective join method. When joining the two sets of partitions, the direct merge join operation handles the row sets logically as if they are in hash order by maintaining a memory-resident data block for each populated partition.

The definition DDL statement text for the 2 tables to be joined in this example query is as follows:

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (
  RANGE_N(o_custkey   BETWEEN 0
              AND 49999
              EACH 100),
  RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
              AND DATE '2006-12-31'
              EACH INTERVAL '1' MONTH))
UNIQUE INDEX (o_orderkey);
CREATE TABLE lineitem (
  l_orderkey      INTEGER NOT NULL,
  l_partkey       INTEGER NOT NULL,
  l_suppkey       INTEGER,
  l_linenumbers   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_supkey   BETWEEN 0
           AND 4999
           EACH 10),
  RANGE_N(l_shipdate BETWEEN DATE '2000-01-01'
           AND DATE '2006-12-31'
           EACH INTERVAL '1' MONTH));

```

The Optimizer applies a single-window merge join to join the orders and lineitem tables when creating a join plan for the following query:

```

SELECT *
FROM orders INNER JOIN lineitem
WHERE o_orderkey = l_orderkey
      AND o_orderdate BETWEEN DATE '2005-04-01'
                      AND DATE '2005-06-30'
      AND o_custkey  IN (618, 973)
      AND l_shipdate BETWEEN DATE '2005-04-01'
                      AND DATE '2005-10-31'
      AND l_supkey = 4131;

```

Sliding-Window Merge Join

About the Sliding-Window Merge Join

A direct merge join cannot be used when one table is partitioned and the other is not, or when both tables are partitioned, but not partitioned identically, because the rows of the two tables are not ordered in the same way. The sliding-window merge join, which is PPI-aware, can be applied by the Optimizer to cover situations that the standard direct merge join cannot. Sliding-window joins can be slower than a merge join or rowkey-based merge join when there are many noneliminated nonempty combined partitions; however, a sliding-window merge join can provide roughly similar elapsed time performance (but with greater CPU utilization and memory consumption) when the number of noneliminated nonempty combined partitions is small.

Sliding-window merge joins follow the general principle of structuring each of the left and right relations into windows of appropriate sizes, appropriate in terms of the number of partitions required as determined by the Optimizer based on the available memory for making the join. The join is done as a product join between each of these left and right window pairs. The operation uses the identical algorithm to that used for a regular

merge join within each window pair with the exception that the rows are not necessarily in row hash order across the multiple partitions within a window.

The obvious way to join a nonpartitioned table to a row-partitioned PI table would be to make one pass over the nonpartitioned table for each noneliminated nonempty partition of the row-partitioned PI table, executing the join as a series of subjoins. It turns out that this is an inefficient way to make the join, especially for a large nonpartitioned table. To escape the inefficiency of this method, the sliding-window merge join uses a similar concept, but minimizes the number of disk reads.

1. The system reads the first data block for the nonpartitioned table, and then reads the first data block of each noneliminated nonempty combined partition of the row-partitioned PI table into memory.
2. The rows from the nonpartitioned data block are compared to the rows of each row-partitioned PI data block.

The join routines present the rows in row hash order, but you can think of the process as visiting the data block for each combined partition in turn.

3. As the rows of a data block are exhausted, the system reads the next data block for that combined partition.

This results in each data block of each table being read only once. This is because of partitioning. Merge joins usually have to read some number of rows in one of the tables multiple times, either from disk or from cache. There is some additional overhead to manage the pool of data blocks, but join performance is not badly degraded when the window covers all noneliminated nonempty combined partitions.

If a non-trivial fraction of the combined partitions can be eliminated because of query conditions or because they are empty, overall performance can be improved, perhaps dramatically, over a traditional merge join, depending on the percentage of combined partitions that can be eliminated or are empty.

A limiting factor for the sliding-window merge join algorithm is the number of data blocks that can be contained in memory at the same time. The file system cache memory provides memory for the data blocks.

The DBS Control performance field PPICacheThrP controls memory usage for this purpose. The value is a number expressed in tenths of a percent of the file system cache available for each query step that needs a sliding window.

The default value for PPICacheThrP is 10, which represents 1%.

A significant degradation of join performance can occur when there are more noneliminated nonempty combined partitions than data block buffers. Assume enough memory has been allocated for 20 data blocks and a table with 100 partitions. In this case, the sliding window method is appropriate. The first 20 combined partitions are processed as they should be, and then the system reads the nonpartitioned table again as the join window slides down to combined partitions 21 through 40. A total of five passes through the nonpartitioned table are required, and, assuming the nonpartitioned table is roughly the same size as the row-partitioned PI table, the join can conceptually take five times longer than a join for which the window covers the entire table. The actual performance degradation is not as bad as a factor of five, because the output spool has exactly the same number of rows in either case, and each smaller window is more sparse with respect to the nonpartitioned table than a larger window would be. There can be an offsetting performance gain from cache usage by the nonpartitioned table, especially when it is fairly small.

An even more expensive situation occurs in the following cases:

- Both tables are partitioned, but have different partitioning expressions
- There are no join conditions specified on the partitioning columns

In both cases, there can be a sliding-window advancing through both tables. The EXPLAIN text for such a query neither indicates that a sliding-window merge join is being used, nor indicates the number of contexts used for each table.

Equation element	Number Specified
LDR_{ntp}	logical data reads when neither table is partitioned.
LDR_{t2p}	logical data reads when the second table is partitioned.
LDR_{btp}	logical data reads when both tables are partitioned.
d_1	data blocks for the first table.
d_2	data blocks for the second table.
p_1	uneliminated combined partitions in the first table.
p_2	uneliminated combined partitions in the second table.
k_1	data blocks that can be contained in memory for the first table.
k_2	data blocks that can be contained in memory for the second table.

This allocation, in units of data blocks ...	Is this value ...
minimum	8 or less, to match the number of noneliminated combined partitions, even if PPICacheThrP is set to 0.
maximum	the smallest of the following: <ul style="list-style-type: none"> • 256 • the largest number which does not exceed the percentage specified in the PPICacheThrP setting • the number of noneliminated combined partitions The maximum allocation can never be less than the minimum allocation.

A larger number in PPICacheThrP allocates relatively more memory, when needed, to partition windowing steps, at the expense of other steps that might use cache for other purposes.

The default PPICacheThrP setting is low enough that steps unrelated to partition windowing are not likely to be short on cache. Assuming there is never more than a maximum of 50 concurrent sliding window joins on a given AMP, each operating on tables with many combined partitions, which could consume up to about 50% of the cache memory allocated for windowing. Even in that worst case scenario, an AMP would have the remaining half of the cache for other data blocks.

The sliding windows are based on the row partitions defined by the combined partitioning expression. For a sliding-window merge join to be cost effective, only a limited number of populated combined partitions can participate in the join; however, multilevel row partitioning usually defines many combined partitions for the combined partitioning expressions. This means using a sliding-window merge join in a multilevel row partitioning situation may be too costly.

The final cost of a sliding-window merge 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 combined partitions in the PPI relation set and the window size.

The Optimizer always estimates the cost of the join plans it evaluates to process a query; however, if an estimate differs from the actual data on the AMPs as it is revealed during processing for single-window join, the system might then choose to substitute a sliding-window merge join. Alternatively, the query plan developed by the Optimizer might specify a sliding-window merge join, but in the final result, the system might instead dynamically reoptimize the request and use a single-window merge join if there are few populated row partitions (see [Product Joins With Dynamic Row Partition Elimination](#)).

In cases where there are conditions on a partitioning column that permit row partition elimination, the Optimizer uses the number of active row partitions rather than the total number of row partitions.

In cases where there is a range constraint between the partitioning column of a PI relation and the other table that can be used to generate a partition-level constraint, the AMP software applies dynamic row partition elimination to the operation.

Sliding-window merge joins are generally not feasible for join operations on a PI table when the number of combined partitions in a window is significantly fewer than the number of populated, noneliminated combined partitions that must be joined.

Note the following:

- **Dynamic Row Partition Elimination for Equality Join Terms on the Primary Index Columns of a Character-Partitioned Table:**
A merge join with dynamic row partition elimination is not supported for character-partitioned joins. The Optimizer never selects this join method as a direct join to a character-partitioned PI table.
- **Dynamic Row Partition Elimination for a Sliding-Window Merge Join Between a Primary-Indexed Table and a NoPI Table:**
The Optimizer cannot use dynamic row partition elimination for a sliding-window merge join between a primary-indexed table and a NoPI table without first spooling the NoPI table.
- **Dynamic Row Partition Elimination for a Sliding-Window Merge Join When One of the Relations is Column-Partitioned:**
Sliding-window merge join with dynamic partition elimination must first spool and then sort the column-partitioned relation when one of the relations is column-partitioned.

Example of a Sliding-Window Merge Join

In the following example, the 2 tables are joined on their primary indexes. The WHERE conditions eliminate all but 2 row partitions for partitioning level 1, and 15 row partitions for partitioning level 2 of orders.

The WHERE conditions eliminate all but 1 row partition for partitioning level 1, and 19 row partitions for partitioning level 2 of lineitem.

After the database applies these conditions, it must join 30 combined partitions of the combined partitioning expression for orders to 19 combined partitions of the combined partitioning expression of lineitem, making a total of 49 combined partitions.

Assume the following properties for this specific query:

- The remaining combined partitions are all populated
- Only 18 combined partitions can be joined at a time based on the PPICacheThrP setting for your system

Because of this situation, the database can join the 2 tables using a sliding-window merge join, assuming the Optimizer estimates it to be the most cost-effective join method.

Assume that the Optimizer decides that 8 combined partitions from orders are to be joined to 10 combined partitions from lineitem at a time.

Also assume that the Optimizer clusters the combined partitions from the respective tables into the following sets of windows for making the join.

- The Optimizer divides the combined partitions of orders into 4 sets of 8, 8, 8, and 6 combined partitions.
- The Optimizer divides the combined partitions of lineitem into 2 sets of combined partitions of 10 and 9 combined partitions.

The database directly merge joins each set of combined partitions from orders to each set of combined partitions from lineitem, making a total of 8 single-window merge joins.

The definition DDL text for the 2 tables, orders and lineitem, to be joined in this example query is as follows:

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (
  RANGE_N(o_custkey   BETWEEN 0
              AND 49999
              EACH 100),
  RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
              AND      DATE '2006-12-31'
              EACH INTERVAL '1' MONTH))
UNIQUE INDEX (o_orderkey);
```

```

CREATE TABLE lineitem (
  l_orderkey      INTEGER NOT NULL,
  l_partkey       INTEGER NOT NULL,
  l_suppkey       INTEGER,
  l_linenumber    INTEGER,
  l_quantity      INTEGER NOT NULL,
  l_extendedprice DECIMAL(13,2) NOT NULL,
  l_discount      DECIMAL(13,2),
  l_tax           DECIMAL(13,2),
  l_returnflag    CHARACTER(1),
  l_linestatus    CHARACTER(1),
  l_shipdate      DATE FORMAT 'yyyy-mm-dd',
  l_commitdate    DATE FORMAT 'yyyy-mm-dd',
  l_receiptdate   DATE FORMAT 'yyyy-mm-dd',
  l_shipinstruct  VARCHAR(25),
  l_shipmode      VARCHAR(10),
  l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey)
PARTITION BY (
  RANGE_N(l_suppkey BETWEEN 0
            AND 4999
            EACH 10),
  RANGE_N(l_shipdate BETWEEN DATE '2000-01-01'
            AND DATE '2006-12-31'
            EACH INTERVAL '1' MONTH));

```

Given these table definitions, the Optimizer selects the direct sliding-window merge join method for the join plan to join the 49 qualifying combined partitions when it processes the following query:

```

SELECT *
FROM orders INNER JOIN lineitem
WHERE o_orderkey = l_orderkey
AND   o_orderdate BETWEEN DATE '2005-04-01'
                        AND   DATE '2006-06-30'
AND   o_custkey IN (618, 973)
AND   l_shipdate  BETWEEN DATE '2005-04-01'
                        AND   DATE '2006-10-31'
AND   l_suppkey = 4131;

```

Hash Join

About the Hash Join

Hash join is a method that performs better than merge join (see [Merge Join](#)) and equality product join (see [Product Join](#)) 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. In some circumstances, an in-memory hash join provides better performance by making better use of the cache, bulk join condition evaluation, and the use of SIMD instructions.

Note:

Unless otherwise noted, the general information about hash joins in the hash join sections also applies to in-memory hash joins.

Vantage can use the various forms of hash join to handle the following join types:

- Hash inner joins (classic, dynamic hash join, and dynamic hash join with dynamic row partition elimination)
- Hash inclusion semijoins (dynamic hash join only; does not apply to in-memory hash joins)
- Hash exclusion semijoins (dynamic hash join only; does not apply to in-memory hash joins)
- Left, right, and full outer hash joins (classical and dynamic hash joins only)
- Hash joins with dynamic row partition elimination (dynamic hash join only)

See [Classic Hash Join](#) and [Dynamic Hash Joins](#) for further information about these hash join types.

Depending on whether spooling the large table in the join might be avoided, the Optimizer might substitute a dynamic hash join (see [Dynamic Hash Joins](#)) or an in-memory dynamic hash join (see [Dynamic In-memory Hash Joins](#)) in place of a standard hash join. If join conditions are specified on a column set that is not the primary index, relocation of rows must be completed 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](#) for details about the negative effect of skewed data on hash join performance).

Like other join methods, hash joins are costed, and the Optimizer compares their cost to the cost of other eligible join methods and plans to determine the least costly plan for a query.

The Optimizer also decides whether or not to create in-memory optimized spools.

In-memory optimized spools are column-partitioned spools with a maximum of 4 column partitions, with the following contents:

- A partition for the hash values.
- A partition for the join column with the best selectivity that is used in a equality join condition of fixed length.
- A partition for all the fixed length columns used in the equality join condition.

- A partition for all the variable-length columns, residual join condition columns, and projected columns.

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 2 join tables, so named because it is used to build the hash table.
Fanout	The maximum number of hash join partitions to be created. When rows are fanned out, they are partitioned into several hash partitions.
Hash join or In-memory 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.
In-memory hash table	A memory-resident table built from left input, designed to increase cache efficiency during join comparisons. It is also designed to enable SIMD instruction execution.
Probe table	The larger of the two join tables in the hash join. Rows from this table are used to probe rows in the hash table for a hash match before the join conditions are evaluated.
Partition	A segment of a table in a hash join. Tables are segmented into a number of hash join partitions. Hash join partitions can be memory-resident, in which case they also constitute the hash table, or a spool. The limit on the number of hash join partitions for a hash join operation is 50. Note that hash join partitions have no relationship to the partitions of a row-partitioned or column-partitioned table or join index (see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144 and <i>Teradata Vantage™ - Database Design</i> , B035-1094). They are entirely different, unrelated things.
Spillover	During the hash table building process, Vantage allocates one segment at a time for storing the hash table rows. When the maximum number of allocated segments (MaxHTSegs) is reached, spillover occurs and the hash join process must start by using the right row hash to probe the hash table looking for a match row to join.

Hash Joins and Performance

The Optimizer might use a hash join instead of a merge join for better performance in the following situations.

- At least one join key is not indexed.
- To provide a performance improvement for the join step.

By using a hash table, hash joins eliminate the sort used prior to a merge join.

See [Controlling the Size of Hash Tables and In-memory Hash Tables](#) for more information about optimizing the performance of hash joins.

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.

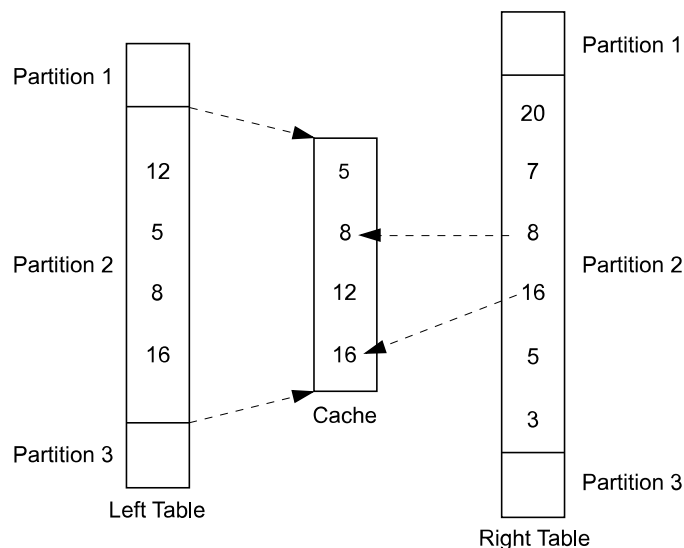
Vantage uses a variant of classic hash join commonly referred to as *hybrid* hash join, as well as a variant of the hybrid hash join referred to as dynamic hash join (see [Dynamic Hash Joins](#)). The method deals with build tables that are too large to fit into available memory by dividing them into chunks called hash join partitions that are small enough to fit into memory (see [Partitioning the Smaller Table of a Hash Join Pair](#) for details about how the system does this).

The classic hash join can be applied to left, right, and full outer joins as well as inner joins.

The process applied by the hybrid hash join algorithm is provided by the following process:

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

The following graphic illustrates the hybrid hash join process:



Hash bucket entries cluster the entries with the same rowhash value together. This facilitates faster searches for a row using its row hash value because each hash bucket entry only needs to be visited once.

Classic In-memory Hash Join

All of the considerations described for classic hash joins also apply to in-memory hash joins. There are a few differences, one of which is that more memory may be used for an in-memory hash join.

The in-memory hash join algorithm is designed for bulk processing; that is, each step of the process that is described in [Classic Hash Join](#) is enhanced to operate on multiple rows. This algorithm helps to improve the memory access pattern and thus, cache efficiency.

Another difference is in the hash table data structure. The hash table structure for in-memory hash joins follows the in-memory spool partition format and stores values for each partition together.

Partitioning the Smaller Table of a Hash Join Pair

A hash table is derived from the smaller table in a hash join pair. It is a single-partition, memory-resident data structure that contains a hash array as well as the rows in the larger table that the hash array points to. The system configures the smaller table in a hash join operation as an ordered linked list of rows that belong to a particular range of hash values. Depending on its size, this table can be decomposed into several smaller partitions.

When the smaller table is too large to fit into the memory available for hash join processing, the system splits it into several smaller partitions. Each partition is small enough to fit into the available space. Partition size is controlled by the settings of several parameters in the DBS Control record (see [Hash Join Control Variables](#)).

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 hash join partitions required to make the join.

For example, suppose 6 hash join 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 hash join partition that can fit into the available memory. The system then hashes each row into one of the 6 hash join 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 hash join partition of the smaller table, which is copied to a spool, into memory. Rows from the matching hash join partition in the other table, which is also in a spool, can then be matched to the rows in memory.

Note that a row from one hash join partition cannot match a row in the other table that is in a different hash join partition because their respective hash values are different.

Each left table hash join 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 hash join partition 2 of the triple-partitioned right table.

The process of creating the hash join partitions is referred to as *fanning out* in EXPLAIN reports (see [Hash Join Examples](#), where the phrases that describe the hash join and the partitioning of the hash join tables are highlighted in boldface).

Effects of Data Skew on Hash Join Processing

Data skew in the build table can seriously degrade the performance of hash joins. One of the premises of hash join is that the hashing algorithm is good enough to ensure that the build relation can be reduced into

relatively equivalent-sized partitions. When there is a large quantity of duplicate row values in the build table, the hash partitioning algorithm might not partition it optimally. Skew in the probe table can also degrade performance if it results in the probe table being smaller than the build table.

To make allowances for possible skew in either table in a hash join operation, you can use the DBS Control utility to size build table partitions proportionately to their parent hash table (see [Hash Join Control Variables](#)).

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.

For in-memory hash joins, data skew in the build table can also degrade join performance. Skew is more problematic for in-memory hash joins due to the higher default size for an in-memory hash table. With higher hash table sizes, the Optimizer may choose direct access for the right table. But if there is hash table overflow, then a larger right table must be scanned multiple times. When using smaller hash table sizes, the planner may choose to partition the two inputs if the available memory is deemed insufficient to hold the entire left input. During in-memory hash join planning, the Optimizer adjusts the cost more conservatively for skew by inflating the row estimate.

Controlling the Size of Hash Tables and In-memory Hash Tables

You can control the size of the hash table using the HTMemAlloc and HTMemAllocBase DBS Control fields (see [Hash Join Control Variables](#)). 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.

If you need to increase the size of the in-memory hash table, contact your Teradata Global Support Center representative.

Hash Join Control Variables

You can access the hash join-related parameters HTMemAlloc and SkewAllowance using the DBS Control utility (see *Teradata Vantage™ - Database Utilities*, B035-1102). Use them to optimize the performance of your hash joins. HTMemAllocBase is an internal DBS Control field that can only be accessed by Teradata Services. Contact your technical support team if you suspect the value of this field needs to be changed.

Note:

HTMemAlloc does not apply to in-memory hash joins; SkewAllowance is used for in-memory hash joins.

DBS Control Field	Function
HTMemAlloc	Varies the hash table size by calculating a percentage of the HTMemAllocBase value. The larger the specified percentage, the larger the hash table. The default is 10. A setting of zero prevents hash joins from being used as optimizations. This setting should be changed only under the direction of the Teradata Global Support Center.
HJ2IMHJ	Determines whether the Optimizer preferentially chooses the in-memory hash join method for performing hash joins and dynamic hash joins. The default value is FALSE (not enabled). When set to FALSE, the optimizer costs an in-memory hash join as a new join method and picks it based on cost. When set to TRUE, the optimizer uses the in-memory hash join method whenever possible, in preference to any other hash join method, regardless of cost.
SkewAllowance	Provides for data skew by making the size of each partition smaller than the hash table. Valid input values range from 20 to 80 (percent). 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.

For more information about DBS Control fields, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Hash Join Examples

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](#)).

Hash table memory allocation is set at 5% and skew allowance is set at 75% (see [Effects of Data Skew on Hash Join Processing](#)) for the partial EXPLAIN.

```
EXPLAIN
SELECT employee.empnum, department.deptname, employee.salary
FROM employee, department
WHERE employee.location = department.location;
```

The following shows a portion of the EXPLAIN output:

```
...
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%

The partial EXPLAIN from the following example shows how the Optimizer handles an in-memory hash join:

```
EXPLAIN
SELECT pit103.c2,pit104.c3,cpt1.c2
FROM pit104,pit103,cpt1
WHERE cpt1.c4 = pit103.c4 AND pit103.c3 = pit104.c4 AND cpt1.c2 > 100;
```

...

5) We execute the following steps in parallel.

- 1) We do an all-AMPs RETRIEVE step from TEST.pit104 by way of an all-rows scan with a condition of ("NOT (TEST.pit104.c4 IS NULL)") into Spool 2 (all_amps), which is built locally on the AMPs. Spool 2 is built as **in-memory optimized spool with 3 column partitions**. The size of Spool 2 is estimated with low confidence to be 20 rows (500 bytes). The estimated time for this step is 0.37 seconds.
- 2) We do an all-AMPs RETRIEVE step from TEST.pit103 by way of an all-rows scan with a condition of ("(NOT (TEST.pit103.c4 IS NULL)) AND (NOT (TEST.pit103.c3 IS NULL))") into Spool 3 (all_amps), which is duplicated on all AMPs. Spool 3 is built as **in-memory optimized spool with 3 column partitions**.

The size of Spool 3 is estimated with low confidence to be 400 rows (11,600 bytes). The estimated time for this step is 0.37 seconds.

- 6) We do an all-AMPs JOIN step from Spool 2 (Last Use), which is joined to Spool 3 (Last Use). Spool 2 and Spool 3 are joined using a **single partition in-memory hash join**, with a join condition of ("c3 = c4"). The result goes into Spool 4 (all_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 4 by the hash code of (TEST.pit103.c4). The size of Spool 4 is estimated with no confidence to be 90 rows (2,250 bytes). The estimated time for this step is 0.33 seconds.
- 7) We do an all-AMPs BULK RETRIEVE step from 3 column partitions of TEST.cpt1 with a condition of ("(TEST.cpt1.c2 >= 101) AND (NOT (TEST.cpt1.c4 IS NULL))") into Spool 5 (all_amps), which is duplicated on all AMPs. Then we do a SORT to order Spool 5 by the hash code of (TEST.cpt1.c4). The size of Spool 5 is estimated with no confidence to be 140 rows (2,940 bytes). The estimated time for this step is 0.11 seconds.
- 8) We do an all-AMPs JOIN step from Spool 4 (Last Use) by way of a RowHash match scan, which is joined to Spool 5 (Last Use) by way of a RowHash match scan. Spool 4 and Spool 5 are joined using a merge join, with a join condition of ("c4 = c4"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 67 rows (3,618 bytes). The estimated time for this step is 0.56 seconds.

...

Dynamic Hash Joins

Dynamic hash join provides the ability to do a binary or *n*-way equality join directly between one or more small tables (or relations) and a large table on non-primary index columns without placing the large table into a spool. For dynamic hash join to be used, each of the small tables must be small enough to fit in a single hash join partition.

Dynamic hash join can be used only when two tables are joined based on non-primary index columns, and small tables, referred to as the *left tables*, are very small compared to the large table, referred to as the *right table*.

The process is as follows:

1. Duplicate the smaller tables.
2. Place each of the smaller tables in a hash array
3. Read a row from the right table
4. Compute the row hash code
5. Match each right row with all rows in each of the hash arrays having the same row hash.

6. Join the matching rows.

The dynamic hash join can be applied to left, right, and full outer joins as well as inner joins, and can also take advantage of equality conditions for dynamic partition elimination.

The inclusion hash join and the exclusion hash join can also use dynamic partition elimination when you specify an equality condition.

Dynamic In-memory Hash Joins

All of the considerations described for dynamic hash joins also apply to in-memory hash joins. There are a few differences, one of which is that the amount of memory in an in-memory hash join can go up to the default value of 100 MB or up to the value set (see [Controlling the Size of Hash Tables and In-memory Hash Tables](#)).

The in-memory dynamic hash join algorithm is designed for bulk processing, and enhanced to operate on multiple rows. This algorithm helps to improve the memory access pattern and thus, cache efficiency.

The hash table data structure that is used for an in-memory dynamic hash join is the same as that used for the classic in-memory hash join.

AllRowsOneAMP In-Memory Hash Joins

This type of hash join is one in which the left table is duplicated to one AMP. The in-memory hash join step reads the rows from one AMP, builds the hash table, and duplicates the hash table segments to all AMPs. All the AMPs receive the hash table segments and continue with probe and output building. One copy of the hash table segments are shared by all the AMPs within a node.

The in-memory hash join is the only consumer of AllRowsOneAMP spool.

This improves I/O performance, compared to using duplicate spools for in-memory hash joins. The performance improvement increases as the number of AMPs in the system increases.

There are two modes of execution:

- In *default mode*, a classic or dynamic in-memory hash join where the left table is duplicated will become an AllRowsOneAMP in-memory hash join. (This is not the case for DPE in-memory hash joins and in-memory hash joins that are part of PRPD joins.)
- In *resource consumption mode*, in addition to default mode behavior, a classic hash join where both tables are redistributed will become an AllRowsOneAMP-Direct or AllRowsOneAMP-Local in-memory hash join. The decision between Direct or Local is cost-based.

In both modes, the Optimizer makes heuristic decisions. Resource consumption mode should be used only to improve resource consumption (I/O, CPU) at the expense of the elapsed time. Resource consumption mode is disabled by default. To enable it, contact the Teradata Support Center.

Hash Joins, In-Memory Hash Joins, and Performance

The Optimizer may use a hash join instead of a merge join of tables for better performance if at least one join key is not indexed.

The hash join eliminates the sort used prior to the merge join by using a hash table instead.

You can optimize system usage of hash joins with the following DBS Control fields:

DBS Control Field	Best Initial Setting
HTMemAlloc Note: Changing HTMemAlloc does not affect in-memory hash joins.	2
SkewAllowance	75

For information on setting hash join-related DBS Control fields, see the DBS Control fields HTMemAlloc, HJ2IMHJ, and Skew Allowance in *Teradata Vantage™ - Database Utilities*, B035-1102.

For information on strategies for using hash join-related DBS Control fields, see *Teradata Vantage™ - Database Administration*, B035-1093.

Recommendations for In-Memory Hash Joins

The following are recommendations when using in-memory hash joins:

- Make the better selective column a fixed-length column and make the fixed-length column either a column partition by itself or a narrow partition.
- Collect multicolumn statistics if join conditions involve multiple columns because incomplete statistics can make the cardinality of the table incorrect. This situation can lead the Optimizer to think that the table can fit in the available memory (by default, 100 MB) and therefore it does not cost hash table overflows, which eventually lead to a lesser estimated cost. This same situation may not occur with a hash join, since it uses less memory for building the hash table and it also uses fan-out, which performs better. Based on these facts, it is recommended that you collect multicolumn statistics on the join columns from the left table.
- Keep your statistics updated to avoid the problem mentioned in the previous bullet.

Outer Hash Joins

Vantage supports left outer hash joins, right outer hash joins, and full outer hash joins. These types of joins support in-memory hash join methods. The processing for each of these join types is described in the topics that follow.

Exclusion Hash Join

An exclusion hash join is a hash join where only the rows that do not satisfy (meaning that they are NOT IN) any condition specified in the request are joined. In other words, exclusion hash join finds rows in the first table that do not have a matching row in the second table.

Exclusion hash join is an implicit form of outer join. Exclusion hash joins are not supported for in-memory hash joins.

Inclusion Hash Join

An inclusion hash join is a hash join where the first right relation row that matches the left relation row is joined to it.

Note:

Inclusion hash joins are not supported for in-memory hash join methods.

Nested Join

A 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. The nested join is one of the most high-performing joins available because it is the only join type that need not touch all AMPs in order to join the relations.

About Nested Joins

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](#).
- Remote nested join is described in [Remote Nested Join](#).

Local Nested Join

Definition of the Local Nested Join

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.

The following local nested join algorithms are available:

- Slow path (see [Slow Path Local Nested Join](#))
- Fast path (see [Fast Path Local Nested Join](#))

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

A local nested join can be applied 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, stages 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](#).

IF the equality condition is on this index type ...	THEN the left table is ...
USI	<ol style="list-style-type: none"> 1. Hash-redistributed based on the joined column. 2. Nested joined with the right table.

IF the equality condition is on this index type ...	THEN the left table is ...
	<ol style="list-style-type: none"> 3. The resulting rows are redistributed by row hashing the rowID of the right table rows. 4. The rowID is used to retrieve the data rows from the right table to complete the join.
NUSI	<ol style="list-style-type: none"> 1. Duplicated on all AMPs. 2. Nested joined with the right table. 3. The resulting rows are redistributed by row hashing the rowID of the right table rows. 4. The rowID is used to retrieve the data rows from the right table to complete the join.

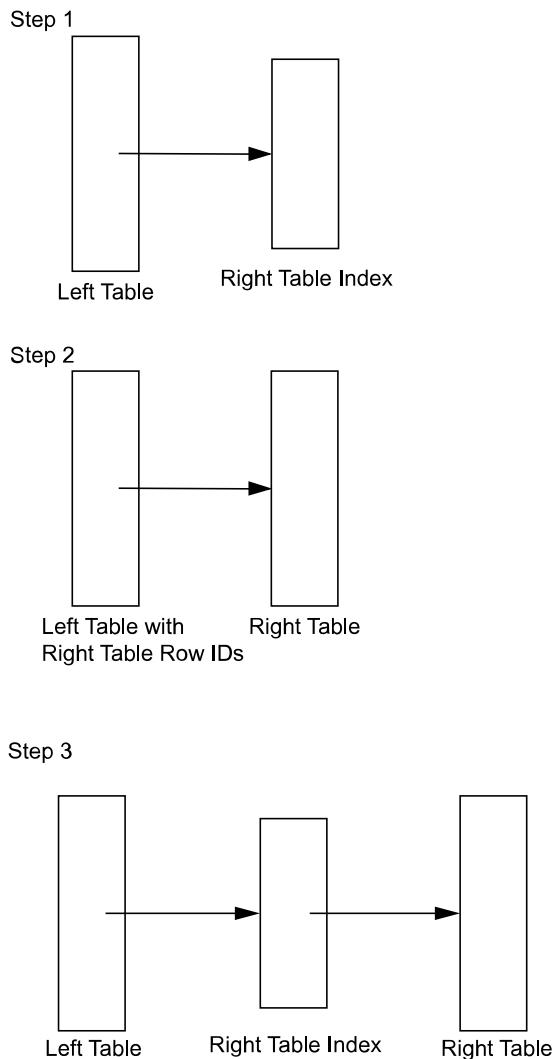
Slow Path Local Nested Join

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.

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



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 dept_name, name, yrs_exp
FROM employee, department
WHERE employee.emp_no = department.mgr_no
AND   department.dept_no = 100;
```

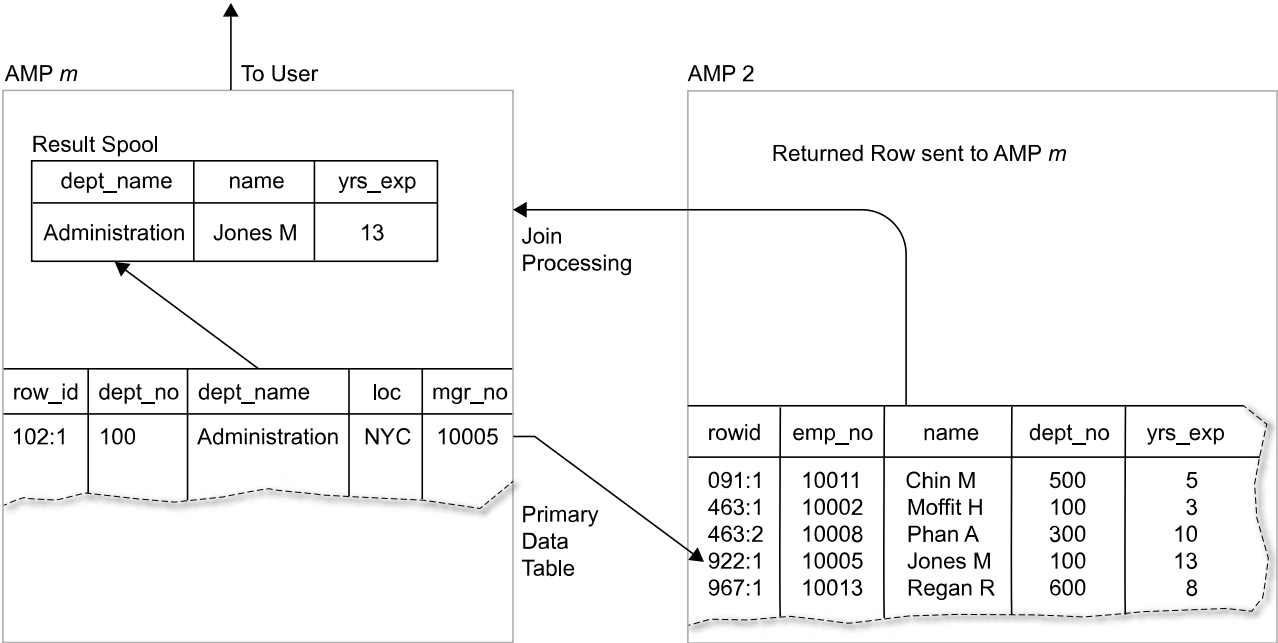
To process this query, the Optimizer uses the unique primary index value `dept_no=100` to access the AMP responsible for the *department* row with that value. The hash code for the *mgr_no* value in that row is calculated.

Note that this *mgr_no* value is the same as the value of *emp_no* (the unique primary index of the employee table) for the employee who manages department 100. Thus, the hash code that is calculated for *mgr_no* is the same as the hash code for the equivalent *emp_no* value.

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

The name and yrs_exp information in this row is sent back to the initiating AMP, which places the information, plus the dept_name for Department 100, in a result spool. This information is returned to the user.

This 2-AMP process is illustrated in the following graphic:



The following 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 Table

e_num	e_name	dept
1	Brown	200
2	Smith	310
3	Jones	310
4	Clay	400
5	Peters	150
6	Foster	400

e_num	e_name	dept
7	Gray	310
8	Baker	310

Column e_num is the UPI and PK for the table, and dept is a FK.

Department Table

dept	dept_name
400	Education
150	Payroll
200	Finance
310	Mfg

Column dept is the UPI and PK for the table.

Fast Path Local Nested Join

Fast Path Local Nested Join Process

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

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

1. Read a row from the left base table and record its hash value.
2. Read the next row from the right NUSI subtable that has a row hash \geq to that of the left base table row.
 - If the row hash values are equal, then join the two rows.
 - If the row hash values are not equal, then use the larger row hash value to read the row from the right NUSI subtable.

The following SELECT request 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 of the Remote Nested Join

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 request 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.

Remote Nested Join Process

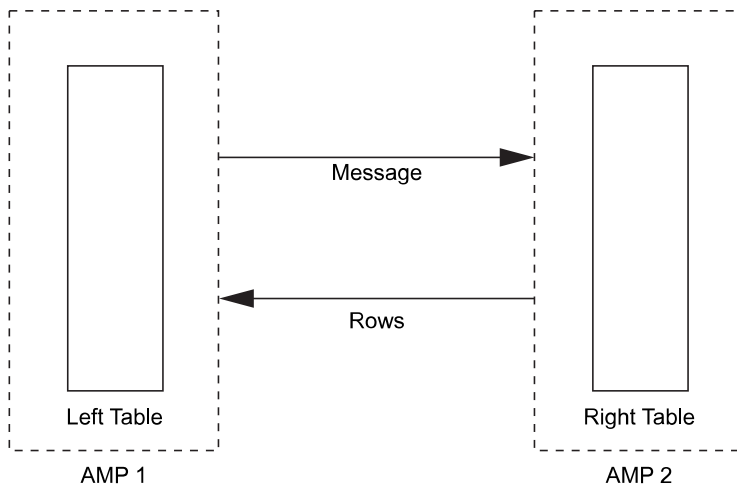
Remote nested join is used for the condition where one table contains the key to the table with which it is to be joined. The key can be of any of the following database objects:

- Unique primary index (UPI)
- Nonunique primary index (NUPI)
- Unique secondary index (USI)
- Nonunique secondary index (NUSI)
- Unindexed 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.

The following graphic illustrates the remote nested join process:



Examples of Remote Nested Join Conditions

A remote nested join can be used when there is no equality condition between the primary indexes of the 2 tables and other conditions. This is illustrated by the following example conditions:

```
(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.

The following 2 examples illustrate cases where there is a suitable join condition:

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

(table_1.NUPI = constant)
AND (table_2.UPI = table_1.field)
AND (few_rows_returned_from_the_table_1.NUPI = constant)
```

Nested Join Examples

Circumstances for the Nested Join Examples

This section provides the partial EXPLAIN outputs for the same two-table join under the following circumstances:

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

Table Definitions

For these examples, assume the following table definitions:

Table 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 request 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

About a Join Plan Without Nested Join

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

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

The total estimated completion time is 1 hour 15 minutes.

EXPLAIN Output for Unoptimized Join Plan

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

- 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.

Join Plan With Nested Join

About a Join Plan With Nested Join

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

The total estimated completion time is 0.46 seconds.

The estimated performance improvement factor is 9782.

EXPLAIN Output for Optimized Join Plan

The following partial EXPLAIN output is generated:

- 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.

Exclusion Join

Exclusion Join -- SQL Operators That Often Cause an Exclusion Join Operation

Exclusion join is a product, merge, or hash 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.

Exclusion product joins with dynamic row partition elimination are not supported for 8-byte partitioning.

Also see [Inclusion and Exclusion Product Joins With Dynamic Row Partition Elimination](#).

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

- Exclusion merge join (see [Exclusion Merge Join](#))
- Exclusion product join (see [Exclusion Product Join](#))
- Exclusion hash join (see [Hash Join](#))

Exclusion Merge Join

Exclusion Merge Join Process

1. The left and right tables are distributed and sorted based on the row hash values of the join columns.
2. For each left table row, read all right table rows having the same row hash value until one is found that matches the join condition.
3. Produce the join result.

If no matching right table rows are found, return the left row.

Examples of Exclusion Merge Joins

The following SELECT request is an example for which the Optimizer uses an exclusion merge join for its join plan:

```
SELECT name
FROM employee
WHERE dept_no NOT IN (SELECT dept_no
                      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 on the same AMP.
3. The spool 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 dept_no is not 600 is placed in a result spool on each AMP.

When the last AMP has completed its portion of the join, the contents of all result spools are sent to the user 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](#)), with the following exceptions:

- Multiple rows are retrieved in stage 2.
- Stages 2 and 4 are combined in the exclusion merge join, and redistribution occurs instead of duplication.
- Stage 3 is removed.
- Stage 5 is changed to an exclusion product join.

Consider the following employee and customer tables for the next exclusion merge join example:

Employee Table

e_num	name	job_code
1	Brown	512101
2	Smith	412101
3	Jones	512101
4	Clay	412101
5	Peters	512101
6	Foster	512101

e_num	name	job_code
7	Gray	413201
8	Baker	512101

Column e_num is the UPI and PK, and job_code is a FK.

Customer Table

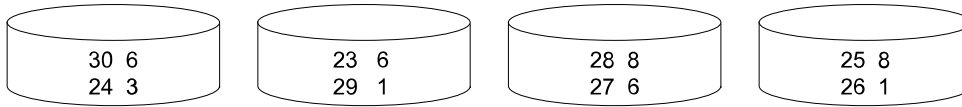
cust	sales_enum
23	6
24	3
25	8
26	1
27	6
28	8
29	1
30	6

Column cust is the UPI and PK, and sales_enum is a FK.

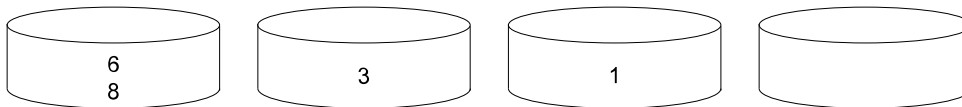
The graphic that follows illustrates the row redistribution caused by the following SELECT request:

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

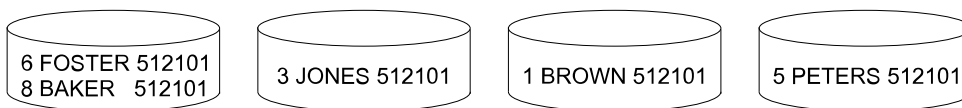
CUSTOMER ROWS HASH DISTRIBUTED ON CUST_NUM (UPI):



CUSTOMER.SALES_ENUM (AFTER HASHING AND DUPLICATE ELIMINATION):



QUALIFYING EMPLOYEE ROWS STILL DISTRIBUTED ON ENUM (UPI):



Exclusion Product Join

Exclusion Product Join Process

1. For each left table row, read all right table rows from the beginning until one is found that can be joined with it.
2. Produce the join result.

If no matching right table rows are found, return the left row.

Note:

Vantage does not support dynamic row partition elimination for exclusion product joins for 8-byte partitioning.

Exclusion Product Join Example

The following request returns the names of those employees who do not work in Chicago:

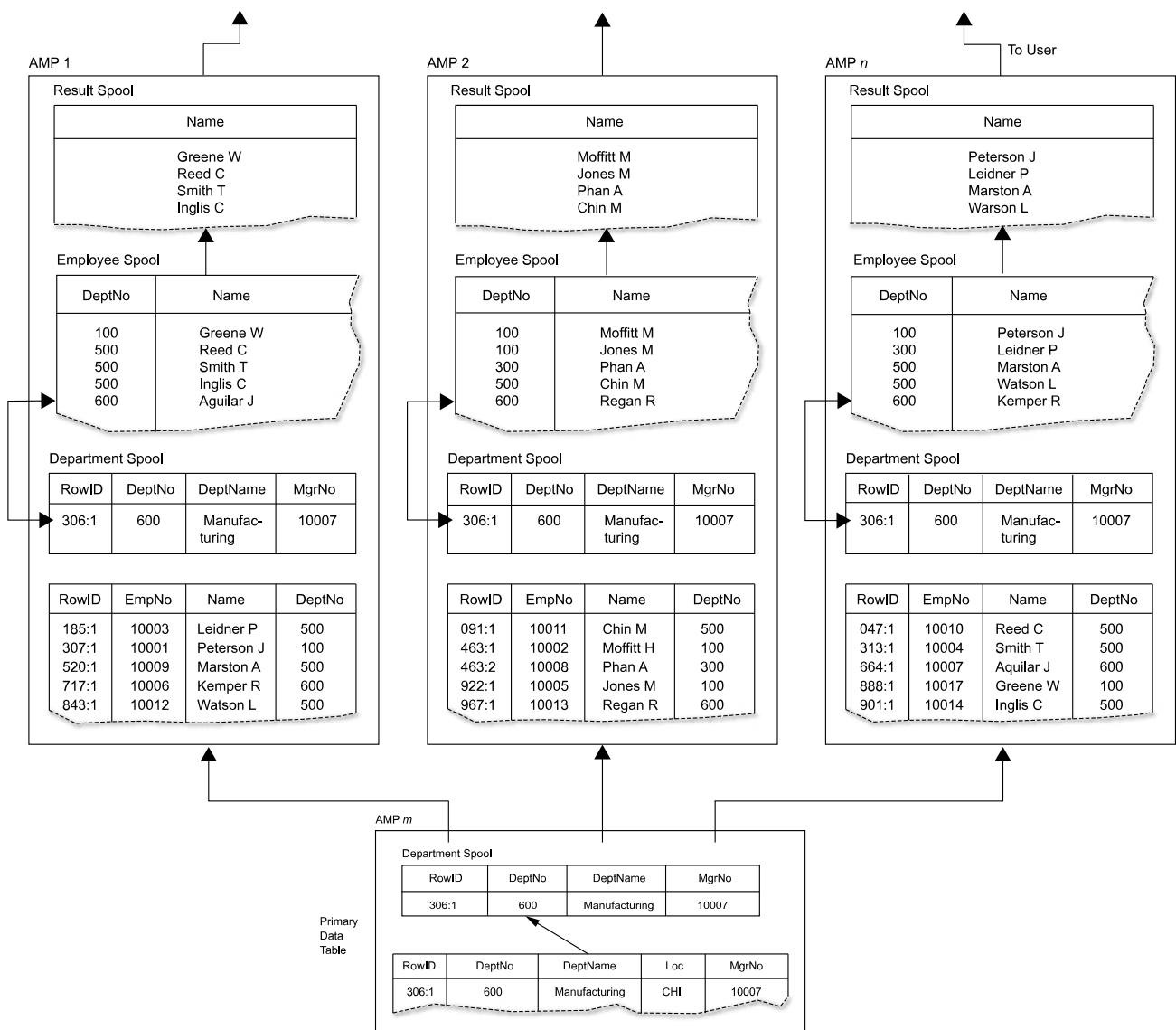
```
SELECT name
FROM employee
WHERE dept_no NOT IN (SELECT dept_no
                      FROM department
                      WHERE loc = 'CHI');
```

Because the subquery returns only one row, the Optimizer selects an exclusion product join for the join plan using 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 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 dept_no is not 600 is placed in a result spool 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.

The following graphic illustrates this process:



Inclusion Join

Definition of the Inclusion Join

An inclusion join is a product, merge, or hash join where the first right table row that matches the left row is joined.

Following are the types of inclusion join:

- Inclusion merge join
- Inclusion product join
- Inclusion hash join

Also see [Inclusion and Exclusion Product Joins With Dynamic Row Partition Elimination](#).

Inclusion Merge Join Process

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

Inclusion Product Join Process

1. For each left table row read all right table rows from the beginning until one is found that can be joined with it.
2. Return the left row if a matching right row is found for it.

Inclusion and Exclusion Product Joins With Dynamic Row Partition Elimination

About Product Joins With Dynamic Row Partition Elimination

When the Optimizer determines that the least costly method for making a join between a row-partitioned table and another relation on either an IN or NOT IN condition, it can choose to use either an inclusion or exclusion product join with DPE, respectively. Whether it uses an inclusion or exclusion product join with DPE depends on the condition on which the relations are joined.

IF the join condition is on this term ...	THEN the Optimizer chooses this product join ...
IN	Inclusion product join with DPE.
NOT IN	Exclusion product join with DPE.

The performance enhancement seen for inclusion and exclusion DPE product joins over standard inclusion and exclusion product joins occurs for two reasons.

- The number of comparisons between left relation and right relation rows is reduced, which in turn reduces the I/O on the left and right relations and the CPU costs to perform the join.
- The following is true for the case when one relation in a join is a table and the other is a spool:

When the number of values in the spool is small, resulting in fewer populated spool partitions than populated row partitions from the row-partitioned table, those table partitions that are not in spool need not be either read or joined, reducing both the I/O and CPU costs of the join.

Because the performance improvement gained from DPE joins is highly dependent on the number of rows in each row partition (and so the number of row comparisons that must be made), the Optimizer does not choose a DPE join if single-column PARTITION statistics have not been collected on the outer (row-partitioned) table in the join. PARTITION statistics improve the ability of the Optimizer to estimate the number of comparisons that must be made. The Optimizer does not choose a DPE join when there is a no confidence estimate on the cardinality of the subquery spool.

For example, suppose you create the following tables and submit the given SELECT request against those tables:

```
CREATE SET TABLE MWS.t1, Fallback, NO BEFORE JOURNAL,
NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  a INTEGER,
  b INTEGER)
PRIMARY INDEX (a);

CREATE SET TABLE MWS.t2, Fallback, NO BEFORE JOURNAL,
NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  a INTEGER,
  b INTEGER,
  c INTEGER,
  d INTEGER)
PRIMARY INDEX (a)
PARTITION BY (RANGE_N(b BETWEEN 1 /* Partitioning level one. */
                      AND 100
                      EACH 7,
NO RANGE OR UNKNOWN),
RANGE_N(c BETWEEN 1 /* Partitioning level two. */
                      AND 100
                      EACH 10,
NO RANGE OR UNKNOWN),
RANGE_N(d BETWEEN 1 /* Partitioning level three. */
                      AND 100
                      EACH 20,
NO RANGE OR UNKNOWN));

SELECT *
FROM t2
WHERE (b,c) IN (SELECT a,b
                FROM t1);
```


For this request to be eligible for an inclusion or exclusion product join with DPE, you must collect the following statistics:

```
COLLECT STATISTICS t1 COLUMN(a);
COLLECT STATISTICS t2 COLUMN(PARTITION);
```

The following rules apply to collecting these statistics:

To enable this type of join...	You must collect statistics on the ...
Product (all kinds) with DPE	<ul style="list-style-type: none"> • join column set of the table that must be duplicated to enable an inclusion or exclusion product join with DPE. • any right table predicate columns that are applied to the table that must be duplicated.
Inclusion or exclusion product with DPE	system-derived PARTITION column set of the row-partitioned table.

Another name for the type of join being done with inclusion and exclusion product joins is *semijoin*. As a result, the inclusion and exclusion product joins with DPE can also be referred to by the name product semijoin with DPE.

When a product semijoin with DPE is applied by the Optimizer, the EXPLAIN text for the query indicates *enhanced by dynamic row partition elimination* for the corresponding AMP step. In the following partial EXPLAIN output, this text is highlighted in boldface type:

```
EXPLAIN SELECT COUNT(*)
  FROM t55
 WHERE (b,c) NOT IN (SELECT 1, 1);
```

- 3) We do an INSERT into Spool 5.
- 4) We do an all-AMPS RETRIEVE step from Spool 5 (Last Use) by way of an all-rows scan into Spool 4 (group_amps), which is redistributed by the hash code of (1, 1) to all AMPS. The size of Spool 4 is estimated with high confidence to be 1 row (22 bytes). The estimated time for this step is 0.03 seconds.
- 5) We do a group-AMP RETRIEVE step from Spool 4 (Last Use) by way of an all-rows scan into Spool 7 (all_amps), which is duplicated on all AMPS. Then we do a SORT to partition by rowkey. The size of Spool 7 is estimated with high confidence to be 2 rows (30 bytes). The estimated time for this step is 0.03 seconds.
- 6) We do an all-AMPS JOIN step from MWS.t55 by way of an all-rows scan with no residual conditions, which is joined to Spool 7 (Last Use) by way of an all-rows scan. MWS.t55 and Spool 7 are joined using an exclusion product join, with a join condition of (

"(MWS.t55.b = Field_2) AND (MWS.t55.c = Field_3)") **enhanced by dynamic row partition elimination.** The input table MWS.t55 will not be cached in memory, but it is eligible for synchronized scanning. The result goes into Spool 3 (all_amps), which is built locally on the AMPs. The result spool file will not be cached in memory. The size of Spool 3 is estimated with index join confidence to be 2,370,668 rows (35,560,020 bytes). The estimated time for this step is 39.38 seconds.

- 7) We do an all-AMPs SUM step to aggregate from Spool 3 (Last Use) by way of an all-rows scan. Aggregate Intermediate Results are computed globally, then placed in Spool 8. The size of Spool 8 is estimated with high confidence to be 1 row (23 bytes). The estimated time for this step is 3.06 seconds.
- 8) We do an all-AMPs RETRIEVE step from Spool 8 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.02 seconds.

The Optimizer applies an inclusion or exclusion product join with DPE when there are equality join terms between the partitioning columns at one or more partitioning levels of a row-partitioned table, which is the outer table of a semijoin, and another relation. Inclusion and exclusion product joins with DPE enhance the performance of such queries for those cases where the row-partitioned table is large and a costly sort in preparation for a regular inclusion or exclusion merge join can be avoided. Such queries further benefit when the number of rows for the join column selected by the subquery is small because in that case, the number of values in the spool is small, resulting in fewer populated spool row partitions than populated table partitions of the row-partitioned table. As a result, the table row partitions that are not in spool neither need to be read nor joined, reducing both the I/O and CPU costs of the join.

For example, assume you have the following table definitions and cardinalities:

```
CREATE SET TABLE t1 (
  a INTEGER,
  b INTEGER,
  c INTEGER)
PRIMARY INDEX ( a );
CREATE SET TABLE t8 (
  a INTEGER,
  b INTEGER,
  c INTEGER)
PRIMARY INDEX (a)
PARTITION BY (RANGE_N(c BETWEEN 1
                      AND 1200
                      EACH 30,
```

```
NO RANGE OR UNKNOWN),
RANGE_N(b BETWEEN 1
        AND 11000
        EACH 7,
NO RANGE OR UNKNOWN));
```

Table Name	Table Cardinality (rows)
t1	1,000
t8	9,000,000

For the following query, the number of rows returned by the subquery is small, so it is possible to eliminate many row partitions in t8 dynamically, making a product join cost effective. Additionally, it is not necessary to sort the t8 rows to make the join.

```
SELECT COUNT(*)
FROM t8
WHERE (b,c) IN (SELECT a,b
                FROM t1
                WHERE c = 1);
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
Count(*)
-----
        65
```

On the other hand, if the Optimizer were to use a merge join instead of an inclusion product join with DPE, then the t8 rows would have to be sorted, which takes the majority of the processing time, and the query takes 57 seconds to complete rather than 1 second.

```
SELECT COUNT(*)
FROM t8
WHERE (b,c) IN (SELECT a,b
                FROM t1
                WHERE c = 1);
*** Query completed. One row found. One column returned.
*** Total elapsed time was 57 seconds.
Count(*)
-----
        65
```

Be aware that the performance enhancement achieved with inclusion and exclusion product joins with DPE is not always this great, and the degree of the enhancement depends heavily on the following factors:

- The number of rows per row partition of the row-partitioned table
- The number of columns projected by the subquery

Costing takes into account those cases for which the row-partitioned table has a highly variable number of rows per row partition, and avoids applying an inclusion or exclusion product join with DPE for those tables.

Inclusion Product Join With Dynamic Row Partition Elimination

Inclusion product join with dynamic row partition elimination (DPE), which is designed for use when the outer relation in a join based on an IN term is a row-partitioned table, differ from inclusion product joins without DPE in that the DPE versions of the join are driven from the inner table instead of the outer, or row-partitioned, table, while the non-DPE versions are driven from the outer table.

For example, suppose you have the following table definitions and query against those tables:

```
CREATE SET TABLE MWS.t1, Fallback, NO BEFORE JOURNAL,
  NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  a INTEGER,
  b INTEGER)
PRIMARY INDEX (a);

CREATE SET TABLE MWS.t2, Fallback, NO BEFORE JOURNAL,
  NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  a INTEGER,
  b INTEGER)
PRIMARY INDEX (a)
PARTITION BY (RANGE_N(a BETWEEN 1 /* Partitioning level 1 */
                      AND 60000
                      EACH 60000,
                      NO RANGE OR UNKNOWN),
              RANGE_N(b BETWEEN -3 /* Partitioning level 2 */
                      AND 31580
                      EACH 1,
                      NO RANGE, UNKNOWN) );

SELECT *
FROM t2
WHERE (b) IN (SELECT a
              FROM t1);
```

If the Optimizer were to choose a standard inclusion product join to execute this request, the system would perform the join as follows:

1. Sort t1 to remove duplicate values of column a.
2. Duplicate the sorted values of t1 in a spool.
3. Read the first row in t2.

If there are no more rows to read, the inclusion product join process is done.

- 4. Search for a match based on the connecting term `t2.b=t1.a` until a match is found or all rows in spool have been read.
- 5. Go to stage 3 and read the next row in `t2`.

The database reads all rows in `t2` sequentially and joins them with the spool in this manner.

For an inclusion product join with DPE, the system does not read `t2` sequentially, nor does it compare each row in `t2` with all rows in the spool.

When a bind term is specified on all, or on a subset of, the partitioning levels of a row-partitioned table, the system can use the information to build a spool having the same row partitioning as the row-partitioned table. It can then join only the equivalent partitioning ranges between the base table row partitions and the spool row partitions.

The setup for the inclusion product join with DPE is the same as that for a standard inclusion product join, except that when the spool is duplicated, it is built with the same partitioning as is defined for the base table, and it is then sorted by partition number. In the previous example, the spool would be row-partitioned using the partitioning expression that is defined for row-partitioned table `t2`:

```
PARTITION BY (RANGE_N(a BETWEEN -3
                        AND 31580
                        EACH 1,
                        NO RANGE, UNKNOWN) );
```

Note that the row partitioning is done on column `a` of the `t1` spool because the bind term `t2.b=t1.a` links partitioning on `t2.b` with `t1.a`.

The system performs the inclusion product join with DPE as follows:

- 1. Sort `t1` to remove duplicate values of column `a`.
- 2. Duplicate the sorted values of `t1` and row-partition them in a spool by the partitioning ranges defined for `t1`.
- 3. Read the first row in the first row partition of the spool.
- 4. Build the list of row partitions dynamically in the row-partitioned table to which the current spool partition is to join.
- 5. Read the first row in the first DPE row partition of the row-partitioned table.
- 6. Compare the row with all rows in the current spool row partition until either a match is found or there are no more rows in the spool row partition.

IF a match is ...	THEN ...
found	return the row-partitioned table row.
not found	go to stage 7.

- 7. Read the next row of the row-partitioned table in the participating DPE partitions.

IF there are ...	THEN go to stage ...
no more rows	6.
more rows	4.

8. Read the first row in the spool with a partition number greater than the current spool row partition.

IF there ...	THEN ...
are no such rows	the join is complete.
is such a row	Go to stage 2.

Exclusion Product Join With Dynamic Row Partition Elimination

The algorithm for an exclusion product join with DPE is similar the algorithm for an inclusion product join with DPE, but with a restriction and with some other differences.

The restriction is that exclusion product join with DPE is only enabled with row-partitioned tables where the partitioning expression at each partitioning level consists solely of a RANGE_N function, and the test expression is a simple column. The reason for this restriction is that rows with null partitioning column values at any bound partitioning level must be joined with all partitions at that level. Because that might eliminate much of the performance improvement gained from DPE, exclusion product join with DPE avoids reading the extra rows of the row-partitioned table by placing all spool rows with null partitioning column values into the error partition (meaning the NO RANGE partition in this case). Because expressions on columns containing nulls might produce non-nulls, there is no simple method of grouping all rows with nulls into one partition other than using the RANGE_N function with a simple column test expression.

To ensure that rows with null partitioning columns are placed in the error partition, the system modifies qualifying RANGE_N expressions to remove both the UNKNOWN and NO RANGE partitions from their definition when they are grouped together as NO RANGE OR UNKNOWN. The join algorithm then joins rows in the error partition with all of the rows of the row-partitioned table.

All connecting terms must reference partitioning columns for the Optimizer to choose an exclusion product join with DPE. If additional non-partitioning columns are components of the connecting conditions, then the Optimizer does not select the exclusion product join with DPE algorithm.

Vantage performs the exclusion product join with DPE as follows:

- Build a list of row partitions that includes all uneliminated partitions in the row-partitioned table.
This is the *unvisited row partition list*, which must be joined with the error partition after all DPE partitions have been joined.
- Read the first row in the first row partition of the spool.
- Dynamically build the list of row partitions in t2 that this current spool row partition is to join.
Remove this set of row partitions from the unvisited row partition list.
- Read the first row in the first DPE partition of the row-partitioned table.

5. Compare the row with all rows in the current spool row partition and with all rows in the error partition.

IF there is ...	THEN ...
a match	go to stage 6.
no match	return the row of the row-partitioned table.

6. Read the next row of the row-partitioned table in the participating DPE partitions.

IF there are ...	THEN go to stage ...
no more rows	7.
more rows	5.

7. Read the first row in the spool with a partition number greater than the current spool partition.

IF there ...	THEN go to stage ...
are no such rows	8.
is such a row	5.

8. Read the first row in the row-partitioned table not eliminated by the unvisited row partition list.

9.

IF ...	THEN ...
all of the connecting term columns are null	do not return the row. Go to stage 12.
some, but not all, of the connecting term columns are null	join the row with the rows in the spool. Go to stage 11.
none of the connecting term columns is null	Go to stage 10.

10. Join this row with all rows in the error partition of the spool.
11. If no match is found, return the row.
12. Read the next row in the row-partitioned table that was not eliminated by the unvisited partition list. If there is such a row, then go to stage 9.

RowID Join

Rules for RowID Joins

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.

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, USI, or join index (if the join index has a rowID of the base table) 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.

RowID Join 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 the join steps for this request 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 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 is sorted into rowID sequence.
5. The rowIDs in the spool 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 together with table_1 values in the rowID join rows.

Stages 2 and 3 are part of a nested join. Stages 4, 5, and 6 describe the rowID join.

The following graphic demonstrates a rowID join:

TABLE ABC; ROWS HASH-DISTRIBUTED ON COL_1 (nupi):

22		B	
22		A	
22		A	
37		B	
37		D	

17		C	
17		D	
10		A	
10		B	
42		C	

17		D	
15		B	
15		A	

29		D	
29		A	
86		C	
86		B	
86		B	

SPOOL FILE WITH QUALIFYING ROWS FROM ABC DUPLICATED ON ALL AMPS:

10		A	
10		B	

10		A	
10		B	

10		A	
10		B	

10		A	
10		B	

SELECTED NUSI ROWS FOR TABLE XYZ:

A	28,1	82,1	89,1
B	15,1	38,1	51,1

A	07,1	47,1	99,1
B	33,1	63,1	72,1

A	19,1	24,1	66,1
B	02,1	45,1	77,1

A	35,1	40,1	94,1
B	12,1	21,1	56,1

THE ROWIDS IN THE SPOOL FILE ARE USED TO EXTRACT THE CORRESPONDING XYZ DATA ROWS. XYZ VALUES IN XYZ DATAROWS TOGETHER WITH ABC VALUES IN THE ROW ID JOIN ROWS, ARE PUT IN THE RESULT SPOOL FILE.

15,1			B
28,1			A
38,1			B
51,1			B
82,1			A
89,1			A

07,1			A
33,1			B
47,1			A
63,1			B
72,1			B
99,1			A

02,1			B
19,1			A
24,1			A
45,1			B
66,1			A
77,1			B

12,1			B
21,1			B
35,1			A
40,1			A
56,1			B
94,1			A

RowID Join Example

Assume that you submit the following SELECT request. The first WHERE condition is on a NUPI and the second is on a NUSI. The Optimizer applies 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

About Correlated Joins

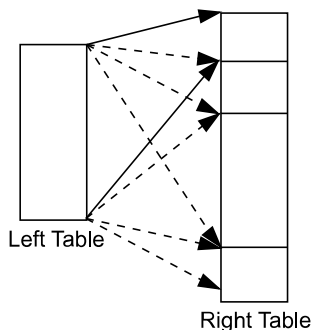
Correlated join constitutes a class of join methods developed to process correlated subqueries. Some types of Correlated join are extensions of the following more general join types:

- Inclusion merge join
- Exclusion merge join
- Inclusion product join
- Exclusion product join

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

Each basic type of correlated join 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 Process](#)) 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 the following 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](#). This join comes in the following 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 Process](#).

- Correlated exclusion product join

Correlated version of standard exclusion product join. See [Exclusion Product Join](#).

- 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.

Self-Join

Self-Join Example

If a join is defined as a request in which rows are retrieved from more than one table, a self-join can loosely be defined as a request in which rows are retrieved from a single table that is redefined as two tables by using correlation names to appear as if there are 2 copies of the same table under different names.

Suppose you want to determine all two-way pairings of employees who live in the same country. To determine this, you would use a self-join of the employee table.

The rows from employee might look something like this:

employee			
emp_num	name	country	dept_num
113722	Lopes	United States	115
225985	Ghazal	United States	115
577321	Korlapati	United States	115
783904	Ramesh	India	378

employee			
emp_num	name	country	dept_num
799106	Manjula	India	378
942764	Ono	Japan	915

To determine the two-way pairings of employees living in the same country, you would submit the following self-join request:

```
SELECT e.emp_num, e.name, f.emp_num, f.name, e.country
FROM employee AS e, employee AS f
WHERE e.country = f.country
AND e.emp_num < f.emp_num
ORDER BY e.emp_num, f.emp_num;
```

By specifying two different correlation names for employee, you are able to join the table to itself. Note that the condition `e.country = f.country` pairs only those employees who live in the same country, while the condition `e.emp_num < f.emp_num` pairs only employees who have different employee numbers.

The request produces the following result table:

employee after self-join on country				
emp_num	name	emp_num	name	country
113722	Lopes	225085	Ghazal	United States
113722	Lopes	577321	Korlapati	United States
225085	Ghazal	577321	Korlapati	United States
783004	Ramesh	799106	Manjula	India

Join Optimizations

Large Table/Small Table Joins

Definition of Large Table/Small Table Joins

A *large table/small table join* (LT/ST join) joins 3 or more small tables (or relations), then joins the result with 1 large table. This is the type of join typically made between a large fact table and its smaller dimensional tables in a dimensional modeling schema. In some cases, this can be further optimized to join the small tables (or relations) to the large table as a single-step, *n*-way join.

The LT/ST algorithm used by the Optimizer performs the following high-level tasks:

- Looks for the large relation in the set of tables to be joined
- Analyzes connections to each index
- Analyzes unindexed cases

About Large Table/Small Table Joins

It is important to collect statistics on the following items to optimize LT/ST joins:

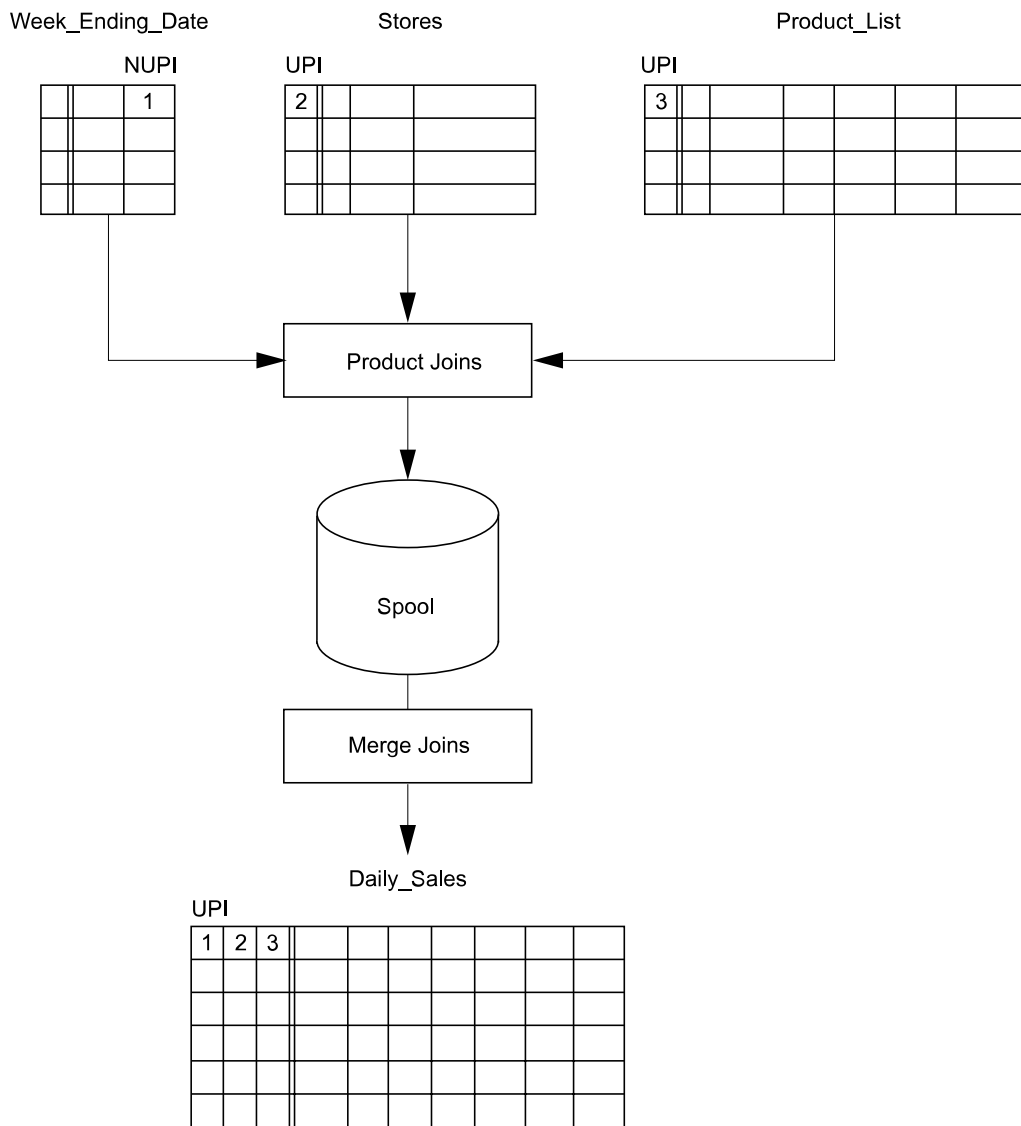
- All indexes
- Small table primary [AMP] indexes
- Columns to be selected, especially if the join is highly selective
- Join columns, especially if the join to the large table is weakly selective

Consider the following points about LT/ST joins and indexes:

- Indexes are an important factor in join performance.
- Consider the choice of indexes.
- Consider indexes on common-join column sets in large tables.

If the primary index of a large, or fact, table is a composite of elements from the smaller, or dimension, tables, as is generally the case for dimensional modeling where the primary index of the fact table is a composite of the primary indexes of its associated dimension tables, the Optimizer uses a product join on the small tables. With the primary index of the large table, the Optimizer can apply a merge join and not read the entire large table, which is much more efficient use of system resources.

For example, suppose you want to examine the sales of 5 products at 5 stores for a one-week time period. This requires joining the *stores* table, the *week_ending_date* table, and the *product_list* table with the *daily_sales* table. The following graphic illustrates this join:



Selected portions of the *stores* table, *week_ending_date* table and *product_list* table are product-joined. The result creates the primary index for the *daily_sales* table. The joined small tables are then joined with the large table, and an answer set is returned. This plan uses significantly fewer system resources and requires less processing time.

Star and Snowflake Join Optimization

About Star and Snowflake Join Optimizations

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.

Note:

The EXPLAIN text for queries involving star/snowflake or LT/ST joins does not explicitly mention them.

Star Joins

A star join schema is one in which a large table, called a fact table, is joined to a set of smaller tables, called dimension tables.

The fact table has a composite primary index. Each dimension table in a set related to a fact table has a simple primary index that corresponds exactly to one of the components of the composite primary index in the fact table.

Star join queries do not place selection criteria directly on a dimension table. Rather they place an IN condition on the primary index of the dimension table that is stored in the fact table. The IN list behaves as if it were a dimension table, thus allowing star join processing to occur in cases where normally the dimension table would have been required.

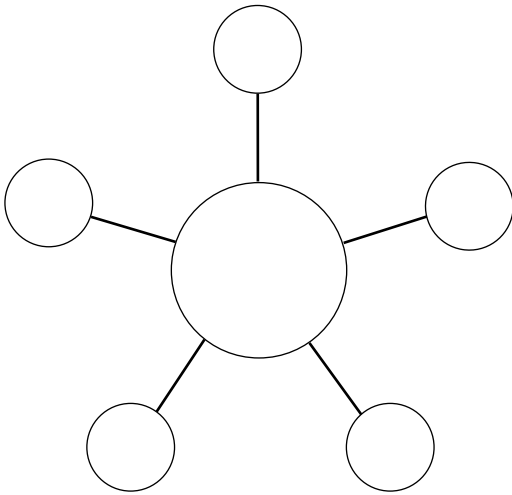
The Optimizer can apply star join processing to requests that join a subset of primary index/NUSI columns of a fact table to dimension tables and qualify the remaining primary index/NUSI columns with IN conditions.

A star join is a join where three or more relations with relatively small cardinality are joined to another relation having a much larger relative cardinality. In dimensional modeling terminology, the large table is called a fact table and the smaller tables are referred to as dimension tables.

The difference between a large relation and a small relation is defined in terms of cardinality ratios on the order of 100:1 at minimum. The smaller relations in a star join do not necessarily derive from base tables having a smaller cardinality; they can be an intermediate result of having applied a condition to a medium- or large-sized base table.

The term star derives from the pictorial representation of such a join, which resembles a childlike drawing of a star, as seen in the following illustration.

As you can see, the graphic representation of such a join resembles a crude drawing of a star.



For more information about star schemas and dimensional database models in physical database design, see *Teradata Vantage™ - Database Design*, B035-1094.

Primary and Secondary Targets of a Star Join

Star joins are a fundamental component of a dimensional database model. 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.

The star join method is useful in the following situations:

- Complex cases involving multiple large relations.
- A product join of several small relations in order to permit a merge join or a hash join with a locally spooled large relation when there is no available index.

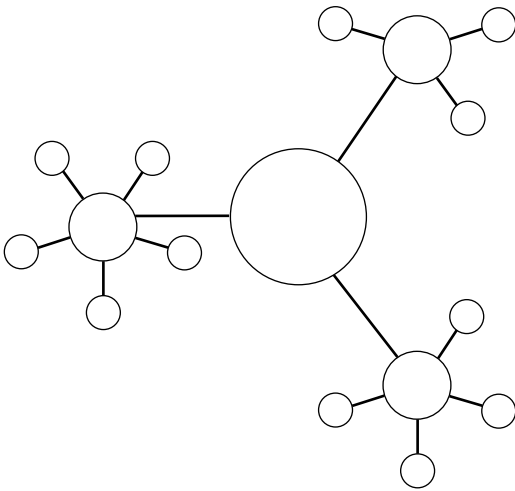
Snowflake Joins

A snowflake join is a join where a large table is joined with three or more smaller base tables or join relations, some or all of which themselves are joined to three or more smaller base tables or join relations. Another way of looking at a snowflake is to think of it as a star with normalized dimension tables.

The concept of snowflake schemas in physical database design is described in *Teradata Vantage™ - Database Design*, B035-1094.

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.

The graphic representation of such a join somewhat resembles a snowflake.



The concept of snowflake schemas in physical database design is described in *Teradata Vantage™ - Database Design*, B035-1094.

Star Join Optimization

With star join optimization, the Optimizer searches for a better join plan in which all the small tables are joined first, after which the resulting relation is joined with the large table. The Optimizer then uses the join plan that has the lowest estimated cost.

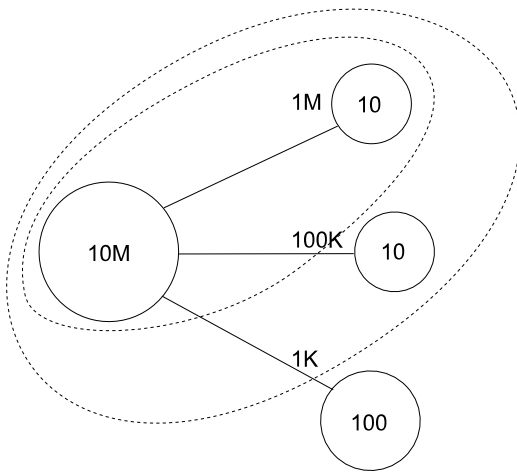
Without star join optimization, the Optimizer does an adequate job joining one or two small tables to a large table. However, when joining three or more small tables to one large table, the Optimizer usually generates a join plan in which a small table (or the join result of small tables) is joined directly with the large table.

When one or more IN conditions are specified on the large table, the Optimizer might choose to combine the IN lists with the small tables first. The query plan would then join the resulting join relation with the large table. The result is a star join plan with more dimension tables than the number of dimension tables explicitly specified in the query (see stage 6 in [Determining the Order of Joins](#)).

Example of Star Join Optimization

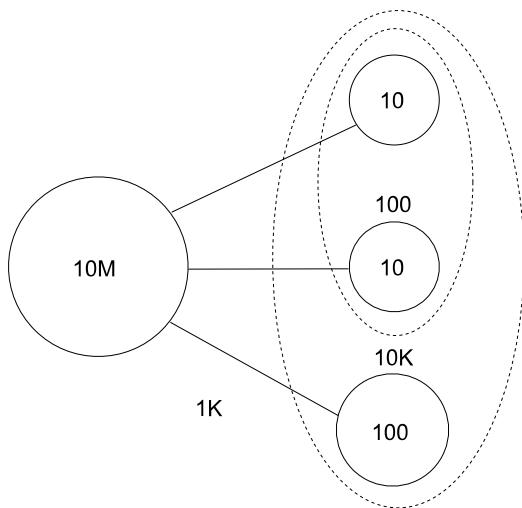
The following graphic illustrates a non-optimal star join plan of four tables. The relative cardinalities of the tables are given as integers within each table (represented in each case by a circle). The relative cost of each join is given as an integer number on the line connecting the joined relations.

In this example, the first join is between the large table and one of the small tables. The relative cost of this join is 1×10^6 . The next join is between this joined relation and another of the small tables. Its relative cost is 1×10^5 . Finally, this relation is joined with the last small table at a relative cost of 1×10^3 .



The next graphic presents an optimized join plan for the same set of tables. This plan uses a compromise join of all the unconnected (small) tables prior to making the join to the large table.

The first join has a relative cost of 1×10^2 , the second a cost of 1×10^4 , and the final join between the large table and its joined small tables relation has a cost of 1×10^3 .



Cost of non-optimized star join = $1 \times 10^6 + 1 \times 10^5 + 1 \times 10^3 = 1,101,000$

Cost of optimized star join = $1 \times 10^2 + 1 \times 10^4 + 1 \times 10^3 = 11,100$

The results indicate that for this example, the optimized star join plan is 2 orders of magnitude cheaper than the non-optimized star join plan for the same 4 tables.

Star Join Categories

Star joins are also referred to as large-table/small-table (LT/ST) joins. There are two basic types of LT/ST joins:

- [LT/ST-J1 Indexed Joins](#)
- [LT-ST-J2 Unindexed Joins](#)

LT/ST-J1 Indexed Joins

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

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.

In special cases, the sum of the sizes of the small tables can fit in memory, so the Optimizer can use a single-step, *n*-way join.

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

Star Join Planning and 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 [AMP] indexes, as well as for each index used in the query.
- If constraints are specified on unindexed columns, statistics must be collected on these columns as well.

Avoiding Hash Synonyms for Star Joins

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 columns are from the same domain (of the same data type), and if numeric, of the same size. If the joined columns 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 columns, 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 columns of the index are of smaller size, then one of the tables might 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 an operand. Consider modifying the equality to isolate the index on one side, exclusive of any expressions.

For example, consider the following conditional expressions.

The condition is stated in the following example using the primary index of table_2, which is table_2.NUPI, in an expression:

```
table_1.x = table_2.NUPI - 1
```

This is a suboptimal way to specify the condition.

The following specification of the condition is optimal because it separates the primary index of table_2 from the expression, moving it to the other side of the equality condition using simple algebraic addition of the same term (+1) to both sides of the equation.

```
table_1.x + 1 = table_2.NUPI
```

Selecting Indexes for Star Joins

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 greatly improved if you create an index 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).

Star Join 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.

Star Join Criteria When Any Join Column Can Be the Primary Index

If any of the collections of join columns meets the criteria for a good candidate primary index (that is, has enough unique values to guarantee that the large table is distributed evenly across the AMPs), then you should consider the following guidelines:

IF LT/ST joins are ...	THEN the primary index should be created using the ...
used with the same frequency	star join that results in the most number of rows selected. This leaves the star joins that select fewer rows for the NUSIs and USIs.
not used with the same frequency	collection of join columns associated with the most often used star join.

For example, if the star join between Widgets and (period, state, country) is used more often than the star join between Widgets and (color, shape, size), the primary index should be created on (period, state, country).

However, if the former join selects far fewer number of rows than the latter join, it may be better to associate the primary index with the latter join (on columns color, shape, and size).

Performance Modeling: Optimizing All Join Queries

To optimize the entire mix of join queries, you should design a more complete performance model for your database.

For example, a user might have a relatively short star join query that is used more frequently than an extremely long query.

In such cases, it might be better to select the primary index favoring the long query, even though the guidelines indicate otherwise. This is because the benefit of the long query may be very great compared to the cost of the short query, and the combination of joins results in a net gain in performance.

Not all join columns of the small tables must join with the large table index in an LT/ST-J1a (Reasonable Indexed) star join.

Using a Common Set of Star 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 the following 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 be defined on the columns (shape, size) if that column set is qualified to be the primary index for widgets.

Star Join Examples

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:

Dimension (Small) Table Definitions		
Table name	Columns	Primary index
color	code, description	description
size	code, description	description
options	code, description	description

Fact (Large) Table Definition		
Table name	Columns	Cardinality
widgets	color, size, options, units, period	1,000,000 rows

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 dimension tables:

Attribute	Cardinality
color	2
size	10
options	10

Large Table Uniqueness Statistics

The following statistics information is given for the large fact 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

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	duplicated options, direct size	2.67
Spool 5: Product Join	duplicated color, direct widgets	6 660.00 This equates to 1 hour, 51 minutes.
Spool 1: Merge Join	duplicated 4, local 5	7.43

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 shows part of the EXPLAIN output that 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.

- 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.

Reasonable Indexed Join Plan With Star Join Optimization

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	duplicated color, direct options	0.31
Spool 5: Product Join	duplicated 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

Part of the EXPLAIN output for this optimized join plan is shown below.

- 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.

Reasonable Indexed Join Plan With Star Join Optimization and a Fact Table USI

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	duplicated color, direct options	0.31
Spool 5: Product Join	duplicated 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

Part of the EXPLAIN output for this optimized join plan is shown below.

- 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.

Reasonable Indexed Join Plan With Star Join Optimization and a Fact Table NUSI

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	duplicated color, direct options	0.31
Spool 5: Product Join	duplicated size, direct 3	4.43
Spool 1: Nested Join	duplicated 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

Part of the EXPLAIN output for this optimized join plan is shown below.

- 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.

Join Plan With Star Join Optimization and Fact Table Subquery Join

Query Used for a Join Plan With Star Join Optimization and Fact Table Subquery Join

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	duplicated color, direct size	0.31
Spool 6: Product Join	local 4, duplicated options	4.46
Spool 1: Nested Join	duplicated 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

Part of the EXPLAIN output for this optimized join plan is shown below.

- 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.

Cardinality and Uniqueness Statistics for the Reasonable Unindexed Join Examples

Dimension 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

Fact 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

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	duplicated color, direct widgets	12 491.00 This equates to 3 hours, 28 minutes, 11 seconds.
Spool 1: Merge Join	duplicated 3, local 4	21.05

Completion Time

The total estimated completion time is 3 hours 28 minutes.

EXPLAIN Output for Unoptimized Join Plan

Part of the EXPLAIN output, generated without LT/ST optimization, is shown below.

- 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.

Reasonable Unindexed Join With Join Optimization

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, duplicated 2	1.24
Spool 1: Merge Join	local widgets, duplicated 3	7,761.00 This equates to 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

Part of the generated EXPLAIN output is shown below.

- 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.

Optimization Using Join Indexes

Join Indexes

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 *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

A Simple Join Query

The following is an example of a simple join query:

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

Part of the EXPLAIN output is shown below.

```
4) We do an all-AMPS JOIN step in TD_MAP1 from DB1.ordertbl by way
of a RowHash match scan with no residual conditions, which is
joined to DB1.lineitem by way of a RowHash match scan with no
residual conditions. DB1.ordertbl and DB1.lineitem are joined
using a sliding-window merge join, with a join condition of (
"DB1.lineitem.l_orderkey = DB1.ordertbl.o_orderkey"). The
result goes into Spool 1 (group_amps), which is built locally on
the AMPS. The size of Spool 1 is estimated with low confidence to
be 11 rows (869 bytes). The estimated time for this step is 0.21
seconds.
```

A Search on a Join Index

The following is an example of a search condition on the join index:

```
EXPLAIN
SELECT o_orderdate, o_custkey, l_partkey, l_quantity,
       l_extendedprice
FROM lineitem, ordertbl
```

```
WHERE l_orderkey=o_orderkey
AND   o_orderdate>'1997-11-01';
```

Part of the EXPLAIN output is shown below.

- 4) We do an all-AMPs JOIN step in TD_MAP1 from 500 partitions of DB1.ordertbl by way of a RowHash match scan with a condition of ("DB1.ordertbl.o_orderdate > DATE '1997-11-01'"), which is joined to DB1.lineitem by way of a RowHash match scan with no residual conditions. DB1.ordertbl and DB1.lineitem are joined using a sliding-window merge join, with a join condition of ("DB1.lineitem.l_orderkey = DB1.ordertbl.o_orderkey"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 2 rows (158 bytes). The estimated time for this step is 0.21 seconds.

Aggregation on a Join Index

The following is an example of an aggregation on a join index.

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

Part of the EXPLAIN output is shown below.

- 4) We do an all-AMPs JOIN step in TD_MAP1 from DB1.ordertbl by way of a RowHash match scan with no residual conditions, which is joined to DB1.lineitem by way of a RowHash match scan with no residual conditions. DB1.ordertbl and DB1.lineitem are joined using a sliding-window merge join, with a join condition of ("DB1.lineitem.l_orderkey = DB1.ordertbl.o_orderkey"). The result goes into Spool 2 (all_amps), which is built locally on the AMPs. The size of Spool 2 is estimated with low confidence to be 11 rows (341 bytes). The estimated time for this step is 0.21 seconds.
- 5) We do an all-AMPs SUM step in TD_Map1 to aggregate from Spool 2 (Last Use) by way of an all-rows scan, grouping by field1 (DB1.lineitem.l_partkey). Aggregate Intermediate Results are computed globally, then placed in Spool 4 in TD_Map1. The size of

Spool 4 is estimated with no confidence to be 9 rows (369 bytes).
The estimated time for this step is 0.23 seconds.

- 6) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 4 (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 no confidence to be 9 rows (522 bytes). The estimated time for this step is 0.16 seconds.

Join Index Used to Join With Another Base Table

The following is an example of a join index used to join with another base table:

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

Part of the EXPLAIN output is shown below.

- 5) We do an all-AMPs JOIN step in TD_MAP1 from DB1.ordertbl by way of a RowHash match scan with no residual conditions, which is joined to DB1.lineitem by way of a RowHash match scan with no residual conditions. DB1.ordertbl and DB1.lineitem are joined using a sliding-window merge join, with a join condition of ("DB1.lineitem.l_orderkey = DB1.ordertbl.o_orderkey"). The result goes into Spool 2 (all_amps), which is redistributed by the hash code of (DB1.ordertbl.o_custkey) to all AMPs in TD_Map1. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated with low confidence to be 11 rows (407 bytes). The estimated time for this step is 0.11 seconds.
- 6) We do an all-AMPs JOIN step in TD_MAP1 from DB1.customer by way of a RowHash match scan with a condition of ("(DB1.customer.c_custkey <= 49999) AND (DB1.customer.c_custkey >= 0)"), which is joined to Spool 2 (Last Use) by way of a RowHash match scan. DB1.customer and Spool 2 are joined using a merge join, with a join condition of ("o_custkey = DB1.customer.c_custkey"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 11 rows (1,188 bytes). The estimated time for this step is 0.21 seconds.

Join Index Used to Resolve Single-Table Query

The following is an example of a join index used to resolve single table query:

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

Part of the EXPLAIN output is shown below.

```
3) We do an all-AMPs RETRIEVE step in TD_MAP1 from DB1.ORDERJOINLINE
   by way of an all-rows scan with a condition of (
   "DB1.ORDERJOINLINE.l_partkey = 1001") 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 (69 bytes). The
   estimated time for this step is 0.15 seconds.
```

Creating and Using a Secondary Index on a Join Index

The following is an example creating and using secondary index on a join index:

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

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

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

Part of the EXPLAIN output is shown below.

```
3) We do an all-AMPs RETRIEVE step in TD_MAP1 from DB1.ORDERJOINLINE
   by way of an all-rows scan with a condition of ("(NOT
   (DB1.ORDERJOINLINE.o_orderdate IS NULL )) AND
   (DB1.ORDERJOINLINE.l_shipdate = DATE '1997-09-18')") into Spool 1
   (group_amps), which is built locally on the AMPs. The size of
   Spool 1 is estimated with no confidence to be 8 rows (632 bytes).
   The estimated time for this step is 0.15 seconds.
```

Join Index Left Outer Joined on 6 Tables

The following join index definition left outer joins table t1 with, in succession, tables t2, t3, t4, t5, and t6 on a series of equality conditions made on foreign key-primary key relationships among the underlying base tables:

```
CREATE JOIN INDEX jiout AS
SELECT a1,b1,c1,c2,d1,d3,e1,e4,f1,g1,h1,i1,j1,j5,k1,k6, x1
FROM t1
LEFT OUTER JOIN t2 ON  a1=a2
                      AND b1=b2
                      AND c1=c2
LEFT OUTER JOIN t3 ON  d1=d3
LEFT OUTER JOIN t4 ON  e1=e4
                      AND f1=f4
LEFT OUTER JOIN t5 ON  g1=g5
                      AND h1=h5
                      AND i1=i5
                      AND j1=j5
LEFT OUTER JOIN t6 ON  k1=k6;
```

Even though the following query references fewer tables than are defined in the join index, you would expect the Optimizer to include join index `ji_out` in its access plan because all the extra outer joins are defined on unique columns and the extra tables are the inner tables in the outer joins. A covering join index whose definition includes one or more tables that is not specified in the query it covers is referred to as a broad join index. A wide range of queries can make use of a broad join index, especially when there are foreign key-primary key relationships defined between the fact table and the dimension tables that enable the index to be used to cover queries over a subset of dimension tables.

The bold EXPLAIN report text indicates that the Optimizer does select `ji_out` for the query plan. This is an example of a broad join index being used to cover a query.

```
EXPLAIN SELECT a1, b1, c1, SUM(x1)
FROM t1,t2
WHERE a1=a2
AND   b1=b2
AND   c1=c2
GROUP BY 1, 2, 3;
```

Part of the EXPLAIN output is shown below.

```
3) We do an all-AMPs SUM step to aggregate from HONG_JI.jiout
   by way of an all-rows scan with no residual conditions, and
   the grouping identifier in field 1. Aggregate Intermediate
```

Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 2 rows. The estimated time for this step is 0.03 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 2 rows. The estimated time for this step is 0.04 seconds.

Many More Tables Referenced by a Join Index Definition Than Referenced by Query

The following join index definition specifies all inner joins on tables t1, t2, t3, t4, t5 and t6 and specifies equality conditions on all the foreign key-primary key relationships among those tables:

```
CREATE JOIN INDEX ji_in AS
SELECT a1,b1,c1,c2,d1,d3,e1,e4,f1,g1,g5,h1,i1,j1,k1,k6, x1
FROM t1,t2,t3,t4,t5,t6
WHERE a1=a2
AND    b1=b2
AND    c1=c2
AND    d1=d3
AND    e1=e4
AND    f1=f4
AND    g1=g5
AND    h1=h5
AND    i1=i5
AND    j1=j5
AND    k1=k6;
```

Even though 6 tables are referenced in the join index definition, and all its join conditions are inner joins, you would expect the Optimizer to include join index ji_in in its query plan for the following query, which only references 2 of the 6 tables, because all the conditions in the join index definition are based on foreign key-primary key relationships among the underlying base tables. This is also an example of a broad join index being used to cover a query.

The bold EXPLAIN report text indicates that the Optimizer does select ji_in for the query plan.

```
EXPLAIN SELECT a1,b1,c1,SUM(x1)
FROM t1,t2
WHERE a1=a2
AND    b1=b2
AND    c1=c2
GROUP BY 1, 2, 3;
```


Part of the EXPLAIN output is shown below.

- 3) We do an all-AMPs SUM step to aggregate from **HONG_JI.ji_in** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 2 rows. The estimated time for this step is 0.03 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 2 rows. The estimated time for this step is 0.04 seconds.

Using a Join Index Defined With a 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;
```

Part of the EXPLAIN output is shown below.

- 3) We do an all-AMPs RETRIEVE step from join index table df2.CustOrderJoinLine by way of an all-rows scan with a condition of ("df2.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.

Maintaining a Join Index for DELETE, INSERT, and UPDATE Operations

As with other indexes (for example, secondary indexes), join indexes are automatically maintained by the system when DELETE, INSERT, or UPDATE requests are issued against the underlying base tables of a join index.

Overhead Costs of Maintaining a Join Index

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. 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 request.

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	<p>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).</p> <p>Join indexes defined with outer joins usually require additional steps to maintain unmatched rows.</p>

As with secondary indexes, updates can cause a physical row of a compressed join index to split into multiple rows. Each newly formed row has the same fixed column value but contains a different list of repeated column values.

The system does not automatically recombine such rows, so the compressed 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 a Join Index During Simple DELETE Operations

Example of a General Method for Maintaining a Join Index During Simple DELETE Operations

The following is an example of a general case method for maintaining a join index during a simple DELETE request.

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 nonmatching outer join rows into the join index.

```
EXPLAIN
DELETE FROM ordertbl
WHERE o_custkey = 1001;
```

Part of the EXPLAIN output is shown below.

5) We do an all-AMPs JOIN step in TD_MAP1 from DB1.lineitem by way of a RowHash match scan with no residual conditions, which is joined to 84 partitions of DB1.Ordertbl by way of a RowHash match scan with a condition of ("DB1.ordertbl.o_custkey = 1001"). DB1.lineitem and DB1.Ordertbl are joined using a sliding-window merge join, with a join condition of ("DB1.lineitem.l_orderkey = DB1.Ordertbl.o_orderkey"). The result goes into Spool 1 (all_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the hash code of (DB1.Ordertbl.o_orderdate).

The size of Spool 1 is estimated with low confidence to be 2 rows (106 bytes). The estimated time for this step is 0.17 seconds.

- 6) We execute the following steps in parallel.
 - 1) We do an all-AMPs MERGE DELETE to DB1.ORDERJOINLINE from Spool 1. The size is estimated with low confidence to be 2 rows. The estimated time for this step is 4.60 seconds.
 - 2) We do an all-AMPs DELETE step in TD_MAP1 from 84 partitions of DB1.ordertbl with a condition of (
"DB1.ordertbl.o_custkey = 1001"). The size is estimated with low confidence to be 1 row. The estimated time for this step is 0.05 seconds.
 - 7) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 1 (Last Use) by way of an all-rows scan into Spool 2 (all_amps), which is redistributed by the hash code of (DB1.lineitem.l_orderkey) to all AMPs in TD_Map1. Then we do a SORT to order Spool 2 by join index. The size of Spool 2 is estimated with low confidence to be 2 rows (106 bytes). The estimated time for this step is 0.16 seconds.
 - 8) We do an all-AMPs MERGE step in TD_MAP1 into DB1.ORDERJOINLINE from Spool 2 (Last Use). The size is estimated with low confidence to be 2 rows. The estimated time for this step is 1.34 seconds.
 - 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.

General Methods of Maintaining a Join Index During Joined DELETE Operations

This topic describes some general methods by which Vantage maintains join indexes during join delete operations on a base table.

About Join Index Maintenance for Joined DELETE Operations

A join delete is done to a base table when a FROM clause specifies more than one table with some join conditions such as the following generic example of a deletion from a base table on which a join index is defined.

```
DELETE delete_table_name
FROM join_table_name_1, join_table_name_n
WHERE condition;
```

Join Delete From a Primary-Indexed Table With a Primary-Indexed Join Index

A join delete happens when the deletion is from a table using a spool of qualified rows. Vantage uses the following plan for a join delete to a primary-indexed base table with a primary-indexed join index.

- 6) We do an all-AMPs JOIN step (No Sum) from Spool 2 (Last Use) by way of a RowHash match scan, which is joined to Spool 3 (Last Use) by way of a RowHash match scan. Spool 2 and Spool 3 are joined using a merge join, with a join condition of ("Spool_3.col2 = Spool_2.y1"). The result goes into Spool 1 (all_amps), which is redistributed by hash code to all AMPs with hash fields ("Spool_3.Field_1025"). Then we do a SORT to order Spool 1 by row hash. The size of Spool 1 is estimated with no confidence to be 2 rows (50 bytes). Spool Asgnlist:
 "Field_1025" = "Spool_3.Field_1025",
 "Field_1026" = "{RightTable}.Field_1026",
 "Field_1027" = "{RightTable}.Field_1027".
 The estimated time for this step is 0.03 seconds.
- 7) We do an all-AMPs MERGE DELETE to DRCP_DELETE.t_pi from Spool 1. The size is estimated with no confidence to be 2 rows. The estimated time for this step is 0.76 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 with hash fields ("Spool_1.col3"). 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 (42 bytes). Spool Asgnlist: "col2",
 "Spool_1.col3".
- 9) We do an all-AMPs MERGE DELETE to drcp_delete.NONCPJI from Spool 4 (Last Use). The size is estimated with no confidence to be 2 rows. The estimated time for this step is 0.74 seconds.

Join Delete From a Column-Partitioned NoPI Table With a Primary-Indexed Join Index

For a column-partitioned NoPI base table with a primary-indexed join index, it could be very costly to retrieve all the columns from the column-partitioned table as was demonstrated in step 6 of the previous example, so in this case only the row ID of the column-partitioned table is spooled, and it is then used for the merge delete operation on the column-partitioned table.

Vantage can then join this row ID spool with the column-partitioned table to provide the qualified join index rows that are used to make the merge delete operation on the primary-indexed join index, as the following EXPLAIN text indicates in step 8.

- 6) We do an all-AMPs JOIN step (No Sum) from Spool 2 (Last Use) by way of a RowHash match scan, which is joined to Spool 3 (Last Use) by way of a RowHash match scan. Spool 2 and Spool 3 are joined using a merge join, with a join condition of ("Spool_3.cpcol2 = Spool_2.y1"). The result goes into Spool 1 (all_amps), which is redistributed by hash code to all AMPs with hash fields ("Spool_3.Field_1"). Then we do a SORT to order Spool 1 by row hash. The size of Spool 1 is estimated with no confidence to be 2 rows (50 bytes). Spool Asgnlist:
 "Field_1" = "Spool_3.Field_1"
 The estimated time for this step is 0.03 seconds.
- 7) We do an all-AMPs JOIN step (No Sum) from Spool 1 (Last Use), which is joined DRCP_DELETE.cpt. Spool 1 and DRCP_DELETE.cpt are joined using a rowid join, with a join condition of ("Spool_1.Field_1 = cpt.ROWID"). The result goes into Spool X (all_amps), which is redistributed by hash code to all AMPs with hash fields ("Spool_X.cpcol3"). Then we do a SORT to order Spool X by row hash. The size of Spool 1 is estimated with no confidence to be 2 rows (50 bytes). Spool Asgnlist:
 "Spool_x.col2",
 "Spool_X.col3".
 The estimated time for this step is 0.03 seconds.
- 8) **We do an all-AMPs MERGE DELETE to DRCP_DELETE.cpt from Spool 1 via the rowid.** The size is estimated with no confidence to be 2 rows.
 The estimated time for this step is 0.76 seconds.
- 9) We do an all-AMPs MERGE DELETE to drcp_delete.NONCPJI from Spool X (Last Use). The size is estimated with no confidence to be 2 rows.
 The estimated time for this step is 0.74 seconds.

Join Delete From a Column-Partitioned NoPI Table With a Column-Partitioned NoPI Join Index

The join delete happens when the deletion is from a table using a spool of qualified rows. For this example, Vantage only needs to add the row ID values from the base table to the qualified row spool (Spool 4) in step 7 (boldface text) rather than the rows themselves to delete the qualified rows. The same join index spool is used to do a merge delete operation on the column-partitioned join index as the following EXPLAIN text demonstrates.

- 6) We do an all-AMPs JOIN step (No Sum) from Spool 2 (Last Use) by way of a RowHash match scan, which is joined to Spool 3 (Last Use) by way of a RowHash match scan. Spool 2 and Spool 3 are joined using a merge join, with a join condition of ("Spool_3.cpcol2 = Spool_2.y1"). The result goes into Spool 1 (all_amps), which is redistributed by hash code to all AMPs with hash fields (

"Spool_3.Field_1") and Field1 ("Spool_3.Field_1"). Then we do a SORT to order Spool 1 by row hash and the sort key in spool field1 eliminating duplicate rows. The size of Spool 1 is estimated with no confidence to be 4 rows (72 bytes). Spool Asgnlist:

"Field_1" = "Spool_3.Field_1".

The estimated time for this step is 0.13 seconds.

- 7) We do an all-AMPs MERGE DELETE to DRCP_DELETE.cpt from Spool 1 (Last Use) via the row id. **Also, deleted rows are inserted in Spool 4.** The size is estimated with no confidence to be 4 rows. The estimated time for this step is 15.04 seconds.
- 8) We do an all-AMPs RETRIEVE step from Spool 4 (Last Use) by way of an all-rows scan into Spool 5 (all_amps), which is redistributed by hash code to all AMPs with hash fields ("Spool_4.ROWID"). Then we do a SORT to order Spool 5 by row hash. The size of Spool 5 is estimated with no confidence to be 4 rows (84 bytes). Spool Asgnlist: "Spool_4.ROWID"
The estimated time for this step is 0.10 seconds.
- 9) We do an all-AMPs MERGE DELETE to drcp_delete.CPJ1 from Spool 5 (Last Use) using FTS_RowIDMatch. The size is estimated with no confidence to be 4 rows.
The estimated time for this step is 10.53 seconds.

Optimized Method of Maintaining a Join Index During DELETE Operations

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 request 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 Delete 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_column_2* is defined, no modifications to *join_index_column_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 of an Optimized Method for Maintaining a Join Index During DELETE Operations

The following is an example of an optimized method for maintaining join index during a DELETE request:

```
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 df2.OrderJoinLine by way of the primary index "df2.OrderJoinLine.l_orderkey = 10" with a residual condition of ("df2.OrderJoinLine.l_orderkey = 10").
 - 2) We do a single-AMP DELETE from df2.lineitem by way of the primary index "df2.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.

General Method of Maintaining a Join Index During INSERT Operations

Example of a General Method for Maintaining a Join Index During INSERT Operations

The following is an example of maintaining join index during an INSERT request.

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;
```

Part of the EXPLAIN output is shown below.

- 6) We do an all-AMPs RETRIEVE step from df2.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 df2.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 df2.lineitem. Spool 3 and df2.lineitem are joined using a merge join, with a join condition of ("df2.lineitem.l_orderkey = Spool_3.o_orderkey"). The input table df2.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 df2.orderjoinline from Spool 2 (Last Use).
 - 12) We do a MERGE into df2.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 a Join Index During UPDATE Operations

Example of a General Method for Maintaining a Join Index During UPDATE Operations

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;

```

Part of the EXPLAIN output is shown below.

- 4) We do an all-AMPs RETRIEVE step in TD_MAP1 from DB1.ORDERJOINLINE by way of an all-rows scan with a condition of ("DB1.ORDERJOINLINE.l_partkey = 50") 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 (53 bytes). The estimated time for this step is 0.15 seconds.
- 5) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 1 by way of an all-rows scan into Spool 2 (all_amps), which is redistributed by the hash code of (DB1.ORDERJOINLINE.l_orderkey) to all AMPs in TD_Map1. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated with no confidence to be 1 row (53 bytes). The estimated time for this step is 0.07 seconds.
- 6) We execute the following steps in parallel.
 - 1) We do an all-AMPs MERGE DELETE to DB1.ORDERJOINLINE from Spool 2 (Last Use). The size is estimated with no confidence to be 1 row. The estimated time for this step is 4.60 seconds.
 - 2) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 1 (Last Use) by way of an all-rows scan into Spool 3 (all_amps), which is redistributed by the hash code of (DB1.ORDERJOINLINE.l_orderkey) to all AMPs in TD_Map1. Then we do a SORT to order Spool 3 by join index. The size of Spool 3 is estimated with no confidence to be 1 row (53 bytes). The estimated time for this step is 0.16 seconds.
- 7) We execute the following steps in parallel.
 - 1) We do an all-AMPs MERGE step in TD_MAP1 into DB1.ORDERJOINLINE from Spool 3 (Last Use). The size is

```

estimated with no confidence to be 1 row. The estimated time
for this step is 1.34 seconds.
2) We do an all-AMPs UPDATE step in TD_MAP1 from DB1.lineitem
by way of an all-rows scan with a condition of (
"DB1.lineitem.l_partkey = 50"). The size is estimated with
no confidence to be 2 rows. The estimated time for this step
is 0.10 seconds.
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 a Join Index During UPDATE Operations

Optimizations have been made for UPDATE requests that allow the affected join index rows to be located via direct access. For example, if an UPDATE request 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 Update 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_column_2* is defined, no modifications to *join_index_column_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 of an Optimal Method for Maintaining a Join Index During an UPDATE Request

The following is an example of an optimized method for maintaining join index during an UPDATE request:

```

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

```
df2.OrderJoinLine by way of the primary index
"df2.OrderJoinLine.l_orderkey = 10" with a residual
condition of ("df2.OrderJoinLine.l_orderkey = 10").
2) We do a single-AMP UPDATE from df2.lineitem by way of the
primary index "df2.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.
```

Interpreting EXPLAIN Output

EXPLAIN Request Modifier

When prepended to an SQL request, the EXPLAIN request modifier returns a summary of the static, step-by-step Optimizer plan for processing the SQL request. For the syntax, usage notes, and other operational aspects of the EXPLAIN request modifier, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

EXPLAIN is an extremely useful utility for query designers. The returned output lists the steps the Optimizer would take to process the request and the estimated time required to complete the request, given the statistics the Optimizer has to work with. The output shows which indexes the Optimizer would use to process the request, identifies any intermediate spool generated, indicates the types of join to be performed, shows whether the requests in a transaction would be dispatched in parallel, and includes other information specific to the request.

EXPLAIN can help you evaluate queries and develop alternative, more efficient, processing strategies. For requests qualifying for incremental planning and execution (IPE), DYNAMIC EXPLAIN provides a similar function.

Although the times reported in the EXPLAIN output are presented in units of seconds, they are actually arbitrary units of time, intended to allow you to compare the relative performance of alternate coding formulations of the same query.

The plan for a request as ultimately executed on the system may differ from the plan presented in the EXPLAIN output if system conditions have changed. For example, due to changes in system statistics and data demographics or due to dynamic planning for IPE, which can change plan steps based on the results of previous steps. You can use DBQL to log the EXPLAIN output associated with the actual execution of a query. For more information on DBQL, see *Teradata Vantage™ - Database Administration*, B035-1093.

Keep EXPLAIN results with your system documentation because they can be of value when you reevaluate your database design, and can help you identify and resolve issues that may appear after a system upgrade or migration.

EXPLAIN Request Modifier Processes SQL Requests Only

You can modify any valid Teradata SQL request with EXPLAIN. You cannot EXPLAIN a USING request modifier, another EXPLAIN request modifier, or individual functions, stored procedures, or methods.

Effect of Parameterized Data Parcel Values Cache Peeking on EXPLAIN Reports

If you specify USING data, or if you specify a DATE, CURRENT_DATE, CURRENT_TIMESTAMP, or USER built-in function in a request, or both, the system can invoke Request Cache peeking (see [Parameterized Requests](#)). When this occurs, the EXPLAIN text reports the peeked literal values.

If you do not specify USING data, or if the USING variables, DATE, CURRENT_DATE, CURRENT_TIMESTAMP, or USER values, or both are not peeked, then there is no impact on either the generated plan or the generated EXPLAIN text.

Note that parameterized requests specified without a USING request modifier, but using either CLlv2 data parcel flavor 3 (Data) or CLlv2 parcel flavor 71 (DataInfo), cannot be explained using any of the following request modifiers or statements:

- EXPLAIN
- DUMP EXPLAIN
- INSERT EXPLAIN

See *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417 or *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418 for details about data parcel flavors.

EXPLAIN Text and Conditional Expressions

Conditional expressions are enclosed in APOSTROPHE characters (U+0027). The EXPLAIN text only includes the first 255 characters of conditional expressions.

EXPLAIN Text and Session Character Sets

The database returns EXPLAIN text literals and object names that include characters not in the repertoire of the character set or that are otherwise unprintable as Unicode delimited literals or identifiers. The database only escapes those characters that are outside the repertoire to hexadecimal.

The rules for handling such characters and the escape character usage are as follows:

1. If BACKSLASH (U+005C) is available in the session character set, the database uses it as the escape character.
2. Otherwise, if the YEN sign (U+00A5) or the WON sign (U+20A9) is present in the session character set at 0x5C, the database uses it as the escape character.
3. If none of those conditions is true, the database uses NUMBER SIGN (U+0023) as the escape character. This character is required in all session character sets supported by Teradata.
4. Vantage adds double quotation marks around object names that are translatable, but not lexically distinguishable as names.

EXPLAIN and Teradata Viewpoint

EXPLAIN reports can be viewed in the Teradata Manager SQL Scratchpad portlet of Viewpoint. To view a summarized version of the EXPLAIN output for an executing request, use the Teradata Viewpoint Query Monitor portlet.

To view the EXPLAIN output for a request executed through Teradata Viewpoint, use the Teradata Viewpoint SQL Scratchpad portlet.

EXPLAIN Confidence Levels

When the Optimizer estimates relation and join cardinalities, it does so with a level of confidence in the accuracy of the estimate. 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 level, 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.

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 dynamic 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.

About 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

Vantage reports Index Join confidence only for join operations.

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. 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.

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. 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.

Understand that *no confidence* by itself does not indicate a plan is bad. Even with no confidence in some steps, the Optimizer can usually generate a reasonable plan, especially if you have collected appropriate statistics. If a step only references base tables, it can mean that you should consider collecting statistics on these tables.

Similarly, 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 an overview of confidence levels only. They are in no way meant to be comprehensive, and do not take into account any special cases.

In general, confidence levels are assigned to the cardinality estimates for only two types of operations:

- Single-table retrievals
- Joins

Note that the confidence levels the Optimizer reports and the confidence levels recorded in the Level attribute of the QCD StatsRec table are not related in any way.

Confidence Levels For Single-Table Retrieval Operations

The following table lists the meaning of each confidence level in context and some of the reasons why each confidence level is assigned for single-table retrieval operations:

Confidence Level	Meaning	Reason
No	Vantage has neither Low nor High confidence in the cardinality and distinct value estimates for the relation. The Optimizer pursues conservative strategies to optimize the relevant steps.	<p>Any of the following situations exists:</p> <ul style="list-style-type: none"> • No statistics have been collected on the column or index sets specified in the predicate. • The predicate contains nondeterministic expressions for which base table statistics cannot be collected. For example, you cannot collect base table statistics for either of the following expressions: <ul style="list-style-type: none"> ◦ <code>udf_random(col1)</code> (UDF declared as nondeterministic) ◦ <code>random(col1)</code> <p>You can collect statistics on a single-table join index or hash index defined using a complex expression (see Using Join Index Statistics to Estimate Single-Table Expression Cardinalities), and the optimizer can use those statistics to make single-table cardinality estimates. If you have collected such statistics, then a cardinality estimate can be made with high confidence.</p> <ul style="list-style-type: none"> • For an aggregate estimation, no statistics have been collected on the grouping columns.
Low	Vantage is moderately certain that the estimated cardinality and distinct value estimates for the relation are accurate. The Optimizer pursues more	<p>One of the following states exists for the relation:</p> <ul style="list-style-type: none"> • There are conditions in the query on an index set for which no statistics have been collected. Cardinality estimates can be made based on sampling the index set. • There are conditions in the query on an index or column set with collected statistics that are ANDed with conditions on unindexed columns.

Confidence Level	Meaning	Reason
	aggressive strategies to optimize the relevant steps.	<ul style="list-style-type: none"> There are conditions in the query on an index or column set with collected statistics that are ORed with other conditions. For an aggregate estimation, there are statistics on single columns of the grouping column set or statistics on some, but not all, of the grouping columns. <p>The confidence for single-AMP dynamic AMP statistical samples is always Low.</p> <p>EXPLAIN reports always express No confidence in estimates where no statistics have been collected, but Vantage always samples dynamically from at least one AMP in such cases when the query is actually executed.</p>
High	Vantage is fairly certain that the estimated cardinality and distinct value estimates for the relation are accurate. The Optimizer pursues more aggressive strategies to optimize the relevant steps.	<ul style="list-style-type: none"> Retrieval from a single relation with no predicates: <ul style="list-style-type: none"> There are conditions in the query on the primary index and statistics have been collected on the primary index. There are conditions in the query on the primary index, but no statistics have been collected on the primary index. <p>The confidence is High under any of the following situations:</p> <ul style="list-style-type: none"> 5 or more AMPs are sampled dynamically. Rows per value are sampled using a dynamic AMP sample. No skew is detected. Retrieval from a single relation with predicates: <ul style="list-style-type: none"> Statistics have been collected on the predicate columns or indexes and there is no skew. There are multiple equality predicates on which covering multitable statistics have been collected. Single-table join index or hash index statistics have been collected on a complex expression specified in a predicate written against the underlying base table of the hash or single-table join index. Retrieval from a single temporal relation with predicates and a unique join index that can be used as an access path: <ul style="list-style-type: none"> Statistics have been collected on the join index that match the query predicate columns. For an aggregate estimation, the confidence is high under any of the following situations: <ul style="list-style-type: none"> The grouping columns are constants. The grouping columns have equality predicates. The grouping columns are all covered by a set of single multicolumn statistics. Statistics have been collected on the single grouping column.
Index Join	Not applicable	Applies only to joins.

For a retrieval operation from a spool, 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, such as join method and join geography, the join plan determines the best order for joining the relations. Because you can join as many as 128 tables and single-table views per join clause, it is critical to minimize join cardinality estimation errors to ensure that an optimal query plan is generated.

Keep in mind that errors in join processing are cumulative, so it is critical to minimize the possibilities for errors to occur in join planning. The only way to ensure optimal join processing is to keep fresh statistics on all your indexes and non-index join columns.

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. 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.

Confidence Level	Meaning	Reason
No	Vantage has neither Low nor High nor Index Join confidence in the estimated join cardinality.	One or both of the relations in the join does not have statistics on the join columns.
Low	Vantage is moderately certain that the estimated join cardinality is accurate.	<ul style="list-style-type: none"> • Statistics have been collected on the join columns of both the left and right relations. • One relation in the join has Low confidence and the other has any of the following confidence levels: <ul style="list-style-type: none"> ◦ Low ◦ High ◦ Index Join
High	Vantage 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 confidence levels: <ul style="list-style-type: none"> • High • Index Join
Index Join	Vantage is fairly certain that the estimated join cardinality is accurate because of a	<ul style="list-style-type: none"> • There is a unique index on the join columns. • There is a foreign key relationship between the two relations in the join.

Confidence Level	Meaning	Reason
	uniqueness constraint on the join columns.	Because of the way Vantage implements PRIMARY KEY and UNIQUE INDEX constraints, these are essentially two ways of saying the same thing.

Effect of Dynamic 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 dynamically selected AMP, the confidence level is always set to low.

EXPLAIN Request Modifier Phrase Terminology

Many of the terms used in EXPLAIN phrases are described in the following list. Self-explanatory phrases are not listed.

Phrase	Explanation
..*	
.	Vantage uses this string to signify that it has replaced values that have been masked from the explain output for a dynamic query plan because a DBS Control field has been set to mask the intermediate results to be inserted into the request during incremental planning.
m_1 column partitions of ...	
	<p>This phrase indicates that up to m_1 column partitions of the column-partitioned table or join index might need to be accessed. There are column partition contexts available for each of the m_1 column partitions.</p> <p>For this phrase to occur, the column-partitioned table or join index does not have row partitioning. $m_1 \geq 2$. One of the column partitions accessed might be the delete column partition.</p> <p>Not all of the m_1 column partitions might need to be accessed if no rows qualify.</p> <p>A using rowid Spool phrase might follow the table or join index name. In this case, m_1 is the number of column partitions accessed when using the rowID spool while m_2 in the using rowid Spool phrase is the number of column partitions accessed to build the rowID spool.</p> <p>This phrase is used in steps such as RETRIEVE, DELETE, JOIN, and MERGE that might read a column-partitioned source.</p> <p>For a delete operation, Vantage does not immediately reclaim storage for any deleted rows other than LOBs and column partitions with ROW format. A subsequent fast path deletion of all the rows of the table or join index reclaims the storage of all deleted rows in the table, including previously deleted rows.</p>

Phrase	Explanation
m_1 column partitions (c_1 contexts) of ...	
	<p>This phrase indicates that up to m_1 column partitions of the column-partitioned table or join index might need to be accessed using c_1 column partition contexts.</p> <p>For this phrase to occur, the column-partitioned table or join index does not have row partitioning.</p> <p>$m_1 \geq 2$.</p> <p>$2 < c_1 < m_1 - 1$</p> <p>c_1 is equal to, or one less than, the number of available column partition contexts.</p> <p>One of the column partitions accessed might be the delete column partition. Not all of the m_1 column partitions might need to be accessed if no rows qualify.</p> <p>A using CP merge Spool or using covering CP merge Spool phrase follows the table or join index name.</p> <p>This phrase is used in steps such as RETRIEVE, DELETE, JOIN, and MERGE that might read a column-partitioned source.</p> <p>To access the m_1 column partitions, columns from the column partitions are merged up to c_1 column partitions at a time into a column-partitioned spool until all the projected columns have been retrieved for the qualifying rows.</p> <p>One or more of the merges might require accessing the delete column partition. For each such merge, the delete column partition is included in m_1.</p> <p>Performance can degrade if c_1 is much less than m_1 and the retrieve is not very selective. In this case, consider decreasing the number of column partitions that need to be accessed, decreasing the data block size for the table, combining column partitions so there are fewer column partitions, or increasing PPICacheThrP if there is enough available memory to do so.</p> <p>For a delete operation, Vantage does not immediately reclaim storage for any deleted rows other than LOBs and column partitions with ROW format. A subsequent fast path deletion of all rows of the table or join index reclaims the storage of all deleted rows in the table, including previously deleted rows.</p>
n partitions of ...	
	<p>This phrase indicates that only n of the combined partitions are accessed. n is greater than one. (all_amps</p> <p>This phrase only applies to a row-partitioned, primary-indexed (RPPI) table or join index.</p>
n_1 combined partitions (one column partition) of ...	
	<p>This phrase indicates that one column partition of multiple row partitions of the column-partitioned table or join index might need to be accessed. For this phrase to occur, the table or join index is accessed via rowIDs.</p> <p>For this phrase to occur, the table or join index is accessed via rowids.</p> <p>A using rowid Spool phrase might follow the table or join index name. In this case, one column partition is accessed when using the rowID spool while m_2 in the using rowid Spool phrase is the number of column partitions accessed to build the rowID spool.</p> <p>The phrase is used in steps such as RETRIEVE, DELETE, JOIN, and MERGE that might read a column-partitioned source.</p>
n_1 combined partitions (m_1 column partitions) of ...	
	<p>This phrase indicates that the rows and columns for up to m_1 combined partitions of the table or join index might need to be accessed.</p>

Phrase	Explanation
	<p>For the column-partitioning level, m_1 column partitions might need to be accessed. There are column partition contexts available for each of the m_1 column partitions.</p> <p>For this phrase to occur, the column-partitioned table or join index has both row and column partitioning. One of the column partitions that might be accessed is the delete column partition. Not all of the m_1 column partitions might need to be accessed if no rows qualify.</p> <p>A using rowid Spool phrase might follow the table or join index name. In this case, m_1 is the number of column partitions accessed when using the rowID spool while m_2 in the using rowid Spool phrase is the number of column partitions accessed to build the rowID spool.</p> <p>The phrase is used in steps such as RETRIEVE, DELETE, JOIN, and MERGE that might read a column-partitioned source.</p> <p>For a delete operation, Vantage does not immediately reclaim storage for any deleted rows other than LOBs and column partitions with ROW format. A subsequent fast path deletion of all the rows in a row partition reclaims the storage of all deleted rows in that row partition, including previously deleted rows in that partition. A subsequent fast path deletion of all the rows of the table or join index reclaims the storage of all deleted rows in the table, including previously deleted rows.</p>
n_1 combined partitions (m_1 column partitions and c_1 contexts) of ...	
	<p>This phrase indicates that the rows and columns for up to n_1 combined partitions of the table or join index might need to be accessed.</p> <p>For the column-partitioning level, m_1 column partitions might need to be accessed. For this phrase to occur, the column-partitioned table or join index has both row and column partitioning.</p> <p>For the column-partitioning level, c_1 column partition contexts are used to access up to m_1 column partitions. $n_1 \geq 2$.</p> <p>$m_1 \geq 2$.</p> <p>$2 < c_1 < m_1 - 1$ and is equal to or one less than the number of available column partition contexts.</p> <p>One of the column partitions that might be accessed is the delete column partition. Not all of the m_1 column partitions might need to be accessed if no rows qualify. A using CP merge Spool or using covering CP merge Spool phrase follows the table or join index name.</p> <p>The phrase is used in steps such as RETRIEVE, DELETE, JOIN, and MERGE that might read a column-partitioned source. To access the m_1 column partitions, columns from the column partitions are merged up to c_1 column partitions at a time into a column-partitioned spool until all the projected columns have been retrieved for the qualifying rows.</p> <p>One or more of the merges might require accessing the delete column partition. For each such merge, the delete column partition is included in m_1.</p> <p>Performance can degrade if c_1 is much less than m_1 and the retrieve is not very selective. In this case, consider decreasing the number of column partitions that need to be accessed, decreasing the data block size for the table, combining column partitions so there are fewer column partitions, or increasing PPICacheThrP if there is enough available memory to do so.</p> <p>For a delete operation, Vantage does not immediately reclaim storage for any deleted rows other than LOBs and column partitions with ROW format. A subsequent fast path deletion of all the rows in a row partition reclaims the storage of all deleted rows in that row partition, including previously deleted rows in that partition. A subsequent fast path delete of all the rows of the table or join index reclaims the storage of all deleted rows in the table, including previously deleted rows.</p>
aggregate results are computed globally ...	
	<p>This phrase indicates that totals are aggregated across all AMPs in an indicated map.</p> <p>For this case, Vantage performs all four steps of the ARSA aggregation algorithm.</p>

Phrase	Explanation
	<ol style="list-style-type: none"> 1. Aggregate locally on each AMP in the map. 2. Redistribute the local aggregations to their target AMPs. 3. Sort the redistributed aggregations. This stage of the process includes the elimination of duplicate rows. 4. Aggregate globally to compute the final aggregation.
aggregate results are computed locally ...	
	This phrase indicates that totals are aggregated locally on each AMP in an indicated map. For this case, local aggregation is the only stage of the ARSA aggregation algorithm that Vantage performs.
all partitions of ...	
	<p>This phrase indicates that Vantage accesses all combined partitions of an AMP for a primary index access to a single AMP for a primary-indexed row-partitioned table or join index.</p> <p>This phrase is not applicable to column-partitioned tables or join indexes because those objects do not have a primary index.</p>
a single column partition of ...	
	<p>This phrase indicates that only one column partition of the column-partitioned table or join index might need to be accessed. For this phrase to occur, the column-partitioned table or join index does not have row partitioning. Also, this phrase indicates the delete column partition does not need to be accessed either because the column partition has ROW format or access is via rowIDs from an index or rowID spool.</p> <p>A using rowid Spool phrase might follow the table or join index name. In this case, one column partition is accessed when using the rowID spool while m_2 in the using rowid Spool phrase is the number of column partitions accessed to build the rowID spool.</p> <p>This phrase is used in steps such as RETRIEVE, DELETE, JOIN, and MERGE that might read a column-partitioned source.</p> <p>For a delete operation, Vantage does not immediately reclaim storage for any deleted rows, other than LOBs and column partitions with ROW format. A subsequent fast path deletion of all of the rows of the table or join index reclaims the storage of all deleted rows in the table, including previously deleted rows.</p>
a single combined partition of ...	
	<p>This phrase indicates that only one column partition of only one row partition of the column-partitioned table or join index might need to be accessed.</p> <p>For this phrase to occur, the column-partitioned table or join index has to have both row and column partitioning. Also, this phrase indicates that the delete column partition does not need to be accessed either because the column partition has ROW format or because access is via rowids.</p> <p>A using rowid Spool phrase may follow the table or join index name. In this case, one column partition is accessed when using the rowID spool while m_2 in the using rowid Spool phrase is the number of column partitions accessed to build the rowID spool.</p> <p>This phrase is used in steps such as RETRIEVE, DELETE, JOIN, and MERGE that might read a column-partitioned source.</p>
a single partition of ...	

Phrase	Explanation
	<p>This phrase indicates that only a single combined partition of a primary-indexed row-partitioned table or join index is accessed.</p> <p>This phrase is not applicable to a column-partitioned table or join index.</p>
(all_amps) ...	
	<p>This phrase indicates that the spool is created on all AMPs in a map to which the step is sent. Because the spool is created on all AMPs, a dynamic AMP group is not needed.</p>
all-AMPs JOIN step in <i>mapname</i> by way of a RowHash match scan ...	
	<p>This phrase indicates the join step is processed on all AMPs in the indicated map. 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.</p> <p>Each row-hash match is located and processed as follows:</p> <ol style="list-style-type: none"> 1. 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. 2. 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. 3. After all rows with the same row-hash are processed from both tables, one more row is read from each table. The row-hashes from these two rows are compared, restarting the compare process.
all-AMPs RETRIEVE step in <i>mapname</i> by way of an all-rows scan ...	
	<p>This phrase indicates that all rows of a table are scanned row by row on all AMPs in the indicated map on which the table is stored.</p>
all partitions of ...	
	<p>This phrase indicates that all combined partitions of an AMP are accessed for a primary index access to a single AMP for a primary-indexed row-partitioned table or join index.</p>
a rowkey-based ...	
	<p>The join is hash-based by partition (that is, by the rowkey). In this case, there are equality constraints on all the partitioning columns and primary index columns. This allows for a faster join since each noneliminated row partition needs to be joined with at most only one other row partition. When the phrase is not given, the join is hash based. That is, there are equality constraints on the primary index columns from which the hash is derived.</p> <p>Note that with either method, the join conditions must still be validated.</p>
<BEGIN ROW TRIGGER LOOP>	
	<p>Processing of the trigger action statements defined in the row trigger starts from the current AMP step, step <i>n</i>, of the EXPLAIN text.</p> <p>All AMP steps from the current step through the step in which the phrase END ROW TRIGGER LOOP for step <i>n</i> appears constitute the row trigger loop.</p>
by skipping global aggregation	
	<p>When estimated to be cost-effective, Vantage avoids redistributing rows and aggregating them globally by skipping global aggregation.</p>

Phrase	Explanation
by skipping local aggregation	
	When estimated to be cost-effective, Vantage avoids the cost of first writing to spool and then sorting the spooled rows by skipping local aggregation.
by the hash code of ([database_name.]table_name.column_name).	
	The spool sort is done on the specified column in row hash order.
by using cache during local aggregation	
	When estimated to be cost-effective, Vantage avoids sorting large numbers of input rows by caching output rows during local aggregation and coalescing any duplicate rows in the cache.
by way of an all-rows scan	
	All rows in the table are scanned because a single-partition scan is not possible for the specified predicate conditions. See Single Partition Scans and BEGIN/END Bound Functions .
by way of index # <i>n</i> and the bit map in Spool <i>n</i> ...	
	The data row associated with the rowID is accessed only if the associated bit is turned on in the bit map.
by way of the primary AMP index ...	
	Indicates that there are conditions on the primary AMP index columns that allow the system to access only a single AMP for the indicated table.
by way of the primary index ...	
	Indicates that there are conditions on the primary index columns that allow the system to access only a single AMP for the indicated table. Primary indexes allow direct access to the rows with that primary index value in the table or in noneliminated combined row partitions.
by way of the sort key in spool field1 ...	
	Field1 is created to allow a tag sort.
by way of the unique primary index ...	
	Indicates that there are conditions on the unique primary index columns that allow the system to access only a single AMP and, at most, a single row for the indicated table.
(compressed columns allowed)	
	The target spool can have compressed values.
computed globally ...	
	The computation involves all the intermediate spool data based on compressed columns in the source data.
condition ...	
	An intermediate condition that join-qualifies table rows (as compared with the overall join condition).

Phrase	Explanation
duplicated on all AMPs in <i>mapname</i> ...	
	A spool containing intermediate data that is used to produce a result is copied to all AMPs in the indicated map that contain data with which the intermediate data is compared.
eliminating duplicate rows ...	
	Duplicate rows can exist in spools, 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 <i>n</i> .>	
	Delimits the running of the last AMP step in the row trigger loop. Control moves to the next step outside the trigger.
END TRANSACTION step ...	
	Indicates that processing is complete and that any locks on the data might be released. Changes made by the transaction are committed.
enhanced by dynamic partition elimination ...	
	Indicates a join condition where dynamic row partition elimination has been used.
estimated size ...	
	This value is the estimated size of the spool file needed to accommodate spooled data. This is only an estimate. Collecting appropriate statistics can improve the accuracy of the estimate. For more information on collecting statistics, see the discussion of the COLLECT STATISTICS (Optimizer Form) statement in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
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. This does not take into account other workloads running on the system.
execute the following steps in parallel ...	
	This phrase identifies a set of AMP steps that can be processed concurrently. The explanatory text immediately following the list describes the execution of each step.
from <i>n</i> sources... from <i>n</i> left sources... from <i>n</i> right sources...	
	RETRIEVE, JOIN, or aggregation from multiple sources that is performed in a single step.
(group_amps)	
	This phrase is used to indicate that a spool is created on a group of AMPs. Each AMP on which the step executes enters into a dynamic AMP group if rows are generated for that AMP. If rows are not generated for that AMP, it does not enter into a dynamic AMP group.

Phrase	Explanation
grouping by field <i>n</i> ([<i>database_name</i>]. <i>table_name</i> . <i>column_expression</i>).	
	The specified grouping column expression is based on column <i>n</i> in the specified table.
Hash table is built from Spool <i>n</i> and is duplicated to all AMPs in <i>mapname</i>	
	This phrase indicates that the step is an AllRowsOneAMP In-Memory hash join step.
in <i>mapname</i>	
	This phrase indicates the map defining a collection of AMPs.
in <i>mapname</i> covering <i>mapname</i> , <i>mapname</i> [, <i>mapname</i> ...]	
	This phrase indicates a map that includes, at least, all the AMPs in the maps listed after "covering".
INSERT into a single column partition of ...	
	This phrase indicates that at least one row is being inserted into a table or join index with only one user-specified column partition.
INSERT into <i>m₃</i> column partitions of ...	
	This phrase indicates that at least one row is being inserted into a table or join index with <i>m₃</i> column partitions. In this case, the value of <i>m₃</i> ≥ 2.
in view <i>view_name</i>	
	The specified [<i>database_name</i>]. <i>table_name</i> is accessed by means of the view <i>view_name</i> .
join condition ...	
	<p>The overall constraint that governs the join. In the following request, <code>employee.empno = department.mgrno</code> is the overall constraint governing the join:</p> <pre>SELECT deptname, name FROM employee, department WHERE employee.empno = department.mgrno;</pre> <p>Join conditions are also specified in the ON clause of an outer join or an ANSI SQL standard-formatted inner join.</p>
JOIN step by way of an all-rows scan ...	
	This phrase indicates that on each AMP on which they reside, spooled rows, primary table rows, or both are searched row by row; rows that satisfy the join condition are joined.
joined using a single partition exclusion hash join, with a join condition of (<i>join_condition</i>)	
	Vantage joins two spools using a single-partition exclusion hash Join on <i>join_condition</i> .
(Last Use)	

Phrase	Explanation
	This term identifies the last reference to a spool that contains intermediate data. The spool is released following this AMP step.
(load committed)...	
	<p>The read mode. Indicates that Vantage reads the rows that are load-committed. If the read mode is (load uncommitted), it indicates that Vantage ignores the logically deleted rows during processing. All other rows are read.</p> <p>For more information about load isolation, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144 and <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146.</p>
(load isolated)...	
	<p>Indicates that the modification operation performed in the step is load isolated. For this phrase to occur, the table being modified must be a load-isolated table or a join index defined on a load-isolated table.</p> <p>Delete operations that are load isolated do not physically delete the rows, rather, they mark the rows as logically deleted and row versioned. Update operations that are load isolated version the qualified rows such that older values are retained. Insert operations that are load isolated mark the rows with a load identity value that provides the commit property of the row to concurrent readers. During such modifications, a concurrent reader session on the table or join index can continue to obtain load-committed rows while the changes in the on-going load are not visible until committed. For more information about load isolation, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144 and <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146.</p>
locally on the AMPs ...	
	That portion of spooled intermediate or result data for which an AMP is responsible is stored on the AMP; it is not duplicated on or redistributed to other AMPs that are processing the same request.
lock ...	
	The Lock Manager places an ACCESS, READ, WRITE, or EXCLUSIVE lock on the database object that is to be accessed by a request.
MERGE into a single column partition of ...	
	This phrase indicates that rows are being merge inserted into a table or join index with only one user-specified column partition.
MERGE into m_3 column partitions of ...	
	<p>This phrase indicates that rows are being inserted into a table or join index with m_3 user-specified column partitions, using one column-partition context.</p> <p>$m_3 \geq 2$.</p> <p>Vantage can use either of two methods to insert the rows.</p> <ul style="list-style-type: none"> Take 1 pass over the source rows to insert the rows. <p>The column values from each source row are buffered with a separate 128KB output buffer for each target column partition. Vantage writes a buffer to its corresponding column partition when it is full or when all the source rows have been read).</p>

Phrase	Explanation
	<p>The source rows either do not need to be sorted for the MERGE step or have already been sorted by a previous AMP step. The phrase with buffered output is reported in the MERGE step for this method.</p> <ul style="list-style-type: none"> Fill a buffer with source rows and then sort it. <p>Vantage then scans the rows in the buffer once for each column partition in order to insert the column partition values for that column partition. This is repeated until all the source rows have been read.</p> <p>This method avoids an AMP step to spool and sort the rows. This is efficient if enough data is inserted for each set of buffered source rows into each combined partition when there is row partitioning for a target column-partitioned table and the source is a NoPI table.</p> <p>If the source is not a NoPI table, this method can cause an inefficient rereading and rewriting of data blocks that can occur when only a few column partition values are going into a combined partition at a time.</p> <p>The phrase with buffered input and sorted input is reported in the MERGE step for this method.</p> <p>The Optimizer recommends the least costly method to use, but an AMP might change to the other method as the rows are being inserted if it determines the other method might perform better.</p>
MERGE into m_3 column partitions (c_3 contexts) of ...	
	<p>This phrase indicates that rows are being inserted into a table or join index with m_3 user-specified column partitions, using c_3 column-partition contexts.</p> <p>$m_3 \geq 2$.</p> <p>$2 < c_3 < m_3$.</p> <p>Vantage can use one of two methods to insert the rows.</p> <ul style="list-style-type: none"> Make $\text{CEILING}(\frac{m_3}{c_3})$ passes over the source rows to insert them. <p>The column values from each source row that is read are buffered with a separate 128KB output buffer for each target column partition being processed by a pass. A buffer is written to its corresponding column partition when it is full or when all the source rows have been read.</p> <p>The source rows either do not need to be sorted for the MERGE step or have already been sorted by a previous AMP step. The source must be spooled if the source is a table or join index with an ACCESS lock. Even though this method requires multiple passes over the source rows, it is often less costly than the next method.</p> <p>Vantage reports the phrase with buffered output in the MERGE step for this method.</p> <ul style="list-style-type: none"> Fill a buffer with source rows and then sort it. <p>Vantage scans the rows in the buffer once for each column partition in order to insert the column partition values for that column partition and repeats this stage of the process until all the source rows have been read. This avoids an AMP step to spool and sort the rows and avoids making multiple passes over the source rows. This method can be efficient if enough data is inserted for each set of buffered source rows into each combined partition when there is row partitioning for a target column-partitioned table and the source is a NoPI table.</p> <p>If the source is not a NoPI table, this method can cause an inefficient rereading and rewriting of data blocks. This inefficiency can occur when only a few column partition values are going into a combined partition at a time.</p> <p>The phrase with buffered input and sorted input occurs in the MERGE step for this method.</p> <p>The Optimizer recommends the method to use to the AMPs, but an AMP might change to the other method as the rows are inserted if it determines that the other method might perform better.</p>
merge with matched updates and unmatched inserts ...	

Phrase	Explanation
	<p>An AMP step that can do any of the three following operations in a single step:</p> <ul style="list-style-type: none"> • Both update and insert operations • Insert operations only • Update operations only <p>The step assumes that the source table is always distributed on the join column of the source table, which is specified in the ON clause as an equality constraint with the primary index of the target table and sorted on the RowKey.</p> <p>The step performs a RowKey-based Merge Join internally, identifying source rows that qualify for updating target rows and source rows that qualify for inserts, after which it performs those updates and inserts.</p> <p>This step is very similar to the APPLY phase of MultiLoad because it guarantees that the target table data block is read and written only once during the MERGE operation.</p>
<p>from <i>foreign_table</i> metadata which is binpacked by size by using external metadata in Spool <i>n</i> to access <i>foreign_table</i></p>	
	<p>Indicates at least one of the tables involved is a foreign table.</p> <p><i>Foreign tables</i> allow Vantage to access data from external, cloud-based data storage, such as AWS S3, without requiring you to manually move the data into the database first from where it natively resides. Foreign tables are identified by a hostname, path, and other metadata that point to the external storage. Vantage can read and process semi-structured or unstructured external data in foreign tables using standard SQL. For example, you can use Teradata analytic functions to examine the data, join it to the relational data in the database, and issue queries against it as you can for other data in Vantage.</p>
(no lock required)	
	<p>The lock required for the operation was acquired in a previous AMP step.</p> <p>This phrase is not always reported for steps where a lock was acquired earlier.</p>
(nonconcurrent load isolated)...	
	<p>Indicates that the modification operation is not load isolated.</p> <p>For this phrase to occur, the table being modified must be a load-isolated table or a join index defined on a load-isolated table.</p> <p>Such modifications are like regular modifications on a non-load-isolated table that do not version the rows. The difference is that EXCLUSIVE locks are applied when such changes occur.</p> <p>During such modification, a concurrent reader session is blocked (including the case when ACCESS lock is used) until the transaction closes.</p> <p>For more information about load isolation, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144 and <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146.</p>
no residual conditions ...	
	<p>Rows are selected in their entirety; there are no specific search conditions. All applicable conditions have been applied to the rows.</p>
normalizing rows ...	
	<p>Indicates that Vantage performs a normalize operation as part of sort.</p>

Phrase	Explanation
of n_3 combined partitions	
	This phrase indicates that all the rows and columns in each of n_3 combined partitions of a table or join index, which has both row and column partitioning in this case, can be completely deleted as a fast path delete and storage for the deleted rows is recovered. $n_3 > 1$.
of n_3 combined partitions from ...	
	This phrase indicates that all the rows and columns in each of n_3 combined partitions of a table or join index that has both row and column partitioning are to be completely deleted as a fast path delete, and the storage for the deleted rows is recovered. n_3 is greater than one.
of n partitions from ...	
	This phrase indicates that all rows in each of n combined partitions can be completely deleted for a primary-indexed row-partitioned table or join index. The value of n is always greater than one. In some cases, this allows for faster deletion of entire partitions. This phrase is not applicable to column-partitioned tables or join indexes because other phrases are used in those cases.
of a single partition from ...	
	This phrase indicates that the Optimizer determined that all rows in a single combined partition for a row-partitioned table or join index with a primary index can be completely deleted. In some cases, this allows for faster deletion of the entire combined partition. This phrase is not applicable to column-partitioned tables or join indexes because there must be at least two column partitions for a column-partitioned table or join index: at least one specified column partition and the delete column partition.
on a reserved RowHash [in all partitions] [in n partitions] [in a single partition]...	
	This phrase indicates that the system has placed a proxy lock on the target table to prevent global deadlock for a later all-AMP lock on the table. The optional phrases "in all partitions," "in n partitions," and "in a single partition" appear only if the table is row-partitioned. The phrase "in all partitions" indicates a proxy for a table-level lock. The phrase "in n partitions" indicates a proxy for a partition-range lock. The phrase "in a single partition" indicates a proxy for a single-partition lock. For more information, see Proxy Locks .
partial SUM	
	A partial GROUP BY optimization has been done on the SUM step.
(partial-covering)	
	The NUSI in question partially covers the query using a constraint scan. For example, the system can probe a NUSI subtable to get the rowID values for base table rows that might qualify for the query, but it must then also access the underlying base table for the NUSI to obtain the rows that definitely qualify.

Phrase	Explanation
post join condition of ...	
	<p>Indicates that the optimizer detected a delayed condition and processed it as a post-join condition. This phrase appears immediately after the regular join condition.</p> <p>The following conditions trigger the use of post-join processing:</p> <ul style="list-style-type: none"> • Predicates that contain a CASE statement, or NULL sensitive predicates that refer to the inner table of an outer join. • Predicates that contain a complex OR term used in conjunction with an outer join. • Predicates that are used in conjunction with the NOT IN or the NOT EXIST connecting term.
redistributed by hash code to all AMPs in <i>mapname</i> ...	
	Given the values of an indexed or unindexed column, rows are sent to the AMPs in the indicated map that are responsible for storing the rows that use these values as a primary index.
right outer joined using a single partition hash join, with a join condition of (<i>join_condition</i>)	
	Vantage joins a spool from the left relation to a right relation using a two-step single-partition right outer hash join on <i>join_condition</i> .
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 # <i>n</i> ...	
	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 ...	
	<p>At most, 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.</p> <p>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.</p>
SORT/GROUP	
	Partial GROUP BY was used to reduce cardinality by simultaneously sorting and grouping.
SORT to normalize ...	
	Vantage performs the normalize operation as part of sort.
SORT to order Spool <i>n</i> by row hash ...	
	Rows in the spool are sorted by hash code to put them the same order as rows in the primary table, or in another spool on the same AMP, with which they are to be matched
SORT to order Spool <i>n</i> by row hash and the sort key in spool field1 eliminating duplicate rows ...	
	Rows in the spool are sorted by hash code using a uniqueness sort to eliminate duplicate rows. Uniqueness is based on the data in field1.

Phrase	Explanation
	<p>The contents of <i>field1</i> depend on the query and might comprise any of the following:</p> <ul style="list-style-type: none"> • A concatenation of all the fields in the spool row (used for queries with SELECT DISTINCT or that involve a UNION, INTERSECT, EXCEPT, or MINUS operation). • A concatenation of the rowIDs that identify the data rows from which the spool row was formed (used for complex queries involving subqueries). <p>Some other value that uniquely defines the spool row (used for complex queries involving aggregates and subqueries).</p>
SORT to order Spool 1 by the sort key in spool field1 ...	
	<p>Rows in the spool are sorted by the value in <i>field1</i>. The contents of <i>field1</i> are determined by the column set defined in the ORDER BY or WITH BY clause of the request being processed.</p> <p>For example, SORT TO <i>partition</i> BY ROWKEY or SORT to order Spool 1 by the sort key in spool field1 (THU.JI1.b1).</p>
SORT to partition by rowkey.	
	<p>This indicates that a join spool is joined using a rowkey-based join and sorted to the appropriate partitions.</p>
SORT to string_1 by string_2 ...	
	<p>Rows are sorted to <i>string_1</i> by their <i>string_2</i> sort key.</p> <p>For example, SORT TO <i>partition</i> BY ROWKEY or SORT to order Spool 1 by the sort key in spool field1 (THU.JI1.b1).</p>
SORT to partition Spool <i>n</i> by rowkey ...	
	<p>The Optimizer determined that a spool is to be partitioned based on the same partitioning expression as a table to which the spool is be joined. That is, the spool is to be sorted by rowkey (partition and hash). Partitioning the spool in this way allows for a faster join with the partitioned table. <i>n</i> is the spool number.</p>
spool <i>n</i> ...	
	<p>Identifies a spool, which is used to contain data during an operation.</p> <ul style="list-style-type: none"> • Spool 1 is normally used to hold a result before it is returned to the user. • Spools 2, 3, etc., contain intermediate data that produces a result.
Spool <i>n</i> statistics go into Spool <i>n</i> .	
	<p>In a dynamic EXPLAIN, identifies the spool where the statistics that were collected during the generation of the spool are located. If step generates multiple spools, all spools are included.</p>
spool_ <i>n</i> .Field ...	
	<p>Identifies the field of the spooled rows that is being used in a join constraint or comparison operation.</p> <p>For example:</p> <pre>personnel.employee.empno = spool_2.mgrno Statement 1...</pre>

Phrase	Explanation
	This term refers to the initiating request.
SUM step to normalize from ...	
	Indicates that Vantage performs a normalize operation as part of the aggregation of an aggregation step.
table-level summary statistics	
	Explains that Vantage is computing the summary statistics, not detailed statistics, for a table.
The actual size of Spool <i>n</i> is <i>n</i> row[s] (<i>n</i> bytes).	
	Reports the actual size of the spool, if available. After the steps where the location of the Spool <i>n</i> statistics is reported and the statistics are consolidated, if the statistics are retrieved from additional runtime information, the size of the spool is given.
The estimated time for this step is <i>nn</i> seconds.	
	The estimated time for the reported Delete, Insert, Join, Merge, Merge Delete, Merge Update, Retrieval, Sum, or Update step in seconds. Note that the time shown is not clock time; you should consider it to be an arbitrary unit of measure that you can use to compare different operations.
the estimated time is <i>nn</i> seconds.	
	The estimated time to completion for the entire request in seconds. The time shown is not clock time; you should consider it to be an arbitrary unit of measure that you can use to compare different operations.
The following is the dynamic explain for the request.	
	This phrase indicates the query plan is a dynamic plan generated by incremental planning and execution (IPE) for this request. This phrase, if included, occurs before the first step of the dynamic plan.
The size of Spool <i>x</i> is estimated with low confidence to be <i>y</i> rows (<i>z</i> bytes).	
	The estimated size of spool <i>n</i> (which is always reported with Low Confidence - see About Optimizer Confidence Levels), where <i>m</i> is the estimated cardinality of the spool and <i>o</i> is the estimated number of bytes in the spool. The spool size in bytes is calculated as follows: Estimated spool size (bytes) = Estimated spool cardinality × Row size Row size is an estimated average row size if the row includes variable length or compressed columns.
... the Spatial index subtable ...	
	The index being created or dropped is a spatial NUSI.
This request is eligible for incremental planning and execution (IPE) but dynamic plan does not support USING request modifier. The following is the static plan for the request:	

Phrase	Explanation
	This is a generic plan for the request that is eligible for incremental planning and execution; however, DYNAMIC EXPLAIN is not supported because the phrase being explained is preceded by a USING request modifier. The phrase, if included, occurs before the first step of the EXPLAIN text with a DYNAMIC EXPLAIN request modifier.
This repeated request is eligible for incremental planning and execution (IPE) but it may also be eligible to be cached so a generic plan is tried. The following is the static plan for the request:	
	This is a generic plan for the request that is eligible for incremental planning and execution; however, this is a repeated request whose generic plan might be eligible to be cached, so Vantage tries a generic plan the second time. A generic plan is always a static plan. The phrase, if included, occurs before the first step of the generic plan in DBC.DBQLExplainTbl if the EXPLAIN text for the plan is being logged by DBQL; it does not occur when you specify an EXPLAIN, STATIC EXPLAIN, or DYNAMIC EXPLAIN request modifier.
This request is eligible for incremental planning and execution (IPE) but doesn't meet thresholds. The following is the static plan for the request:	
	Vantage reports this phrase before the first step of a static plan. This is a specific static plan that is eligible for incremental planning and execution, but does not meet certain system-specified cost thresholds. If Vantage executes the request, this static plan is generated and executed.
This request is eligible for incremental planning and execution (IPE) but dynamic explain does not support USING request modifier. The following is the static plan for the request:	
	The request is eligible for incremental planning and execution based on this static plan; however, the dynamic plan cannot be displayed for requests with a USING request modifier, so a generic plan is displayed instead. This phrase, if included, occurs before the first step of the static plan. A generic plan is always a static plan. If the request is executed, a generic plan, a specific static plan, or a dynamic plan might be generated and executed. This phrase does not occur in DBC.DBQLExplainTbl in the EXPLAIN text for a plan being logged by DBQL.
This request is eligible for incremental planning and execution (IPE). The following is the static plan for the request:	
	Vantage reports this phrase before the first step of a static plan. This is a specific static plan that is eligible for incremental planning and execution and, if the request is executed, a dynamic plan might be generated and executed instead of this static plan. However, this static plan is the one used for evaluating workload filters, throttles, and classification criteria.
This request is eligible for incremental planning and execution (IPE). IPE is disabled. The following is the static plan for the request:	
	This is a static plan for this request that is eligible for incremental planning and execution; however IPE is disabled for the system. If included, Vantage reports the phrase before the first step of the static plan. If the request is executed and IPE is still disabled, Vantage generates and executes a static plan; if the request is executed and IPE is enabled, a dynamic plan is generated. To enable dynamic plans, contact Teradata Services.
two-AMP RETRIEVE step by way of unique index #n ...	

Phrase	Explanation
	<p>A row of a table is selected based on a USI value:</p> <ul style="list-style-type: none"> At most, a single row in the USI subtable is selected using the index value that hashes to the AMP on which the subtable row is stored. The hash value in the index rowID determines the AMP on which the data row is stored.
unsatisfiable	
	<p>The Optimizer has added an implied constraint to identify an unsatisfiable condition for a Period column when a DML request is defined with a specific set of BEGIN and END search condition constraints. See Predicate Simplification.</p>
using covering CP merge Spool q (s_2 subrow partitions and Last Use) using covering CP merge Spool q ($s_1 + s_2$ subrow partitions and Last Use)	
	<p>These phrases indicate that a column-partitioned merge spool is created and used to merge column partitions from the table or join index, then the resulting s_2 subrow column partitions are read from the column-partitioned merge spool.</p> <p>s_1 indicates the number of intermediate subrow column partitions needed for the merges, while $s_1 + s_2$ indicates the number of merges needed.</p> <p>If s_1 is not included, no intermediate subrow column partitions are needed for the merges, and s_2 merges are required.</p> <p>The number of merges needed depends on the number of column partitions that need to be accessed and the number of available column partition contexts as indicated in a preceding from phrase. This is the last usage of this column-partitioned merge spool so it can be deleted.</p> <p>q is the spool number for the column-partitioned merge spool.</p>
using covering CP merge Spool q (s_2 subrow partitions and Last Use) and rowid Spool k (Last Use) using covering CP merge Spool q ($s_1 + s_2$ subrow partitions and Last Use) and rowid Spool k (Last Use)	
	<p>These phrases indicate that a column-partitioned merge spool is created and used to merge column partitions from the table or join index, and then the resulting s_2 subrow column partitions are read from the column-partitioned merge spool, driven by the rowid spool.</p> <p>s_1 indicates the number of intermediate subrow column partitions needed for the merges.</p> <p>$s_1 + s_2$ indicates the number of merges needed.</p> <p>If s_1 is not included, no intermediate subrow column partitions are needed for the merges and in addition, there are s_2 merges needed.</p> <p>The number of merges needed depends on the number of column partitions that need to be accessed and the number of available column partition contexts as indicated in a preceding from phrase.</p> <p>This is the last usage of both this column-partitioned merge spool and rowID spool so they can be deleted.</p> <p>q is the spool number for the column-partitioned merge spool.</p> <p>k is the spool number for the rowID spool.</p> <p>If rowID Spool k is generated by this AMP step instead of a previous step, a built from phrase follows this using phrase.</p>
using CP merge Spool q (one subrow partition and Last Use) using CP merge Spool q (s_2 subrow partitions and Last Use) using CP merge Spool q ($s_1 + s_2$ subrow partitions and Last Use)	

Phrase	Explanation
	<p>These phrases indicate that a column-partitioned merge spool is created and used to merge column partitions from the table or join index. Then, other column partitions from the table or join index and the resulting one subrow column partition or s_2 subrow column partitions from the column-partitioned merge spool are read.</p> <p>s_1 indicates the number of intermediate subrow column partitions needed for the merges.</p> <p>$s_1 + s_2$ indicates the number of merges needed.</p> <p>If s_1 is not included, no intermediate subrow column partitions are needed for the merges, and there is also one or s_2 merges needed.</p> <p>The number of merges needed depends on the number of column partitions that need to be accessed and the number of available column partition contexts, as indicated in a preceding from phrase.</p> <p>This is the last usage of this column-partitioned merge spool, so they can be deleted.</p> <p>q is the spool number for the column-partitioned merge spool.</p>
	<p>using CP merge Spool q (one subrow partition and Last Use) and rowid Spool k (Last Use)</p> <p>using CP merge Spool q (s_2 subrow partitions and Last Use) and rowid Spool k (Last Use)</p> <p>using CP merge Spool q ($s_1 + s_2$ subrow partitions and Last Use) and rowid Spool k (Last Use)</p>
	<p>These phrases indicate that a column-partitioned merge spool is created and used to merge column partitions from the table or join index, and then some other column partitions from the table or join index and the resulting one subrow column partition or s_2 subrow column partitions from the column-partitioned merge spool are read, driven by the rowid spool.</p> <p>s_1 indicates the number of intermediate subrow column partitions needed for the merges.</p> <p>$s_1 + s_2$ indicates the number of merges needed.</p> <p>If s_1 is not included, no intermediate subrow column partitions are needed for the merges, and there is also one or s_2 merges needed.</p> <p>This is the last usage of both this column-partitioned merge spool and rowid spool so they can be deleted.</p> <p>q is the spool number for the column-partitioned merge spool.</p> <p>k is the spool number for the rowID spool.</p> <p>If rowID Spool k is generated by this AMP step instead of a previous step, a built from phrase follows this using phrase.</p>
	using rowid Spool k (Last Use)
	<p>This phrase indicates that a rowID spool contains the rowIDs of rows in the table that qualify, and then the column-partitioned table or join index is read, driven by this rowID spool.</p> <p>This is the last usage of this rowID spool, so it can be deleted.</p> <p>k is the spool number for the rowID spool.</p> <p>If rowID Spool k is generated by this AMP step instead of a previous step, a built from phrase follows this using phrase.</p> <p>This phrase is used in AMP steps, such as RETRIEVE, JOIN, and MERGE, that might read a column-partitioned source.</p>
	using rowid Spool k (Last Use) built from m_2 column partitions
	<p>This phrase indicates that the rowID Spool k is built containing the rowIDs of rows in the table that qualify, and then the column-partitioned table or join index is read, driven by this rowID spool.</p>

Phrase	Explanation
	<p>Up to m_2 column partitions of the column-partitioned table or join index may need to be accessed to evaluate predicates and build the rowID spool.</p> <p>There are column partition contexts available for each of the m_2 column partitions.</p> <p>For this phrase to occur, the column-partitioned table or join index would not have row partitioning.</p> <p>$m_2 \geq 2$.</p> <p>One of the column partitions accessed may be the delete column partition.</p> <p>Not all of the m_2 column partitions may actually need to be accessed if no rows qualify.</p> <p>This phrase is used in AMP steps such as RETRIEVE, JOIN, and MERGE that might read a column-partitioned source.</p>
using rowid Spool k (Last Use) built from m_2 column partitions (c_2 contexts)	
	<p>This phrase indicates that a rowID spool is created that contains the rowIDs of rows in the table or join index that qualify. Then the column-partitioned table or join index is read, driven by this rowID spool.</p> <p>Up to m_2 column partitions of the column-partitioned table or join index may need to be accessed using c_2 column-partition contexts to evaluate predicates and build the rowID spool. For this phrase to occur, the column-partitioned table or join index does not have row partitioning.</p> <p>k is the spool number for the rowID spool.</p> <p>$m_2 \geq 2$.</p> <p>$2 < c_2 < m_2 - 1$ and is equal to or one less than the number of available column-partition contexts.</p> <p>One of the column partitions accessed might be the delete column partition. Not all of the m_2 column partitions might need to be accessed if no rows qualify.</p> <p>This phrase is used in AMP steps such as RETRIEVE, JOIN, and MERGE that might read a column-partitioned source.</p>
using rowid Spool k (Last Use) built from n_2 combined partitions (m_2 column partitions)	
	<p>This phrase indicates that the rowID Spool k is built containing the rowids of rows in the table or join index that qualify, and then the column-partitioned table or join index is read, driven by this rowID spool.</p> <p>The rows and columns for up to n_2 combined partitions of the table or join index (which has both row and column partitioning in this case) might need to be accessed to evaluate predicates and build the rowID spool.</p> <p>k is the spool number for the rowID spool.</p> <p>There are column partition contexts available for each of the m_2 column partitions.</p> <p>$m_2 \geq 2$.</p> <p>One of the column partitions accessed might be the delete column partition.</p> <p>Not all of the m_2 column partitions might need to be accessed if no rows qualify.</p> <p>This phrase is used in AMP steps such as RETRIEVE, JOIN, and MERGE that might read a column-partitioned source.</p>
using rowid Spool k (Last Use) built from n_2 combined partitions (m_2 column partitions and c_2 contexts)	
	<p>This phrase indicates that a rowID spool is created that contains the rowIDs of rows in the table that qualify. Then the column-partitioned table is read, driven by this rowID spool. This is the last usage of this rowID spool, so it can be deleted.</p> <p>k is the spool number for the rowID spool.</p>

Phrase	Explanation
	<p>The rows and columns for up to n_2 combined partitions of the table or join index, which has both row and column partitioning, might need to be accessed to evaluate predicates and build the rowID spool.</p> <p>For the column-partitioning level, c_2 column-partition contexts are used to access up to m_2 column partitions.</p> <p>$n_2 \geq s$ and $m_2 \geq 2$.</p> <p>$2 < c_2 < m_2 - 1$ and is equal to or one less than the number of available column-partition contexts.</p> <p>One of the column partitions that might be accessed is the delete column partition. Not all of the m_2 column partitions might need to be accessed if no rows qualify.</p> <p>This phrase is used in AMP steps such as RETRIEVE, JOIN, and MERGE that might read a column-partitioned source.</p>
we compute the table-level summary statistics from spool n and save them into spool m	
	<p>This phrase indicates that Vantage takes rolled up pre-aggregated statistics information from spool n, computes table-level summary statistics from that data, and saves the summary statistics it computed in spool m.</p>
we do a BMSMS... (bit map set manipulation step)	
	<p>BMSMS (bit map set manipulation step) is a method for handling weakly selective secondary indexes that have been ANDed; NUSI bit mapping.</p> <p>BMSMS intersects the following rowID bit maps:</p> <div style="background-color: #f0f0f0; padding: 10px; margin: 10px 0;"> <pre> 1) The bit map built for ... by way of index # $n...$ 2) The bit map built for ... by way of index # $n...$... </pre> </div> <p>The resulting bit map is placed in Spool $n...$</p> <div style="background-color: #f0f0f0; padding: 10px; margin: 10px 0;"> <p>BMSMS...</p> </div> <p>Indicates that two or more bit maps are intersected by ANDing them to form one large bit map.</p> <div style="background-color: #f0f0f0; padding: 10px; margin: 10px 0;"> <p>index # $n...$</p> </div> <p>Identifies, in the order in which they are ANDed, each nonunique secondary index used in the intersection.</p> <div style="background-color: #f0f0f0; padding: 10px; margin: 10px 0;"> <p>resulting bit map is placed in Spool $n...$</p> </div> <p>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.</p> <p>See also EXPLAIN Request Modifier: Examples.</p>
we do a GEOSPATIAL COLLECT STATISTICS summary step ...	
	<p>This phrase indicates that Vantage uses a SUM step to aggregate geospatial NUSI statistics when index statistics are collected on a geospatial column.</p>
we do an ABORT test ...	

Phrase	Explanation
	An ABORT or ROLLBACK statement was detected.
we do [an all-AMPs a single-AMP a group-AMPs] DISPATCHER RETRIEVE step from Spool 2 (Last use) by way of an all-rows scan and send the rows back to the Dispatcher.	
	This phrase indicates that Vantage uses either an all-AMPs, a single AMP, or a group-AMPs step to retrieve rows from the spool. The consolidated rows from multiple AMPs are then sent back to the dispatcher. This step is used to process row-based triggers, noncorrelated scalar subqueries, and so on.
we do [an all-AMPs a single-AMP a group-AMPs] FEEDBACK RETRIEVE step from Spool <i>n</i> (Last Use) of Spool <i>n</i> statistics.	
	This phrase indicates that Vantage uses either an all-AMPs, a single AMP, or a group-AMPs step to retrieve statistics. When the statistics are computed during the spool building time, a feedback retrieve step consolidates the statistics feedback from the AMP local statistics to global statistics and sends them back to the optimizer.
we do [an all-AMPs a single-AMP a group-AMPs] FEEDBACK RETRIEVE step from <i>n</i> % of Spool <i>n</i> to collect statistics.	
	This phrase indicates that it was not possible for Vantage to generate statistics while the spool was being generated, so the statistics were derived by sampling with the feedback retrieve step.
we do [an all-AMPs a single-AMP a group-AMPs] FEEDBACK RETRIEVE step from Spool <i>n</i> (Last use).	
	This phrase indicates that the Teradata optimizer generated results feedback steps.
we do an all-AMPs SUM step in <i>mapname</i> to aggregate from <i>database_name.table_name</i> by way of an all-rows scan with no residual conditions, grouping by field1 (<i>database_name.table_name.column_name_1</i>).	
	This phrase indicates that Vantage uses an all-AMPs SUM step in the indicated map to aggregate statistics information from a single column of table or join index <i>table_name</i> in database or user <i>database_name</i> and groups them by <i>column_name_1</i> .
we do an all-AMPs SUM step in <i>mapname</i> to aggregate from <i>database_name.table_name</i> by way of an all-rows scan with no residual conditions, grouping by field1 (<i>database_name.table_name.column_name_1</i> , <i>database_name.table_name.column_name_2</i>)	
	<p>This phrase indicates that Vantage uses an all-AMPs SUM step in the indicated map to aggregate statistics information from table or join index <i>table_name</i> in database or user <i>database_name</i> and groups it by (<i>column_name_1</i>, <i>column_name_2</i>).</p> <p>For this case, <i>column_name_1</i> and <i>column_name_2</i> are highly nonunique. Because of this, the optimizer made a cost-based decision to pre-aggregate on those two columns and to roll up the result to individual columns. This optimization guarantees that the base table is read only once and the aggregations on the individual columns run faster because they are working on pre-aggregated data.</p> <p>This optimization is used only for recollecting statistics, and is possible only when the applicable statistics are specified in a single COLLECT STATISTICS request.</p>
we do an all-AMPs SUM step in <i>mapname</i> to aggregate from <i>database_name.table_name</i> by way of a traversal of index # <i>n</i> without accessing the base table with no residual conditions, grouping by field1 (<i>database_name.table_name.column_name_1</i> , <i>database_name.table_name.column_name_2</i>).	

Phrase	Explanation
	<p>This phrase indicates that Vantage uses an all-AMPs SUM step in the indicated map to aggregate statistics information from multiple columns of table or join index <i>table_name</i> in database or user <i>database_name</i> and groups them by <i>column_name_1</i>.</p> <p>For this particular case, single-column statistics are collected for <i>column_name_1</i> and for <i>column_name_2</i>, and multicolumn statistics are collected for (<i>column_name_1</i>, <i>column_name_2</i>).</p> <p>The aggregation for the multicolumn case is done first. The result of this aggregation is used to roll up the statistics for individual statistics on <i>column_name_1</i> and <i>column_name_2</i>. This way, the base table is read only once and the aggregations on the individual columns runs faster because they are working on the pre-aggregated data.</p> <p>This optimization is possible only when all the three statistics are specified in a single COLLECT STATISTICS request.</p>
we do a SMS (set manipulation step) ...	
	Combine rows under control of a UNION, EXCEPT, MINUS, or INTERSECT operator.
we lock a distinct [<i>databasename</i> .]"pseudo table" for [locking_severity] on a RowHash for deadlock prevention	
	This phrase indicates the pseudo lock that is used to serialize primary and fallback RowHash locks on a dictionary table for DDL processing. See Pseudo Table Locks for details.
we lock [<i>databasename</i> .] <i>tablename</i> for exclusive use on a reserved RowHash to prevent global deadlock	
	This phrase indicates the proxy lock that serializes the exclusive lock for <i>databasename.tablename</i> in the lock phrase that follows this one.
we lock [<i>databasename</i> .] <i>tablename</i> for [lock_severity] we lock [<i>databasename</i> .] <i>tablename</i> for [lock_severity] on a single partition we lock [<i>databasename</i> .] <i>tablename</i> for [lock_severity] on <i>n</i> partitions	
	The Lock Manager sets an EXCLUSIVE lock on <i>database_name.table_name</i> . The last phrase occurs only if DDL processing is occurring on the DBC.AccessRights dictionary table.
we lock [<i>databasename</i> .] <i>tablename</i> for [locking_severity] on a [locking_level] for deadlock prevention	
	<p>The Lock Manager sets a lock for the indicated locking severity at the indicated locking level on <i>databasename.tablename</i>.</p> <p>[locking_level] can be one of the following levels:</p> <ul style="list-style-type: none"> • reserved RowHash • reserved RowHash in a single partition • reserved RowHash in <i>n</i> partitions • reserved RowHash in all partitions <p>The first levels is used for non-row-partitioned table and levels 2 to 4 are used for row-partitioned tables.</p>
we lock [<i>databasename</i> .] <i>tablename</i> for single writer on a reserved RowHash to prevent global deadlock we lock [<i>databasename</i> .] <i>tablename</i> for single writer on a reserved RowHash in all partitions to prevent global deadlock we lock [<i>databasename</i> .] <i>tablename</i> for single writer we lock [<i>databasename</i> .] <i>tablename</i> for single writer on a reserved RowHash in all partitions to serialize concurrent updates on the partition-locked table to prevent global deadlock	

Phrase	Explanation
	we lock [<i>database</i> .] <i>tablename</i> for single writer to serialize concurrent updates on the partition-locked table
	<p>The first phrase indicates a proxy lock that is used for a non-row-partitioned table.</p> <p>The second phrase indicates a proxy lock that is used for a row-partitioned table.</p> <p>Both the first and second phrases indicate serialization locks for the proxy lock in the third phrase. A table-level single writer lock allows concurrent reads on the data that is not rowhash write locked.</p> <p>The fourth phrase indicates a proxy lock used to serialize the all-AMP single writer lock at phrase 5. This type of single writer lock is used in certain cases of partition-locked bulk DML, where NUSI, USI or RI maintenance on the target table is required.</p> <p>The fifth phrase indicates a single-writer lock that allows concurrent reads on the data in a table that is not write locked during bulk DML processes.</p>
	we save the UPDATED STATISTICS for (' <i>a</i> ',' <i>b</i> ') from Spool <i>m</i> into Spool <i>n</i> , which is built locally on a single AMP in <i>mapname</i> derived from the hash of the table id. Statistics are collected since the age of the statistics (<i>y</i> days) exceeds the time threshold of <i>x</i> days and the estimated data change of <i>z</i> % exceeds the system-determined change threshold of <i>zx</i> %.
	The user specified a time threshold that is met and the specified system change threshold is also met, so Vantage recollects the requested statistics.
	we save the UPDATED STATISTICS for (' <i>a</i> ',' <i>b</i> ') from Spool <i>m</i> into Spool <i>n</i> , which is built locally on a single AMP in <i>mapname</i> derived from the hash of the table id. Statistics are collected since the estimated update of <i>z</i> % exceeds the system-determined update threshold of <i>x</i> %.
	The system-determined change threshold is met and the columns of interest have been updated after the last recollection of statistics, so Vantage recollects the requested statistics.
	we save the UPDATED STATISTICS for (' <i>a</i> ',' <i>b</i> ') from Spool <i>m</i> into Spool <i>n</i> , which is built locally on a single AMP in <i>mapname</i> derived from the hash of the table id. Statistics collected since the estimated data change could not be determined.
	<p>The Optimizer has a difficult time estimating the cardinality changes and update activity on related columns, so Vantage does recollect the requested statistics.</p> <p>This happens when the Optimizer detects one of the following 4 conditions:</p> <ul style="list-style-type: none"> • UDI counts are not reliable because they are not consistent with dynamic AMP sampling or reliable trends. • The target table is small, where aggregation is estimated to take less than 0.5 seconds. • There is no reliable trend on the change of major statistics. • Sampled statistics in the latest collection did not follow the statistics estimated by trending.
	we save the UPDATED GEOSPATIAL STATISTICS for <i>column</i> from Spool <i>m</i> (Last Use) into Spool <i>n</i> ...
	Vantage copies the updated geospatial NUSI statistics collected on <i>column</i> locally from spool <i>m</i> into spool <i>n</i> .
	We send an END PLAN FRAGMENT step for plan fragment <i>n</i> .
	<p>This phrase indicates to the dispatcher the end of a plan fragment in a dynamic plan. The step occurs at the end of each plan fragment for a dynamic plan except for the last. This step is not sent to the AMPs.</p> <p><i>n</i> indicates the number of the plan fragment. There is no step in the plan to indicate the beginning of a plan fragment.</p>

Phrase	Explanation
	For a dynamic plan, the first step is the beginning of the first plan fragment. The first step after an END PLAN FRAGMENT step is the beginning of the next plan fragment. The last step of a dynamic plan is the end of the last plan fragment, and there is no END PLAN FRAGMENT step for the last plan fragment. The number of plan fragments in the plan and the number of the last plan fragment is the number in the last END PLAN FRAGMENT step plus 1.
	We SKIP collecting STATISTICS for ('a,b'), because the age of the statistics (<i>n</i> days) does not exceed the user-specified time threshold of <i>m</i> days and the estimate data change of <i>o</i> % does not exceed the user-specified change threshold of <i>p</i> %.
	The user specified a time threshold and a system-determined change threshold is enabled. Neither threshold is met, so Vantage does not recollect the specified statistics.
	We SKIP collecting STATISTICS for ('a,b'), because the estimated data change of 5% does not exceed the user-specified change threshold of 20%.
	The user specified a data change threshold that the statistics do not meet, so Vantage does not recollect the specified statistics.
	which is duplicated on all AMPs in <i>mapname</i> ...
	Relocating data in preparation for a join.
	which is redistributed by hash code to all AMPs in <i>mapname</i> ...
	Relocating data by hash code in preparation for a join.
	which is redistributed by the hash code of ([database_name.]table_name.column_expression) which is redistributed by the hash code of ([database_name.]table_name.column_expression) to few AMPs in <i>mapname</i> . which is redistributed by the hash code of ([database_name.]table_name.column_expression) to few or all AMPs in <i>mapname</i> . which is redistributed by the hash code of ([database_name.]table_name.column_expression) to all AMPs in <i>mapname</i> .
	Relocating data by the hash code of the specified column expression based on a column set from <i>table_name</i> in preparation for a join.
	which is redistributed randomly ...
	When you specify HASH BY RANDOM for an INSERT ... SELECT request, the Optimizer redistributes blocks of selected rows randomly before they are optionally ordered locally by specifying a LOCAL ORDER BY clause, and inserting the rows locally into the target table. The Optimizer does this by taking a generated response spool, generating a random hash value for it, and changing the existing row hash value before redistributing the rows. This is useful when the result of the SELECT operation does not provide an even distribution.

EXPLAIN Request Modifier: Examples

This section examines the usefulness of the EXPLAIN request modifier in different situations.

Note:

The precise EXPLAIN output in these examples may differ from what you encounter because execution plans can vary, depending on the system configuration and database release. To simplify the examples, some EXPLAIN phrases (such as in *map_name*) and steps (such as locking and END TRANSACTION) are omitted from some of the examples, where only the relevant steps are shown.

You should always use EXPLAIN reports 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 returning the identical result set.

EXPLAIN request modifier terms are defined in [EXPLAIN Request Modifier Phrase Terminology](#).

In the examples that use the personnel.employee table, it is defined to have a unique primary index defined on the emp_no column and a nonunique secondary index defined on the name column.

ANSI Versus Teradata Session Mode Update Example

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 PERSONNEL.employee for write
   on a reserved RowHash to prevent global deadlock.
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 PERSONNEL.employee for write on a
   reserved RowHash to prevent global deadlock.
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.

Few-AMPs Row Redistribution Example

Assume that you create a multiset table and a second table, as follows:

```
CREATE MULTISET TABLE customer_order (customer_name CHAR(10), order_no INTEGER)
PRIMARY INDEX (customer_name);

CREATE TABLE order_detail (order_no INTEGER, product_name CHAR(20))
PRIMARY INDEX (order_no);
```

The EXPLAIN request modifier generates the few AMPs text in step 1, showing that the step uses few AMPs for row redistribution. Query step 1 uses few AMPs because of the following factors:

- the step is initiated in a single AMP that has rows with 'Smith' (NUPI), AND
- the Smith rows are redistributed by the hash code of the order_no column, AND
- the number of the redistributed rows is small and the rows are hashed to only few AMPs.

```
EXPLAIN
SELECT customer_name, product_name
FROM customer_order CO, order_detail OD
WHERE CO.customer_name = 'Smith' AND CO.order_no = OD.order_no;
```

Explanation

-
- 1) First, we do a single-AMP RETRIEVE step from JW.CO by way of the primary index "JW.CO.customer_name = 'Smith '" with a residual condition of ("NOT (JW.CO.order_no IS NULL)") into Spool 2 (group_amps), which is redistributed by the hash code of (JW.CO.order_no) to **few 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 2 rows (54 bytes). The estimated time for this step is 0.01 seconds.
 - 2) Next, we do a group-AMPs JOIN step from JW.OD by way of a RowHash match scan, which is joined to Spool 2 (Last Use) by way of a RowHash match scan. JW.OD and Spool 2 are joined using a merge join, with a join condition of

("order_no = JW.OD.order_no"). The result goes into Spool 1 (group_amps), which is built locally on that AMP. The size of Spool 1 is estimated with index join confidence to be 3 rows (153 bytes). 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.

-> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.11 seconds.

One-Step Multisource RETRIEVE for a UNION ALL

- 5) We do an all-AMPs RETRIEVE step in TD_MAP1 from 2 sources:
- a) TEST.t1 by way of an all-rows scan with a condition of ("NOT (TEST.t1.a1 IS NULL)").
 - b) TEST.t2 by way of an all-rows scan with a condition of ("NOT (TEST.t2.a2 IS NULL)").

The result goes into Spool 1 (all_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 126 rows (3,150 bytes). The estimated time for this step is 0.15 seconds.

One-Step Multisource JOIN

- 6) We do an all-AMPs JOIN step in TD_Map1 from 2 left sources, which is joined to TEST.t1 by way of an all-rows scan using a dynamic hash join. The left sources are:
- a) Spool 3 (Last Use) by way of an all-rows scan, with a join condition of ("TEST.t1.b1 = a2").
 - b) Spool 4 (Last Use) by way of an all-rows scan, with a join condition of ("TEST.t1.b1 = a3").

The result goes into Spool 2 (group_amps), which is built locally on the AMPs. The size of Spool 2 is estimated with no confidence to be 90 rows (2,880 bytes). The estimated time for this step is 0.23 seconds.

One-Step Multisource SUM

- 4) We do an all-AMPs SUM step in TD_MAP1 to aggregate from 2 sources:
- a) TEST.t1 by way of a cylinder index scan with no residual conditions.
 - b) TEST.t2 by way of a cylinder index scan with no residual conditions.

Aggregate Intermediate Results are computed globally, then placed in Spool 9 in TD_Map1. The size of Spool 9 is

estimated with high confidence to be 1 row (23 bytes). The estimated time for this step is 0.18 seconds.

EXPLAIN Request Modifier and Standard Indexed Access

The EXPLAIN request modifier is useful in determining whether the indexes defined for a table are properly defined, useful, and efficient.

As illustrated in other join examples, EXPLAIN identifies any unique indexes that might 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 *dept_no*, *salary*, *yrs_exp*, and *ed_lev* 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 subtable rowIDs is the faster retrieval method.

Full-Table Scan Example

This request returns the following EXPLAIN report:

```
EXPLAIN
SELECT COUNT(*)
FROM employee
WHERE dept_no = 500
AND salary > 25000
AND yrs_exp >= 3
AND ed_lev >= 12;
```

Explanation

-
- 1) First, we lock PERSONNEL.employee for read on a reserved RowHash to prevent global deadlock.
 - 2) Next, we lock PERSONNEL.employee for read.
 - 3) We do an all-AMPS 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.

- 4) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of an all-rows scan into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (32 bytes). The estimated time for this step is 0.06 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.

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.
computed globally	The final computation involves all the intermediate spool data.

Unique Primary Index Example

The EXPLAIN request modifier generates the following response for this request:

```
EXPLAIN
SELECT name, dept_no
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.
```

SELECT Example With a Condition on a 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 request modifier generates the following response for this request:

```
EXPLAIN
SELECT emp_no, dept_no
FROM employee
WHERE name = 'Smith T';
```

Explanation

-
- 1) First, we lock PERSONNEL.employee for read on a RowHash to prevent global deadlock.
 - 2) Next, we lock PERSONNEL.employee for read.
 - 3) We do an all-AMPs RETRIEVE step from PERSONNEL.employee by way of index # 4 "PERSONNEL.employee.Name = 'Smith T '" with no residual conditions into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 33,563 rows (1,443,209 bytes). The estimated time for this step is 0.34 seconds.
 - 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.34 seconds.

SELECT Example With a Condition on a Unique Secondary Index

Assume that the employee table has another column (socsecno), where soc_sec_no is defined as a unique secondary index.

If the WHERE condition is based on this column, then the EXPLAIN request modifier generates the following response for this request:

```
EXPLAIN
SELECT name, emp_no
FROM employee
WHERE soc_sec_no = '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.

Indexed Access With Bit Mapping Example

This request returns the following EXPLAIN report:

```
EXPLAIN
SELECT COUNT(*)
FROM main
```



```
WHERE num_a = '101'
AND num_b = '02'
AND kind = 'B'
AND event = '001';
```

Explanation

```
...
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 built for TESTING.Main by way of index # 8
   "TESTING.Main.Num_B = '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.Num_A = '101'" and the bit map in Spool
3 (Last Use) with a residual condition of
("(TESTING.Main.Num_B = '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.
...
```

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: The bit map built for ... by way of index # <i>n</i> ... The bit map built for ... by way of index # <i>n</i> ...	On each nonunique secondary index subtable, each data rowID is assigned a number from 0-32,767. This number is used as an index into a bit map in which the bit for each qualifying row is turned on. BMSMS indicates that the intersection of sets of qualifying rows is computed by applying the logical AND operation to the bitmap representation of the sets.
residual condition	Selected rows are further qualified by one or more conditional expressions.

EXPLAIN Request Modifier and Join Processing

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.

Product Join Example

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.

```
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
-----
...
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.
...
```

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.
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 Vantage.

Merge Join Example

This request returns the following EXPLAIN report:

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

Explanation

```
-----
...
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.
...
```

New terminology in this explanation is defined as follows:

Phrase	Definition
merge join	One of the join methods used by Vantage.

Hash Join Example

This request returns the following EXPLAIN report:

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

Explanation

```
-----
...
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.
...
```

New terminology in this explanation is defined as follows:

Phrase	Definition
hash join	One of the types of join processing performed by Vantage.

Nested Join Example

This request returns the following EXPLAIN report:

```
EXPLAIN
SELECT dept_name, name, yrs_exp
FROM employee, department
WHERE employee.emp_no = department.mgr_no
AND department.dept_no = 100;
```

Explanation

```
-----
1) First, we do a single AMP JOIN step from PERSONNEL.Department by way
of the unique primary index "PERSONNEL.Department.Dept_No = 100" with no
residual condition which is joined to PERSONNEL.Employee by way of the
unique primary index "PERSONNEL.Employee.Emp_No = PERSONNEL.Department.Mgr_No".
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.
...
```

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. 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 Vantage.

Exclusion Merge Join Example

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 subquery (for example, if the request was `SELECT name, dept_no, loc FROM employee, department`), the result would be a Cartesian product.

```
EXPLAIN
SELECT name, dept_no
FROM employee
WHERE dept_no NOT IN (SELECT dept_no
```

```
FROM department
WHERE loc = 'CHI')

ORDER BY name;
```

This request returns the following EXPLAIN report:

```
Explanation
-----
...
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
      Spool 3 an all-rows scan, which is joined to Spool 2 (Last Use).
      and Spool 2 are joined using an exclusion merge join, with a
      join condition of ("Spool_3.Dept_No = Spool_2.Dept_No"). 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.
...

```

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 are sorted by hash code using a uniqueness sort to eliminate duplicate rows. Uniqueness is based on the data in spool field1. The contents of spool field1 depend on the query and might comprise any of the following:

Phrase	Definition
	<ul style="list-style-type: none"> • A concatenation of all the fields in the spool row (used for queries with SELECT DISTINCT or that involve a UNION, INTERSECT, or MINUS operation). • A concatenation of the rowIDs that identify the data rows from which the spool row was formed (used for complex queries involving subqueries). • Some other value that uniquely defines the spool row (used for complex queries involving aggregates and subqueries).
SORT to order Spool 1 by the sort key in spool field1	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 Vantage.

SELECT Example With a Condition Based on a 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 request modifier generates the following response for this request:

```
EXPLAIN
SELECT name, emp_no
FROM employee
WHERE emp_no IN (SELECT emp_no
                  FROM charges)
ORDER BY name;
```

Explanation

```
-----
...
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.
...
```

EXPLAIN Request Modifier and Parallel Steps

The EXPLAIN request modifier also reports whether or not requests 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 multistatement request, a macro, or a solitary statement that affects multiple rows).

Up to 20 parallel steps can be processed at a time 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.

The preceding limits apply to the number of steps that can be simultaneously executed in parallel. Another parallel step can be started once the number of steps that are executing drops below the limit. Therefore, the number of parallel steps listed in an EXPLAIN output is not restricted by the above limits.

Parallel Steps 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 request 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.

```

Implicit Multistatement INSERT Example

In the following BTEQ multistatement request, which is treated as an implicit transaction, the statements are processed concurrently.

In Teradata session mode, the EXPLAIN request 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.

## EXPLAIN Request Modifier and Partitioned Primary Index Access

EXPLAIN reports indicate partition accesses, deletions, joins, and eliminations performed during query optimization.

## Partitioned Primary Index Examples

This set of examples shows some of the more important EXPLAIN text phrases associated with optimizing queries made on tables that have partitioned primary indexes.

The following example demonstrates a partial EXPLAIN report for accessing a subset of row partitions for a SELECT request. 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;
```

...

```
3) We do an all-AMPs RETRIEVE step in TD_MAP1 from 8 partitions of
 mws.t1 with a condition of ("mws.t1.b >= 3") into Spool 1
 (group_amps), which is built locally on the AMPs. The size of
 Spool 1 is estimated with no confidence to be 3 rows (129 bytes).
 The estimated time for this step is 0.15 seconds.
```

...

The following example demonstrates a partial EXPLAIN report for a SELECT request 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 row 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;
```

```
...
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.
...
```

The following example demonstrates a partial EXPLAIN request report for row-partitioned primary index access without any constraints on the partitioning column. The relevant phrase is in boldface type. The report indicates that all row 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 following example demonstrates the processing of a SELECT request without any row 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;
```

...

3) We do an all-AMPs RETRIEVE step in TD\_MAP1 from mws.t1 by way of an all-rows scan with a condition of ("NOT (mws.t1.b IS NULL)") into Spool 1 (group\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 8 rows (344 bytes). The estimated time for this step is 0.15 seconds.

...

Two steps are generated to perform the partial and full row partition deletions, respectively, as demonstrated in the following partial EXPLAIN reports. The relevant phrases appear 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;
```

...

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)").

...

```
EXPLAIN
DELETFROM t2
WHERE b BETWEEN 4 AND 8;
```

...

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)").

...

The following example demonstrates a spool with a row-partitioned primary index and a rowkey-based merge join. The relevant phrases appear 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;
```

...

- 4) We do an all-AMPs RETRIEVE step in TD\_MAP1 from mws.t3 by way of an all-rows scan with a condition of "(NOT (mws.t3.a IS NULL )) AND ((mws.t3.b <= 65535) AND (mws.t3.b >= 1 ))" into Spool 2 (all\_amps), which is built locally on the AMPs. Then we do a **Sort to partition Spool 2 by rowkey**. The size of Spool 2 is estimated with no confidence to be 2 rows (42 bytes). The estimated time for this step is 0.05 seconds.
- 5) We do an all-AMPs JOIN step in TD\_MAP1 from mws.t4 by way of a RowHash match scan, which is joined to Spool 2 (Last Use) by way of a RowHash match scan. mws.t4 and Spool 2 are joined using a **rowkey-based** merge join, with a join condition of "(b = mws.t4.b) AND (a = mws.t4.a)". The result goes 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 (130 bytes). The estimated time for this step is 0.09 seconds.

...

The following example demonstrates one step for joining two tables having the same partitioning and primary keys. The relevant phrases appear 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.
...

```

The following example demonstrates row partition elimination in an aggregation. The relevant phrase appears 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;

```

```

...
3) We do an all-AMPs SUM step in TD_MAP1 to aggregate from 7 partitions of
mws.t1 with a condition of ("mws.t1.b >= 4").
Aggregate Intermediate Results are computed globally, then placed
in Spool 3 in TD_Map1. The size of Spool 3 is estimated with high
confidence to be 1 row (19 bytes). The estimated time for this
step is 0.21 seconds.
4) We do an all-AMPs RETRIEVE step in TD_Map1 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 (32 bytes). The estimated time for
this step is 0.03 seconds.
```

```
...
```

New terminology in these explanations are defined as follows:

| Phrase                                      | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>n</i> partitions of                      | Only <i>n</i> of the row partitions are accessed, where $n > 1$ . In this case, $n = 8$ .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| a single partition of                       | Only one row partition is accessed in processing this request.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| all partitions of                           | All row partitions are accessed for primary index access in processing this request.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| of a single partition                       | The Optimizer determined that all rows in a single row partition can be deleted. In some cases, this allows for faster deletion of the entire partition.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| of <i>n</i> partitions                      | The Optimizer determined that all rows in each of <i>n</i> row partitions can be deleted, where $n > 1$ . In some cases, this allows for faster deletion of entire row partitions.                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| SORT to partition Spool <i>n</i> by rowkey. | The Optimizer determined that a spool is to be row-partitioned based on the same partitioning expression as a table to which the spool is to be joined. That is, the spool is to be sorted by rowkey (partition and hash). Partitioning the spool in this way enables a faster join with the row-partitioned table. <i>n</i> is the spool number.                                                                                                                                                                                                                                                                                                |
| a rowkey-based                              | The join is hash-based by row partition (rowkey). In this case, there are equality constraints on both the partitioning and primary index columns. This enables a faster join since each uneliminated row partition needs to be joined with at most only one other row 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 row-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. |

## EXPLAIN Request Modifier and Column-Partition Access

EXPLAIN reports indicate column partition accesses, deletions, joins, and column partition eliminations performed during query optimization.

### Table Definitions for the Examples

The following set of CREATE TABLE requests defines the tables used in the examples for this topic. All of the tables are created in database *PLS*. You should assume that 20 column partition contexts are available for each of the examples presented.

```
CREATE TABLE t1 (
 a INTEGER,
```

```

b INTEGER,
c INTEGER,
d INTEGER,
e INTEGER,
f INTEGER,
g INTEGER,
h INTEGER,
i INTEGER,
j INTEGER,
k INTEGER,
l INTEGER,
m INTEGER,
n INTEGER,
o INTEGER,
p INTEGER,
q INTEGER,
r INTEGER,
s INTEGER,
t INTEGER,
u INTEGER,
v INTEGER,
w INTEGER,
x INTEGER,
y INTEGER,
z INTEGER)
PARTITION BY COLUMN;

CREATE TABLE t2 AS t1
WITH NO DATA
PRIMARY INDEX (a,b);

CREATE TABLE t3 AS t1
WITH NO DATA;

CREATE TABLE t4 AS t1
WITH NO DATA
NO PRIMARY INDEX
PARTITION BY (COLUMN, RANGE_N(b BETWEEN 1
 AND 10
 EACH 1));

CREATE TABLE t5 (
 a INTEGER,
 b INTEGER,

```

```
c INTEGER,
d INTEGER,
e INTEGER,
f INTEGER,
g INTEGER,
h INTEGER,
i INTEGER,
j INTEGER)
PARTITION BY COLUMN;

CREATE TABLE t6 AS t5
WITH NO DATA
PRIMARY INDEX (a,b);
```

**Selecting a Few Columns from a Column-Partitioned Table**

This example shows part of the EXPLAIN text for selecting 4 of the columns from the column-partitioned table *PLS.t1*.  
The 5 column partitions reported by the EXPLAIN text includes the 4 column partitions specified in the select list of the SELECT request plus the delete column partition.

```
EXPLAIN
SELECT a, b, g, p
FROM t1;

...
3) We do an all-AMPs RETRIEVE step from 5 column partitions of
 PLS.t1 by way of an all-rows scan with no residual conditions
 into Spool 1 (all_amps), which is built locally on the AMPs.
 The size of Spool 1 is estimated with low confidence to be 2 rows
 (614 bytes). The estimated time for this step is 0.01 seconds.
...

```

New terminology in this explanation is defined as follows:

| Phrase                      | Definition                                                                                                                                                                |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| from 5 column partitions of | The 4 requested columns plus the delete column can be accessed from the reported column partitions. There are sufficient contexts to process all five partitions at once. |

**Selecting Many Columns From a Column-Partitioned Table**

This example shows part of the EXPLAIN text for selecting 19 of the 26 columns from the column-partitioned table *PLS.t1*.

The count of 20 for the number of column partitions includes the 19 column partitions specified in the select list of the SELECT request plus the delete column partition.

```
EXPLAIN
SELECT a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,r,s,t
FROM t1;
```

...

3) We do an all-AMPs RETRIEVE step **from 20 column partitions of** PLS.t1 by way of an all-rows scan with no residual conditions into Spool 1(all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 2 rows (614 bytes). The estimated time for this step is 0.01 seconds.

...

### Selecting More Columns From a Column-Partitioned Table Than There Are Contexts Available

This example shows part of the EXPLAIN text for selecting 20 of the 26 columns from the column-partitioned table *PLS.t1*. In this example, there are not enough contexts available to process the request optimally. (For this example, there are 20 contexts available; for other systems, the number of available contexts is usually higher.)

```
EXPLAIN
SELECT a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,r,s,t,u
FROM t1;
```

...

3) We do an all-AMPs RETRIEVE step **from 21 column partitions (20 contexts) of PLS.t1 using covering CP merge Spool 2 (2 subrow partitions and Last Use)** by way of an all-rows scan with no residual conditions into Spool 1 (all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 2 rows (614 bytes). The estimated time for this step is 0.03 seconds.

...

The retrieved 21 column partitions in step 3 includes the 20 selected partitions specified in the select list of the SELECT request and the delete column partition from the column-partitioned table.

21 exceeds the 20 available column partition contexts on this system, so Vantage merges 20 column partitions from the column-partitioned table *PLS.t1*, including the delete column partition, into the first subrow column partition of the CP merge spool. The remaining column partition that needs to be accessed from the column-partitioned table is copied to the second subrow column partition in the CP merge spool for a total of 2 subrow column partitions. This reduces the number of column partitions to be accessed at one

time, which is limited by the number of available column partition contexts. The result is then retrieved from the 2 subrow column partitions of the CP merge spool.

New terminology in this explanation is defined as follows:

| Phrase                                                                                                                      | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| from 21 column partitions (20 contexts) of <i>PLS.t1</i> using covering CP merge Spool 2 (2 subrow partitions and Last Use) | Up to 21 column partitions of <i>PLS.t1</i> might need to be accessed using 20 column partition contexts. A CP merge spool is created and used to merge some column partitions from <i>PLS.t1</i> , then the resulting 2 subrow column partitions are read from the CP merge spool.<br><br>The number of merges needed depends on the number of column partitions that need to be accessed and the number of available column partition contexts as indicated in the preceding from 21 column partitions (20 contexts) phrase. |

### Selecting All Columns From a Column-Partitioned Table

This example shows part of the EXPLAIN text for selecting all of the columns from the column-partitioned table *PLS.t1*.

```
EXPLAIN
SELECT *
FROM t1;
```

...

```
3) We do an all-AMPs RETRIEVE step from 27 column partitions (20
 contexts) of PLS.t1 using covering CP merge Spool 2 (2 subrow
 partitions and Last Use) by way of an all-rows scan with no
 residual conditions into Spool 1 (all_amps), which is built locally
 on the AMPs. The size of Spool 1 is estimated with low confidence
 to be 2 rows (614 bytes). The estimated time for this step is
 0.03 seconds.
```

...

### Selecting All Columns From a Multilevel Column-Partitioned and Row-Partitioned Table

This example shows part of the EXPLAIN text for selecting all of the columns from the multilevel column-partitioned and row-partitioned table *PLS.t4*.

This is similar to the previous example except only two 2 partitions of the mixed column- and row-partitioned table need to be read. Because there are 27 column partitions to be accessed and 2 row partitions, there are 52 combined partitions to be accessed: 2 row partitions from each of the 27 column partitions.

```
EXPLAIN
SELECT *
```

```
FROM t4
WHERE b BETWEEN 4 AND 5;
```

```
...
3) We do an all-AMPs RETRIEVE step from 52 combined partitions (27
 column partitions and 20 contexts) of PLS.t4 using covering CP
 merge Spool 2 (2 subrow partitions and Last Use) with a condition
 of "(PLS.t4.b <= 5) AND (PLS.t4.b >= 4)" 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 (307 bytes). The
 estimated time for this step is 0.03 seconds.
...
```

New terminology in this explanation is defined as follows:

| Phrase                                                                                                                                          | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| from 52 combined partitions (27 column partitions and 20 contexts) of PLS.t4 using covering CP merge Spool 2 (2 subrow partitions and Last Use) | <p>This phrase indicates that the rows and columns for up to 52 combined partitions of table <i>t4</i> might need to be accessed. For the column-partitioning level, 27 column partitions may need to be accessed.</p> <p>For this phrase to occur, the column-partitioned table or join index would have both row and column partitioning, as <i>t4</i> does</p> <p>For the column-partitioning level, 20 column partition contexts are used to access up to 27 column partitions.</p> <p>52 combined partitions and 27 column partitions are both greater than 2, as is required for this phrase to be reported.</p> <p>20 contexts is greater than 2 and less than 27-1, or 26, as is required for this phrase to be reported.</p> <p>One of the column partitions that might be accessed is the delete column partition. Not all of the m1 column partitions might need to be accessed if no rows qualify. The phrase <b>using covering CP merge Spool</b> follows the table name, which is <i>t4</i>.</p> |

**Selecting Multiple Columns From a Multilevel Column- and Row-Partitioned Table**

This example shows part of the EXPLAIN text for selecting 20 of the 26 columns from the multilevel column-partitioned and row-partitioned table *PLS.t4*.

```
EXPLAIN
SELECT a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,r,s,t,u
FROM t4
WHERE b BETWEEN 4 AND 5;
```

```
...
3) We do an all-AMPs RETRIEVE step from 42 combined partitions (21
 column partitions and 20 contexts) of PLS.t4 using covering CP
 merge Spool 2 (2 subrow partitions and Last Use) with a
```

```
condition of ("PLS.t4.b <= 5) AND (PLS.t4.b >= 4)") 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 (307 bytes).
The estimated time for this step is 0.03 seconds.
...
```

### Selecting Columns From a Multilevel Column- and Row-Partitioned Table

This example shows part of the EXPLAIN text for selecting 19 of the 26 columns from the multilevel column-partitioned and row-partitioned table *PLS.t4*.

```
EXPLAIN
SELECT a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,r,s,t
FROM t4
WHERE b BETWEEN 4 AND 5;
```

```
...
3) We do an all-AMPs RETRIEVE step from 40 combined partitions (20
column partitions) of PLS.t4 with a condition of
("PLS.t4.b <= 5)AND (PLS.t4.b >= 4)") into Spool 1(all_amps),
which is builtlocally on the AMPs. The size of Spool 1 is estimated
with noconfidence to be 1 row (307 bytes). The estimated
time for this step is 0.03 seconds.
...
```

### Selecting 2 Columns From a Column-Partitioned Table With a Predicate Specified for 26 Columns in the WHERE Clause

This example shows part of the EXPLAIN text for selecting 2 columns from the column-partitioned table *PLS.t1* with a WHERE clause condition specified on each of the 26 columns from the table.

```
EXPLAIN
SELECT a, q
FROM t1
WHERE a>1
AND b<=1000
AND c=10
AND d=11
AND e<40
AND f<>87
AND g=92
AND h>=101
AND i=3000
AND j>=5
AND k=12
```

```

AND l=0
AND m=0
AND n IS NOT NULL
AND o=1
AND p=-1
AND q=1
AND r<10
AND s=9
AND t>33
AND u=0
AND v=0
AND w=0
AND x>101
AND y=0
AND z=0;

```

...

3) We do an all-AMPs RETRIEVE step from 2 column partitions of *PLS.t1* using rowid Spool 2 (Last Use) built from 27 column partitions (20 contexts and Last use) with a condition of

```

(" (PLS.t1.a > 1) AND (PLS.t1.b <= 1000) AND (PLS.t1.c = 10)
AND (PLS.t1.d = 11) AND (PLS.t1.e < 40) AND (PLS.t1.f <> 87)
AND (PLS.t1.g = 92) AND (PLS.t1.h >= 101) AND
(PLS.t1.i = 3000) AND (PLS.t1.j >= 5) AND (PLS.t1.k = 12) AND
(PLS.t1.l = 0) AND (PLS.t1.m = 0) AND (PLS.t1.n IS NOT NULL)
AND (PLS.t1.o = 1) AND (PLS.t1.p = -1) AND (PLS.t1.q = 1) AND
(PLS.t1.r < 10) AND (PLS.t1.s = 9) AND (PLS.t1.t > 33) AND
(PLS.t1.u = 0) AND (PLS.t1.v = 0) AND (PLS.t1.w = 0) AND
(PLS.t1.x > 101) AND (PLS.t1.y = 0) AND
(PLS.t1.z = 0) ") 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 (307 bytes). The estimated time for this step is
0.03 seconds.

```

...

New terminology in this explanation is defined as follows:

| Phrase                                                                                                                       | Definition                                                                                                                                                                                                                                                                                                     |
|------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| from 2 partitions of <i>PLS.t1</i> using rowid Spool 2 (Last Use) built from 27 column partitions (20 contexts and Last use) | Up to 2 column partitions of <i>PLS.t1</i> might need to be accessed. There are column partition contexts available for each of the 2 column partitions. A rowID spool is created that contains the rowids of rows in the table or join index that qualify and then the column-partitioned table or join index |



| Phrase | Definition                                                                                                                                             |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
|        | is read driven by this rowid spool. This is the last usage of this rowid spool so it can be deleted. <i>k</i> is the spool number for the rowid spool. |

### Selecting 21 Columns From a Column-Partitioned Table With a Predicate Specified for 26 Columns in the WHERE Clause

This example shows part of the EXPLAIN text for selecting 21 of the 26 columns from a column-partitioned table with a predicate specified for each of those columns in the WHERE clause.

```

EXPLAIN
SELECT a,b,c,d,e,f,g,h,i,j,k,l,m,o,p,q,r,s,t,u,v
FROM t1
WHERE a>1
AND b<=1000
AND c=10
AND d=11
AND e<40
AND f<>87
AND g=92
AND h>=101
AND i=3000
AND j>=5
AND k=12
AND l=0
AND m=0
AND n IS NOT NULL
AND o=1
AND p=-1
AND q=1
AND r<10
AND s=9
AND t>33
AND u=0
AND v=0
AND w=0
AND x>101
AND y=0
AND z=0;

```

...

3) We do an all-AMPs RETRIEVE step from 21 column partitions of PLS.t1 using covering CP merge Spool 3 (20 contexts and Last Use) and rowid Spool 2

(Last Use) built from 27 column partitions (20 contexts and Last use) with a condition of "(PLS.t1.a > 1) AND (PLS.t1.b <= 1000) AND (PLS.t1.c = 10) AND (PLS.t1.d = 11) AND (PLS.t1.e < 40) AND (PLS.t1.f <> 87) AND (PLS.t1.g = 92) AND (PLS.t1.h >= 101) AND (PLS.t1.i = 3000) AND (PLS.t1.j >= 5) AND (PLS.t1.k = 12) AND (PLS.t1.l = 0) AND (PLS.t1.m = 0) AND (PLS.t1.n IS NOT NULL) AND (PLS.t1.o = 1) AND (PLS.t1.p = -1) AND (PLS.t1.q = 1) AND (PLS.t1.r < 10) AND (PLS.t1.s = 9) AND (PLS.t1.t > 33) AND (PLS.t1.u = 0) AND (PLS.t1.v = 0) AND (PLS.t1.w = 0) AND (PLS.t1.x > 101) AND (PLS.t1.y = 0) AND (PLS.t1.z = 0)" 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 (307 bytes). The estimated time for this step is 0.03 seconds.

...

New terminology in this explanation is defined as follows:

| Phrase                                                                                                                                                                                        | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| from 21 column partitions of <i>PLS.t1</i> using covering CP merge Spool 3 (20 contexts and Last Use) and rowid Spool 2 (Last Use) built from 27 column partitions (20 contexts and Last use) | Up to 21 column partitions of <i>PLS.t1</i> might need to be accessed. There are column partition contexts available for each of the 2 column partitions. A column-partitioned merge spool is created and used to merge column partitions from <i>PLS.t1</i> and then the resulting column partitions are read from the column-partitioned merge spool.<br>A rowID spool is created that contains the rowIDs of rows in <i>PLS.t1</i> that qualify and then the column-partitioned table is read driven by this rowID spool. This is the last usage of this rowID spool so it can be deleted. |

### INSERT ... SELECT from Nonpartitioned Table with Primary Index into Column-Partitioned Table Using Default Locking

This example shows part of the EXPLAIN text for inserting all of the columns from the nonpartitioned, primary-indexed table *PLS.t6* into the column-partitioned table *PLS.t5* using the default locking for *PLS.t6*.

```
EXPLAIN INSERT INTO t5
SELECT *
FROM t6;
```

...

- 4) We do an all-AMPs **MERGE into 10 column partitions of** *PLS.t5* from *PLS.t6*. The size is estimated with no confidence to be 2 rows. The estimated time for this step is 0.71 seconds.
- 5) We spoil the parser's dictionary cache for the table.

...

The 10 column partitions are the default user-specified column partitions from *PLS.t6*, whose definition is copied from *PLS.t5*, but is neither column-partitioned nor row-partitioned and has a primary index defined on columns *a* and *b*. The delete column partition is not affected.

New terminology in this explanation is defined as follows:

| Phrase                             | Definition                                                                                                                                 |
|------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| MERGE into 10 column partitions of | The rows and columns of table <i>PLS.t6</i> (which has column partitioning in this case) for up to 10 combined partitions can be accessed. |

### **INSERT ... SELECT from Nonpartitioned Table With Primary Index into Column-Partitioned Table Using LOCKING Modifier**

This example shows part of the EXPLAIN text for using an INSERT ... SELECT request to insert all of the columns from the non-column-partitioned table *PLS.t6* into the column-partitioned table *PLS.t5* while using a LOCKING request modifier to lock *PLS.t6* for ACCESS.

```
EXPLAIN LOCKING t6 FOR ACCESS
INSERT INTO t5
 SELECT *
 FROM t6;
```

- 1) First, we lock a distinct PLS."pseudo table" for write on a RowHash to prevent global deadlock for PLS.t5.
  - 2) Next, we lock a distinct PLS."pseudo table" for access on a RowHash to prevent global deadlock for PLS.t6.
  - 3) We lock PLS.t5 for write, and we lock PLS.t6 for access.
  - 4) We do an all-AMPs **MERGE into 10 column partitions of** PLS.t5 from PLS.t6. The size is estimated with no confidence to be 2 rows. The estimated time for this step is 0.71 seconds.
  - 5) We spoil the parser's dictionary cache for the table.
  - 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.  
The total estimated time is 0.72 seconds.

### **INSERT ... SELECT from Nonpartitioned Source Table with Primary Index into Column-Partitioned Target Table**

This example shows EXPLAIN text for an INSERT ... SELECT request that inserts all of the columns from the primary-indexed, nonpartitioned table *PLS.t2* into the column-partitioned table *PLS.t1*.

```
EXPLAIN INSERT INTO t1
 SELECT *
 FROM t2;
```

```
...
4) We do an all-AMPs MERGE into 26 column partitions (20 contexts) of
 PLS.t1 from PLS.t2. The size is estimated with no
 confidence to be 2 rows. The estimated time for this
 step is 0.71 seconds.
5) We spoil the parser's dictionary cache for the table.
...
```

New terminology in this explanation is defined as follows:

| Phrase                                           | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MERGE into 26 column partitions (20 contexts) of | <p>The columns of table <i>PLS.t1</i> (which has column partitioning in this case) can be accessed.</p> <p>20 contexts are used to access the 26 combined partitions. Because the available 20 contexts are not enough to accommodate 26 combined partitions, there must be 2 scans of source table <i>t2</i>.</p> <p>Merge 20 columns into <i>t1</i>.</p> <p>Merge the remaining 6 columns into <i>t1</i>.</p> <p>Because there is a READ lock on <i>t2</i>, the rows read in each scan are the same.</p> |

### **INSERT ... SELECT from Nonpartitioned Source Table with Primary Index and Locking Request into Column-partitioned Target Table**

This example is similar to the previous example except that it specifies a LOCKING request modifier on the source table. The example shows EXPLAIN text for inserting all of the columns from the primary-indexed, nonpartitioned table *PLS.t2* into the column-partitioned table *PLS.t1*. In this case, the source table *PLS.t2* must be spooled before its rows can be inserted into the target column-partitioned table, *PLS.t1*.

Also note that if table *t2* had been read twice as was done in the previous example, the rows could change from the first scan to the second and corrupt table *t1*.

```
EXPLAIN LOCKING t2 FOR ACCESS
INSERT INTO t1
SELECT *
FROM t2;
```

```
...
4) We do an all-AMPs RETRIEVE step from PLS.t2 by way of an all-rows
 scan with no residual conditions into Spool 1 (all_amps), which is
 spooled locally on the AMPs. The size of Spool 1 is estimated with
 low confidence to be 2 rows (614 bytes). The estimated time for this
 step is 0.01 seconds.
5) We do an all-AMPs MERGE into 26 column partitions (20 contexts) of PLS.t1
 from Spool 1 (Last Use). The size
 is estimated with low confidence to be 2 rows. The estimated
```

```

 time for this step is 0.71 seconds.
6) We spoil the parser's dictionary cache for the table.
...

```

### INSERT ... SELECT from Column-Partitioned Source Table into Column-Partitioned Target Table

This example shows part of the EXPLAIN text for inserting all of the columns from the column-partitioned table *PLS.t3* into the column-partitioned table *PLS.t1*.

```

EXPLAIN INSERT INTO t1
SELECT *
FROM t3;

...
4) We do an all-AMPs RETRIEVE step from 27 column partitions
 (20 contexts) of PLS.t3 using CP merge Spool 2 by way of an
 all-rows scan with no residual conditions into Spool 1 (all_amps),
 which is spooled locally on the AMPs. The size of Spool 1
 is estimated with low confidence to be 2 rows (614 bytes).
 The estimated time for this step is 0.01 seconds.
5) We do an all-AMPs MERGE into 26 column partitions (20 contexts) of
 PLS.t1 from Spool 1 (Last Use). The size is estimated with no
 confidence to be 2 rows. The estimated time for this step is 0.71
 seconds.
5) We spoil the parser's dictionary cache for the table.
...

```

This example does not introduce any new EXPLAIN text terminology.

### Inserted Column Data from VALUES Clause

This example shows part of the EXPLAIN text for a single-row insert with values into the column-partitioned table *PLS.t1*. Because multiple column partition contexts are not needed for this INSERT request, the EXPLAIN text only specifies the number of column partitions.

```

EXPLAIN
INSERT INTO t1 VALUES (1, 2,,,,,,,,,,,,,,,,,,,,,,,,,,,,);

1) First, we do an INSERT into 26 column partitions of PLS.t1.
 The estimated time for this step is 0.03 seconds.
...

```

New terminology in this explanation is defined as follows:

| Phrase                                            | Definition                                                        |
|---------------------------------------------------|-------------------------------------------------------------------|
| INSERT into 26 column partitions of <i>PLS.t1</i> | One row is inserted into <i>PLS.t1</i> with 26 column partitions. |

### Inserted Column Data from USING Request Modifier

This example shows part of the EXPLAIN text for a single-row insert with values taken from a USING request modifier. Apart from the data being inserted into *PLS.t1* coming from a USING request modifier via host variables or parameters in the VALUES clause rather than being explicitly specified in the VALUES clause, the EXPLAIN text and explanation for this example are identical to the EXPLAIN text and explanation for the previous example.

```
EXPLAIN
USING (a INTEGER, b INTEGER, c INTEGER, d INTEGER,
 e INTEGER, f INTEGER, g INTEGER, h INTEGER,
 i INTEGER, j INTEGER, k INTEGER, l INTEGER,
 m INTEGER, n INTEGER, o INTEGER, p INTEGER,
 q INTEGER, r INTEGER, s INTEGER, t INTEGER,
 u INTEGER, v INTEGER, w INTEGER, x INTEGER,
 y INTEGER, z INTEGER)
INSERT INTO t1 VALUES (:a, :b, :c, :d, :e, :f, :g, :h, :i,
 :j, :k, :l, :m, :n, :o, :p, :q, :r,
 :s, :t, :u, :v, :w, :x, :y, :z);
```

1) First, we do an **INSERT into 26 column partitions of**  
*PLS.t1*. The estimated time for this step is 1.72 seconds....

## EXPLAIN Request Modifier and MERGE Conditional Steps

The EXPLAIN request modifier is useful in determining the conditional steps in MERGE processing. 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.

### 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);

```

### Simple Conditional Insert Processing Without Trigger or Join Index Steps

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);

```

- 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).
...

```

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).

### More Complex Conditional Insert Processing Without Trigger Steps

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 THENINSERT(4,5);

```

```

...
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).
...

```

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.

### Conditional Insert Processing With an Upsert Trigger

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);
```

```
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.
...

```

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.

### Conditional Insert Processing With an Abort Trigger

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);

```

- 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.
- ...

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 Request Modifier Terminology for These Examples

New terminology in this set of EXPLAIN reports is defined as follows:

| Phrase                            | Definition                                                                                                                                                 |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| if no update in<br><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><br>not executed, | Indicates the action taken if the first step in an UPDATE block is not performed.<br>Reported only if <step number> is greater than 1.                     |

## EXPLAIN and UPDATE (Upsert Form) Conditional Steps

The EXPLAIN modifier is useful in determining the conditional steps in UPDATE (Upsert Form) processing. Atomic upsert conditional steps are insert operations that are performed after an unconditional update operation does not meet its matching condition.

### Database Object DDL for Examples

The following DDL statements create the tables accessed by the explained DML request examples.

```
CREATE TABLE t1 (x1 INTEGER, y1 INTEGER);

CREATE TABLE t2 (x2 INTEGER NOT NULL, y2 INTEGER NOT NULL);
```

### Simple Conditional Upsert Processing Without Trigger or Join Index Steps

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);
```

```
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.
...

```

Both the unconditional update attempt and the conditional insert are combined in a single step.

### More Complex Upsert Processing With Join Index Steps

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);
```

A portion of the EXPLAIN output for the UPDATE is below:

```
EXPLAIN
UPDATE t1
SET y1 = 3 WHERE x1 = 2
ELSE INSERT t1(2, 3);
```

...

- 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).
...
```

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.

### Upsert With a Simple Trigger

The following upsert statement involves a simple trigger. The relevant phrases in the EXPLAIN report appear 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));
```

A portion of the EXPLAIN output is below:

```
EXPLAIN
UPDATE t1
SET y1 = 3 WHERE x1 = 2
ELSE INSERT t1(2, 3);
```

```
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.
...
```

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.

### Upsert With an Abort Trigger

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 appear in boldface type.

```
EXPLAIN
UPDATE t4
SET y4 = 3 WHERE x4 = 2
ELSE INSERT t4(2, 3);
```

- 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.
- ...

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.

### Complicated Upsert With Multiple Triggers

This example uses the 5 tables defined by the following DDL table definition requests:

```
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);
```

The relevant phrases in the EXPLAIN report appear in boldface type.

- 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.
...

```

This EXPLAIN report is more complicated because of triggers r6 through 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 through r10.

### Complicated Upsert That Disables Triggers r6 Through r10

This example disables the triggers r6 through r10 from the previous example and invokes the following newly defined triggers. The DDL requests 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);
```

The relevant phrases in the EXPLAIN report are highlighted in boldface type.

- 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.
...
```

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.

### Complicated Upsert That Disables Triggers r11 Through r15

This example disables the triggers r11 through r15 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 request:

```
EXPLAIN
UPDATE t1
```

```
SET y1 = 20 WHERE x1 = 30
ELSE INSERT t1(30, 20);
```

The relevant phrases in the EXPLAIN report appear in boldface type.

- 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.
- ...

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.

### Complicated Upsert With Different Triggers Than the Previous Example

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 the previous example, 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);

```

The relevant phrases in the EXPLAIN report appear in boldface type.

- 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.
...

```

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.

### Complicated Upsert With Multiple Triggers From Previous Examples

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);

```

The relevant phrases in the EXPLAIN report appear in boldface type.

- 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.
- ...



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.

### EXPLAIN Request Modifier Terminology for These Examples

New terminology in this set of EXPLAIN reports is defined as follows:

| Phrase                            | Definition                                                                                                                                                 |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| if no update in<br><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><br>not executed, | Indicates the action taken if the first step in an UPDATE block is not performed.<br>Reported only if <step number> is greater than 1.                     |

## EXPLAIN Request Modifier and Triggers

### Row Trigger with Looping Trigger Action Example

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;
```

- 1) First, we lock mws.t3 in TD\_MAP1 for write on a reserved RowHash to prevent global deadlock.
- 2) Next, we lock mws.t2 in TD\_MAP1 for write on a reserved RowHash to prevent global deadlock.
- 3) We lock mws.t1 in TD\_MAP1 for write on a reserved RowHash to prevent global deadlock.
- 4) We lock mws.t3 in TD\_MAP1 for write, we lock mws.t2 in TD\_MAP1 for write, and we lock mws.t1 in TD\_MAP1 for write.
- 5) We do an all-AMPs MERGE step in TD\_MAP1 into mws.t1 from mws.t3 followed by an insert in Spool 1. The size is estimated with low confidence to be 8 rows. The estimated time for this step is 3.94 seconds.
- 6) **<BEGIN ROW TRIGGER LOOP>**

we do an all-AMPs UPDATE step in TD\_MAP1 from mws.t2 by way of an all-rows scan with no residual conditions. The size is estimated with low confidence to be 8 rows. The estimated time for this step is 0.09 seconds.

- 7) We do an all-AMPs DELETE step in TD\_MAP1 from mws.t2 by way of an all-rows scan with no residual conditions. The size is estimated with low confidence to be 8 rows. The estimated time for this step is 2.44 seconds.
- 8) We do an all-AMPs DELETE step in TD\_MAP1 from mws.t3 by way of an all-rows scan with no residual conditions. The size is estimated with low confidence to be 8 rows. The estimated time for this step is 2.44 seconds.

**<END ROW TRIGGER LOOP>** for step 6.

- 9) We spoil the parser's dictionary cache for the table.
- 10) We spoil the parser's dictionary cache for the table.
- 11) We spoil the parser's dictionary cache for the table.
- ...

## EXPLAIN Request Modifier and Recursion

### Recursive SELECT Example

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:

```
...
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.
...

```

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 pool 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.

## STATIC EXPLAIN

### About Static Query Plans

A query plan that is not generated by IPE is referred to as a static query plan. Static query plans are the query plans that are traditionally returned for an explained request.

Sometimes the Optimizer generates a static query plan for a request that is eligible for IPE. The following circumstances can cause a static query plan to be generated when the request being explained is eligible for IPE.

- The request is eligible for IPE, but does not satisfy certain system-controlled threshold criteria.

In this case, Vantage precedes the reported steps with the following phrases: "This request is eligible for incremental planning and execution (IPE) but doesn't meet thresholds. The following is the static plan for the request."

- The request is eligible for IPE, and if it is executed, the Optimizer might generate a dynamic plan instead of the reported static plan, but Vantage uses a static plan or summary information from static plan to evaluate workload filters, throttles, and classification criteria.

In this case, Vantage precedes the reported steps with the following phrase: “This request is eligible for incremental planning and execution (IPE). The following is the static plan for the request.”

- The request is eligible for IPE, but IPE is disabled at the site.

In this case, Vantage reports the static query plan with the following phrase:

```
This request is eligible for incremental planning and execution (IPE). IPE is
disabled. The following is the static plan for the request.
```

## Requesting a Static EXPLAIN Report

To ask for a static EXPLAIN report for a request, specify the keyword **STATIC** preceding the keyword **EXPLAIN**.

The default for an EXPLAIN request modifier is **STATIC**, so you will generate a static EXPLAIN report at most sites, regardless of whether you precede EXPLAIN with the keyword **STATIC**.

You can request that the report be returned in XML format by specifying the keywords **IN XML** following the EXPLAIN keyword.

For more information about the EXPLAIN request modifier, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## STATIC EXPLAIN Report When the Request Is Eligible for Incremental Planning and Execution

This example assumes that your system has been set never to report a dynamic plan. When this is the case, Vantage the static plan with an indication that the request is eligible for IPE. In this case, the **STATIC** keyword is implicit and does not need to be specified as part of the EXPLAIN request modifier.

```
STATIC EXPLAIN
SELECT *
FROM t1, t2
WHERE t1.a1 < (SELECT MAX(t3.a3)
 FROM t3)
AND t1.a1 = t2.a2;
```

This shows a portion of the EXPLAIN output:

Explanation

```

This request is eligible for incremental planning and execution (IPE). The
following is the static plan for the request.
```

```
...
```

- 5) We do an all-AMPs SUM step to aggregate from OPTBASE\_PCTDB.t3 by way of an all-rows scan with no residual conditions. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (19 bytes). The estimated time for this step is 0.02 seconds.
  - 6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.00 seconds.
  - 7) We do an all-AMPs DISPATCHER RETRIEVE step from Spool 1 (Last Use) by way of an all-rows scan and send the rows back to the Dispatcher.
  - 8) We do an all-AMPs JOIN step from OPTBASE\_PCTDB.t1 by way of a RowHash match scan with a condition of ("OPTBASE\_PCTDB.t1.a1 < :%SSQ20"), which is joined to OPTBASE\_PCTDB.t2 by way of a RowHash match scan with no residual conditions. OPTBASE\_PCTDB.t1 and OPTBASE\_PCTDB.t2 are joined using a sliding-window merge join, with a join condition of ("OPTBASE\_PCTDB.t1.a1 = OPTBASE\_PCTDB.t2.a2"). The input tables OPTBASE\_PCTDB.t1 and OPTBASE\_PCTDB.t2 will not be cached in memory, but OPTBASE\_PCTDB.t1 is eligible for synchronized scanning. The result goes into Spool 5 (group\_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with no confidence to be 33,333,334 rows (2,166,666,710 bytes). The estimated time for this step is 1 minute and 10 seconds.
- ...

### **STATIC EXPLAIN Where the Request Is Eligible for Incremental Planning and Execution But Does Not Meet Thresholds**

This example demonstrates an EXPLAIN report where the request being explained is eligible for IPE, but does not meet certain system-determined thresholds for execution costs.

The example assumes that your system has been set never to report a dynamic plan. When this is the case, Vantage reports the static plan with an indication that the request is eligible for IPE. In this case, the STATIC keyword is implicit and does not need to be specified as part of the EXPLAIN request modifier.

```

STATIC EXPLAIN
SELECT *
FROM t1, t2
WHERE t1.a1 < (SELECT MAX(t3.a3)
 FROM t3)
AND t1.a1 = t2.a2;

```

This shows a portion of the EXPLAIN output:

#### Explanation

**This request is eligible for incremental planning and execution (IPE) but doesn't meet thresholds. The following is the static plan for the request.**

...

- 5) We do an all-AMPs SUM step to aggregate from OPTBASE\_PCTDB.t3 by way of an all-rows scan with no residual conditions. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (19 bytes). The estimated time for this step is 0.02 seconds.
- 6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.00 seconds.
- 7) We do an all-AMPs DISPATCHER RETRIEVE step from Spool 1 (Last Use) by way of an all-rows scan and send the rows back to the Dispatcher.
- 8) We do an all-AMPs JOIN step from OPTBASE\_PCTDB.t2 by way of a RowHash match scan, which is joined to OPTBASE\_PCTDB.t1 by way of a RowHash match scan with a condition of ("OPTBASE\_PCTDB.t1.a1 < :%SSQ20"). OPTBASE\_PCTDB.t2 and OPTBASE\_PCTDB.t1 are joined using a merge join, with a join condition of ("OPTBASE\_PCTDB.t1.a1 = OPTBASE\_PCTDB.t2.a2"). The result goes into Spool 5 (group\_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with no confidence to be 4 rows (260 bytes). The estimated time for this step is 0.02 seconds.

...

#### **STATIC EXPLAIN Where the Request is Eligible for IPE But IPE is Disabled**

When you submit a STATIC EXPLAIN for a request that is eligible for IPE but IPE is disabled, the reported text is identical to that returned if you had just submitted EXPLAIN except for the additional sentences that appear in boldface before the first step.

```

STATIC EXPLAIN
SELECT *
FROM t1, t2
WHERE t1.a1 < (SELECT MAX(t3.a3)
 FROM t3)
AND t1.a1 = t2.a2;
```

This shows a portion of the EXPLAIN output:

#### Explanation

-----  
**This request is eligible for incremental planning and execution (IPE). IPE is disabled. The following is the static plan for the request.**

...

- 5) We do an all-AMPs SUM step to aggregate from OPTBASE\_PCTDB.t3 by way of an all-rows scan with no residual conditions. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (19 bytes). The estimated time for this step is 0.02 seconds.
- 6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.00 seconds.
- 7) We do an all-AMPs DISPATCHER RETRIEVE step from Spool 1 (Last Use) by way of an all-rows scan and send the rows back to the Dispatcher.
- 8) We do an all-AMPs JOIN step from OPTBASE\_PCTDB.t2 by way of a RowHash match scan, which is joined to OPTBASE\_PCTDB.t1 by way of a RowHash match scan with a condition of ("OPTBASE\_PCTDB.t1.a1 < :%SSQ20"). OPTBASE\_PCTDB.t2 and OPTBASE\_PCTDB.t1 are joined using a merge join, with a join condition of ("OPTBASE\_PCTDB.t1.a1 = OPTBASE\_PCTDB.t2.a2"). The result goes into Spool 5 (group\_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with no confidence to be 4 rows (260 bytes). The estimated time for this step is 0.02 seconds.

...

#### **STATIC EXPLAIN Where the Request Is Not Eligible for Incremental Planning and Execution**

This example demonstrates an EXPLAIN report where the request being explained is not eligible for IPE.

The example assumes that your system has been set never to report a dynamic plan. For this example, the request is not eligible for IPE, so Vantage does not report anything about the eligibility of the request for IPE. For this example, the STATIC keyword is implicit and does not need to be specified as part of the EXPLAIN request modifier.

#### **Note:**

Because this request is not eligible for IPE, the report is identical to the report generated for this query when IPE is disabled.



```

STATIC EXPLAIN
SELECT *
FROM t1
WHERE a3 > 10;

```

Explanation

-----

...

3) We do an all-AMPs RETRIEVE step from OPTBASE\_PCTDB.t1 by way of an all-rows scan with a condition of ("OPTBASE\_PCTDB.t1.a3 > 10") 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 (54 bytes). The estimated time for this step is 0.03 seconds.

...

## DYNAMIC EXPLAIN

A query plan generated by IPE is referred to as a dynamic query plan. Dynamic query plans are not the query plans that are traditionally returned for an explained request.

IPE provides feedback of statistical information and results from the execution of a plan fragment. This enables many optimizations such as partition elimination (see [Row Partition Elimination](#) and [Column Partition Elimination](#)) and sparse join index qualification, and also enables many query rewrites such as transitive closure, predicate simplification, and unsatisfiability (see [Query Rewrite, Statistics, and Optimization](#) for more information about the various query rewrite methods).

### Requesting a Dynamic EXPLAIN Report

To ask for a dynamic EXPLAIN report for a request, specify the keyword DYNAMIC preceding the keyword EXPLAIN. The default for an EXPLAIN request modifier is STATIC, so you must specify DYNAMIC at most sites to return a dynamic EXPLAIN report.

You can request that the report be returned in XML format by specifying the keywords IN XML following the EXPLAIN keyword. The actual plan used when the query is executed may differ from the EXPLAIN output due to changes in the data from the time the EXPLAIN was run.

For more information about the EXPLAIN request modifier, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Alternatively, use DBQL EXPLAIN plan logging, including DBQL XMLPlan logging. This provides the actual plan used for execution of a query. DBQL does not log requests preceded by any form of the EXPLAIN request modifier, though it does log additional information for requests that are optimized to use a dynamic query plan, such as the number of plan fragments and the plan fragment to which the current step belongs.

## Security Issues for Dynamic Query Plan EXPLAIN Reports

IPE produces a dynamic query plan by inserting the intermediate results from the execution of the plan fragments into the main query. A dynamic query plan processes the inserted request fragment constants as elements of the query predicates. You might not want to reveal intermediate results in the EXPLAIN report to end users who might not have the authorization to view them. For example, an end user might have SELECT access to a view that can retrieve data from many other views or tables for which the user does not have SELECT access.

If your system is set to mask intermediate dynamic query plan results (this is the default), Vantage masks the intermediate result values in EXPLAIN reports by replacing them with the character string `.*`. See [EXPLAIN Request Modifier Phrase Terminology](#) for further information about the `.*` character string and its meaning.

### DYNAMIC EXPLAIN for an Unmasked Dynamic Query Plan

This example assumes that your system has been set to allow an unmasked dynamic plan to be displayed. For unmasked dynamic plans, the word *unmasked* signifies that text reporting intermediate results has not been removed from the report. In this report, the unmasked text is the values **10000** and **9999** (appearing in boldface) in the conditions ("OPTBASE\_PCTDB.t1.a1 < 10000") and ("OPTBASE\_PCTDB.t2.a2 <= 9999") in step 9.

```
DYNAMIC EXPLAIN
SELECT *
FROM t1, t2
WHERE t1.a1 < (SELECT MAX(t3.a3)
 FROM t3)
AND t1.a1 = t2.a2;
```

#### Explanation

-----  
The following is the dynamic explain for the request.

...

- 5) We do an all-AMPs SUM step to aggregate from OPTBASE\_PCTDB.t3 by way of an all-rows scan with no residual conditions. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (19 bytes). The estimated time for this step is 0.02 seconds.
- 6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.00 seconds.
- 7) We do an all-AMPs FEEDBACK RETRIEVE step from Spool 1 (Last Use) The size is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.01 seconds. The actual size

```

of Spool 1 is 1 row (25 bytes.)
8) We send an END PLAN FRAGMENT step for plan fragment 1.
9) We do an all-AMPs JOIN step from 9999 partitions of
 OPTBASE_PCTDB.t1 by way of a RowHash match scan with a condition
 of ("OPTBASE_PCTDB.t1.a1 < 10000"), which is joined to
 OPTBASE_PCTDB.t2 by way of a RowHash match scan with a condition
 of ("OPTBASE_PCTDB.t2.a2 <= 9999"). OPTBASE_PCTDB.t1 and
 OPTBASE_PCTDB.t2 are joined using a sliding-window merge join,
 with a join condition of ("OPTBASE_PCTDB.t1.a1 =
 OPTBASE_PCTDB.t2.a2"). The input tables OPTBASE_PCTDB.t1 and
 OPTBASE_PCTDB.t2 will not be cached in memory, but
 OPTBASE_PCTDB.t1 is eligible for synchronized scanning. The
 result goes into Spool 1 (group_amps), which is built locally on
 the AMPs. The size of Spool 1 is estimated with low confidence to
 be 9,999 rows (649,935 bytes). The estimated time for this step
 is 8.24 seconds.
...

```

### DYNAMIC EXPLAIN for a Masked Dynamic Query Plan

This example assumes the default is to allow a masked dynamic plan to be reported. For masked dynamic plans, the word *masked* signifies that text reporting intermediate results has been removed from the report. This is controlled by DBS Control field settings.

Masked values are indicated by the character string `:*` in the EXPLAIN text, as you can see in Step 9 of this dynamic plan. This character string replaces the masked values **10000** and **9999** seen in the previous example.

```

DYNAMIC EXPLAIN
SELECT *
FROM t1, t2
WHERE t1.a1 < (SELECT MAX(t3.a3)
 FROM t3)
AND t1.a1 = t2.a2;

```

Explanation

-----  
The following is the dynamic explain for the request.

```

...
5) We do an all-AMPs SUM step to aggregate from OPTBASE_PCTDB.t3 by
 way of an all-rows scan with no residual conditions. Aggregate
 Intermediate Results are computed globally, then placed in Spool 3.
 The size of Spool 3 is estimated with high confidence to be 1 row
 (19 bytes). The estimated time for this step is 0.02 seconds.
6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of

```

an all-rows scan into Spool 1 (all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.00 seconds.

7) We do an all-AMPs FEEDBACK RETRIEVE step from Spool 1 (Last Use). The size is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.01 seconds. The actual size of Spool 1 is 1 row (25 bytes).

8) **We send an END PLAN FRAGMENT step for plan fragment 1.**

9) We do an all-AMPs JOIN step from 9999 partitions of OPTBASE\_PCTDB.t1 by way of a RowHash match scan with a condition of ("OPTBASE\_PCTDB.t1.a1 < :\*"), which is joined to OPTBASE\_PCTDB.t2 by way of a RowHash match scan with a condition of ("OPTBASE\_PCTDB.t2.a2 <= :\*"). OPTBASE\_PCTDB.t1 and OPTBASE\_PCTDB.t2 are joined using a sliding-window merge join, with a join condition of ("OPTBASE\_PCTDB.t1.a1 = OPTBASE\_PCTDB.t2.a2"). The input tables OPTBASE\_PCTDB.t1 and OPTBASE\_PCTDB.t2 will not be cached in memory, but OPTBASE\_PCTDB.t1 is eligible for synchronized scanning. The result goes into Spool 1 (group\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 9,999 rows (649,935 bytes). The estimated time for this step is 8.24 seconds.

...

### **DYNAMIC EXPLAIN When the Display of a Dynamic Plan is Not Allowed But the Request is Eligible for IPE**

This example demonstrates a DYNAMIC EXPLAIN report when the DBS Control field setting for displaying dynamic plans has been set not to allow dynamic plans to be reported. In this case, Vantage reports the static plan for the request.

```
DYNAMIC EXPLAIN
SELECT *
FROM t1, t2
WHERE t1.a1 < (SELECT MAX(t3.a3)
 FROM t3)
AND t1.a1 = t2.a2;
```

#### Explanation

-----

**This request is eligible for incremental planning and execution (IPE). The following is the static plan for the request.**

...

- 5) We do an all-AMPs SUM step to aggregate from OPTBASE\_PCTDB.t3 by way of an all-rows scan with no residual conditions. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (19 bytes). The estimated time for this step is 0.02 seconds.
  - 6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.00 seconds.
  - 7) We do an all-AMPs DISPATCHER RETRIEVE step from Spool 1 (Last Use) by way of an all-rows scan and send the rows back to the Dispatcher. The size is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.01 seconds.
  - 8) We do an all-AMPs JOIN step from OPTBASE\_PCTDB.t1 by way of a RowHash match scan with a condition of ("OPTBASE\_PCTDB.t1.a1 < :%SSQ20"), which is joined to OPTBASE\_PCTDB.t2 by way of a RowHash match scan with no residual conditions. OPTBASE\_PCTDB.t1 and OPTBASE\_PCTDB.t2 are joined using a sliding-window merge join, with a join condition of ("OPTBASE\_PCTDB.t1.a1 = OPTBASE\_PCTDB.t2.a2"). The input tables OPTBASE\_PCTDB.t1 and OPTBASE\_PCTDB.t2 will not be cached in memory, but OPTBASE\_PCTDB.t1 is eligible for synchronized scanning. The result goes into Spool 5 (group\_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with no confidence to be 33,333,334 rows (2,166,666,710 bytes). The estimated time for this step is 1 minute and 10 seconds.
- ...

### **DYNAMIC EXPLAIN for an IPE-Eligible Request Preceded by a USING Request Modifier**

The explained request for this example is not eligible for IPE because it is preceded by a USING request modifier. As a result, the DYNAMIC EXPLAIN request returns the static plan for the request.

```
DYNAMIC EXPLAIN
USING (x INTEGER)
SELECT *
FROM t1, t2
WHERE t1.a1 < (SELECT MAX(t3.a3)
 FROM t3)
AND t1.a1 = t2.a2
AND t1.a1 = :x;
```

## Explanation

-----  
 This request is eligible for incremental planning and execution (IPE) but dynamic plan does not support USING request modifier. The following is the static plan for the request.

...

- 3) We do an all-AMPs SUM step to aggregate from OPTBASE\_PCTDB.t3 by way of an all-rows scan with no residual conditions. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (19 bytes). The estimated time for this step is 0.05 seconds.
- 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.02 seconds.
- 5) We do an all-AMPs DISPATCHER RETRIEVE step from Spool 1 (Last Use) by way of an all-rows scan and send the rows back to the Dispatcher. The size is estimated with high confidence to be 1 row. The estimated time for this step is 0.02 seconds.
- 6) We do a single-AMP JOIN step from OPTBASE\_PCTDB.t2 by way of the unique primary index "OPTBASE\_PCTDB.t2.a2 = :x", which is joined to OPTBASE\_PCTDB.t1 by way of the unique primary index "OPTBASE\_PCTDB.t1.a1 = :x" with a residual condition of ( "(OPTBASE\_PCTDB.t1.a1 < :%SSQ21) AND (OPTBASE\_PCTDB.t1.a1 = :x)"). OPTBASE\_PCTDB.t2 and OPTBASE\_PCTDB.t1 are joined using a merge join, with a join condition of ("OPTBASE\_PCTDB.t1.a1 = OPTBASE\_PCTDB.t2.a2"). The result goes into Spool 5 (one-amp), which is built locally on that AMP. The size of Spool 5 is estimated with high confidence to be 1 row (87 bytes). The estimated time for this step is 0.02 seconds.

...

**DYNAMIC EXPLAIN When a Request is Not Eligible for IPE**

The explained request for this example is not eligible for IPE because it specifies no single-row access or scalar subquery operations. The static plan for the request is displayed.

There are no changes in the EXPLAIN text report from would be reported if IPE were disabled.

```
DYNAMIC EXPLAIN
SELECT *
FROM t1
WHERE a3 > 10;
```

## Explanation

```

...
3) We do an all-AMPs RETRIEVE step from OPTBASE_PCTDB.t1 by way of an
 all-rows scan with a condition of ("OPTBASE_PCTDB.t1.a3 > 10") 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 (54 bytes). The
 estimated time for this step is 0.03 seconds.
...

```

### **DYNAMIC EXPLAIN Where the Request Is Eligible for Incremental Planning and Execution But Does Not Meet Thresholds**

This example demonstrates an EXPLAIN report where the request being explained is eligible for IPE, but does not meet certain system-determined thresholds for execution costs.

Because system-determined thresholds for IPE are not met, Vantage displays the static plan for the request instead of the dynamic plan.

```

DYNAMIC EXPLAIN
SELECT *
FROM t1, t2
WHERE t1.a1 < (SELECT MAX(t3.a3)
 FROM t3)
AND t1.a1 = t2.a2;

```

## Explanation

**This request is eligible for incremental planning and execution (IPE) but does not meet thresholds. The following is the static plan for the request.**

```

...
5) We do an all-AMPs SUM step to aggregate from OPTBASE_PCTDB.t3 by
 way of an all-rows scan with no residual conditions. Aggregate
 Intermediate Results are computed globally, then placed in Spool 3.
 The size of Spool 3 is estimated with high confidence to be 1 row
 (19 bytes). The estimated time for this step is 0.02 seconds.
6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of
 an all-rows scan into Spool 1 (all_amps), which is built locally
 on the AMPs. The size of Spool 1 is estimated with high
 confidence to be 1 row (25 bytes). The estimated time for this
 step is 0.00 seconds.
7) We do an all-AMPs DISPATCHER RETRIEVE step from Spool 1 (Last Use)
 by way of an all-rows scan and send the rows back to the
 Dispatcher. The size is estimated with high confidence to be 1

```

row. The estimated time for this step is 0.02 seconds.

- 8) We do an all-AMPs JOIN step from OPTBASE\_PCTDB.t2 by way of a RowHash match scan, which is joined to OPTBASE\_PCTDB.t1 by way of a RowHash match scan with a condition of ("OPTBASE\_PCTDB.t1.a1 < :%SSQ20"). OPTBASE\_PCTDB.t2 and OPTBASE\_PCTDB.t1 are joined using a merge join, with a join condition of ("OPTBASE\_PCTDB.t1.a1 = OPTBASE\_PCTDB.t2.a2"). The result goes into Spool 5 (group\_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with no confidence to be 4 rows (260 bytes). The estimated time for this step is 0.02 seconds.

...

### **DYNAMIC EXPLAIN Where the Request is Eligible for IPE, But IPE is Disabled**

When you submit a DYNAMIC EXPLAIN for a request that is eligible for IPE, but IPE is disabled for your system, Vantage displays the static plan for the request. The first sentences of EXPLAIN text, highlighted in boldface type in the example, explain that the request is eligible for IPE, but IPE is disabled.

```
DYNAMIC EXPLAIN
SELECT *
FROM t1, t2
WHERE t1.a1 < (SELECT MAX(t3.a3)
 FROM t3)
AND t1.a1 = t2.a2;
```

Explanation

-----  
**This request is eligible for incremental planning and execution (IPE). IPE is disabled. The following is the static plan for the request.**

...

- 5) We do an all-AMPs SUM step to aggregate from OPTBASE\_PCTDB.t3 by way of an all-rows scan with no residual conditions. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (19 bytes). The estimated time for this step is 0.02 seconds.
- 6) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (all\_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (25 bytes). The estimated time for this step is 0.00 seconds.
- 7) We do an all-AMPs DISPATCHER RETRIEVE step from Spool 1 (Last Use) by way of an all-rows scan and send the rows back to the Dispatcher. The size is estimated with high confidence to be 1 row. The estimated time for this step is 0.02 seconds.



8) We do an all-AMPs JOIN step from OPTBASE\_PCTDB.t2 by way of a RowHash match scan, which is joined to OPTBASE\_PCTDB.t1 by way of a RowHash match scan with a condition of ("OPTBASE\_PCTDB.t1.a1 < :%SSQ20"). OPTBASE\_PCTDB.t2 and OPTBASE\_PCTDB.t1 are joined using a merge join, with a join condition of ("OPTBASE\_PCTDB.t1.a1 = OPTBASE\_PCTDB.t2.a2"). The result goes into Spool 5 (group\_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with no confidence to be 4 rows (260 bytes). The estimated time for this step is 0.02 seconds.

...

# Transaction Processing

This section describes how Vantage processes transactions.

## Database Transactions

Processing in Vantage is transaction-based. The principal purpose of transaction management is to optimize concurrency: to ensure that as many sessions as possible can access the information in the database concurrently without compromising the consistency or integrity of the data.

### Definition of a Database Transaction

A transaction is a sequence of  $n$  actions that are treated both as a unit of work and as a unit of recovery, though for the purposes of this discussion, it is viewed primarily as a unit of work.

- As a unit of work, a transaction defines a 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 called the atomic property of transactions.
- As a unit of recovery, a transaction defines a set of rollback operations to be performed on a database in order to change its state back to an earlier consistent state in the event that the transaction cannot successfully complete.

Beside its common definitions as a unit of work and a unit of recovery, a transaction can also be a unit of consistency, because it takes the database from one consistent state to another consistent state. In the course of a sequence of update operations, a database object may become transiently inconsistent as it undergoes a change to a new consistent state. Transactions are devised to provide consistency in the face of potential transient inconsistency. All transient inconsistencies may be isolated within the transaction and are never seen by other users. 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 Defined](#).

### Definitions of Statements and Requests in Teradata SQL

It is important to understand the definitions of a request and a statement in Teradata SQL.

*Request* is a Teradata-specific term used to describe a minimal unit of work that is transmitted from a client system to the database in a single message. Among the components of the message are the following.

- Request-level API options
- Zero or more SQL statements
- Request-related metadata
- Request-related data

ANSI SQL does not use the request concept and refers only to SQL statements, which do not define units of work.

Any SQL statement can be a single-statement request, but not every request is a single SQL statement.

The following topics describe the differences between statements and requests in the database in more detail.

- [Statement Processing](#)
- [Request Processing](#)

## Definitions of a Transaction in Teradata SQL

The definition of a transaction in Teradata SQL depends in part on the session mode in which it is submitted. In Teradata session mode, a transaction can either be implicit or explicit. In ANSI session mode, there are only ANSI transactions.

### Definition of an Implicit Transaction

Each request submitted in Teradata session mode is an implicit transaction unless it is preceded by a BEGIN TRANSACTION statement, in which case the transaction becomes explicit.

In Teradata session mode, because an implicit 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.

ANSI session mode does not have implicit transactions.

### Definition of an Explicit Transaction

You can also define transactions explicitly in Teradata session mode by using the BEGIN TRANSACTION statement.

Teradata session mode explicit transactions are completed by one of the following:

- An END TRANSACTION statement.
- An ABORT or ROLLBACK statement.
- A failure response for a request.
- The session is logged off, which aborts the transaction.
- The system restarts.

You cannot submit BEGIN TRANSACTION and END TRANSACTION statements in ANSI session mode. If you attempt to do so, the database aborts the request and returns an error or a failure response to the requestor.

When the Parser receives a BEGIN TRANSACTION statement, it immediately looks for an SQL statement keyword in the SQL text that follows. Keep this in mind when determining the placement of request modifiers such as EXPLAIN, LOCKING, NONTEMPORAL, and USING.

For example, if the first request in an explicit transaction is associated with a USING request modifier, that USING request modifier must precede, not follow, the BEGIN TRANSACTION request. In other words, the USING request modifier must be specified outside the BEGIN TRANSACTION boundary.

## Definition of ANSI Transactions

In ANSI session mode, a transaction begins when a request is submitted and there is not an uncompleted transaction. ANSI mode transactions are completed by one of the following:

- A COMMIT statement.
- An ABORT or ROLLBACK statement.
- A failure response for a request.
- The session is logged off, which aborts the transaction.
- The system restarts.

You cannot submit COMMIT requests in Teradata session mode. If you attempt to do so, the database aborts the request and returns an error to the requestor.

## Locking for Explicit and ANSI Transactions

When several requests are submitted as an explicit or ANSI transaction, the requests are processed one at a time. This means that the Optimizer has no way of determining what locks are needed by the transaction as a whole.

Because of this, locks are placed as each request is received. Locks are held until one of the following events completes, depending on the session mode, but regardless of when the user receives the data. (For example, a spool might exist beyond the end of the transaction.)

- The outermost END TRANSACTION statement in Teradata session mode.
- Submission of a COMMIT or ROLLBACK statement in ANSI session mode.

You must explicitly commit or rollback all transactions in ANSI session mode; otherwise they continue until you do commit them or roll them back.

- The system restarts.

## Errors and Failures

The following table briefly describes the conditions under 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.<br>The system does not return error codes in Teradata session mode; a failure code is returned instead. |
| Failure            | ANSI         | Rolls back the entire transaction that contains the error-generating request.                                           |
|                    | Teradata     |                                                                                                                         |

**Note:**

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.

Note that the system does not release any locks placed for a request that is rolled back because of an error-generating request. All such locks remain in effect until the transaction either commits or rolls back.

Note that errors do not complete ANSI session mode transactions, but failures do.

**ACID Properties of Transactions**

The general concept of transaction processing is encapsulated by their so-called ACID properties. ACID is an acronym for the following set of properties that characterize a transaction:

- Atomicity
- Consistency
- Isolation
- Durability

The specific meanings of these expressions in terms of 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.<br>Note that ANSI transaction semantics do not necessarily satisfy this property. |
| Consistency | A transaction transforms one consistent database state into another. Intermediate inconsistencies in the database are not permitted.                                                                                                                                                                                                                  |
| Isolation   | The operations of any transaction are concealed from all other transactions until that transaction commits. In practice, a transaction may not necessarily adhere to this property. See <a href="#">Levels of Isolation Defined</a> .                                                                                                                 |
| 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 persistence.                                                                                                                                                                                     |

It should be clear that not only are these four factors not rectangular, the degree of shared variance among them varies considerably. 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 for the reasons why this is true.

Note that transactions are not always atomic in ANSI session mode because when a request within a transaction fails with an Error response, only that request, and not the entire transaction, is rolled back. The remaining requests in the transaction continue until it either commits or rolls back.

### Using the Transient Journal to Roll Back Transactions

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. The before-change row images for a transaction are saved in the Transient Journal.

When a transaction fails for any reason, the system rolls back its updates using 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.

## Transactions, Requests, and Statements

The following topics describe how the database processes statements, requests, and transactions.

### Statement Processing

In Teradata SQL, statements have the following properties:

- They require locks on the database objects they access in order to ensure a level of isolation.

See [Database Locks, Two-Phase Locking, and Serializability](#).

- 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 single-statement request or a multistatement request.

Depending on the circumstances, a multistatement request can also be functionally equivalent to a database transaction.

- They can be components of a macro.
- They can be components of an SQL procedure.

### Request Processing

A request is composed of zero or more SQL statements and other information. A request that contains more than one SQL statement is called a multistatement request. A request with one SQL statement is called a single-statement request.

A request is sent to the database. It is processed and a response is returned. Then another request can be sent.

The following segments of system input analysis (see [Request Parsing](#) for details) are performed with one or more SQL statements for a request.

- SQL statement syntax checking (see [Syntaxer](#)).
- SQL statement syntax annotation (see [Resolver](#)).
- Privilege checking (see [Security Checking](#)).
- Request optimization (see [Optimizer](#) and [Query Rewrite, Statistics, and Optimization](#)).

The most restrictive locks required by the SQL statements in a request are acquired as early as possible. This is fundamental to the two-phase locking protocol (see [Database Locks, Two-Phase Locking, and Serializability](#)). Locks are never released within a request or transaction. They are released when the transaction completes.

For example, consider the following multistatement request:

```
SELECT
FROM employee
; UPDATE employee
 SET salary_amount = salary_amount * 1.1;
```

The SELECT statement in this multistatement request only requires a READ lock, but the UPDATE statement requires a WRITE lock. Because the WRITE lock is the most restrictive lock required by the multistatement request, the system applies it to the multistatement request for both the SELECT and the UPDATE statement components of the multistatement request.

## Processing Multistatement Requests

A Teradata request terminates with a SEMICOLON character at the end of a line. A semicolon placed at any other point in the request does not terminate it.

You can use this property to specify 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;
```

Multistatement requests have the following properties:

- They can only include DML statements.  
DCL and DDL statements cannot be components of a multistatement request.  
This property distinguishes multistatement requests from macros, which can contain a single DDL statement as long as it is the last statement in the request.
- They cannot include CALL statements.  
CALL is a disallowed DML statement in a multistatement request.

- The database executes the statements within a multistatement request in the order in which they are specified with knowledge of dependencies.
- The most exclusive locks required by any individual statement within the request are held for the entire length of the request and the transaction.
- With the exception of multistatement INSERT requests the outcome of a multistatement request, like the outcome of a transaction in Teradata session mode, is all or nothing.

If one statement in the request fails, the entire request fails and the database rolls back the transaction.

For multistatement INSERT requests that use statement independence, only those statements that fail are rolled back, and the successful statements in the multistatement request are committed.

Consult the appropriate Teradata Tools and Utilities documentation to determine whether a client API supports statement independence.

For more information on statement independence, see INSERT and INSERT ... SELECT in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

- A multistatement request can be committed either implicitly by a line-ending SEMICOLON character in Teradata session mode or explicitly with a COMMIT request in ANSI session mode or an END TRANSACTION request in Teradata session mode.

You can only submit an END TRANSACTION request to terminate an explicit transaction that was initiated with a BEGIN TRANSACTION request (for more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).

## Transaction Processing

The most commonly used example of a transaction is a debit-credit transaction at a bank ATM. Suppose you withdraw 10 dollars from your checking account and deposit it in your savings account. This is a two-part transaction: withdrawal of money from the checking account (debit phase) and depositing it in the savings account (credit phase). 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.

Suppose the debit phase of the transaction completes successfully, but the credit phase does not. Do those 10 dollars just disappear? Without proper transaction management, they just might. In the scenario presented here, the system 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 a subset of its work.

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.



# Transaction Semantics Differences in ANSI and Teradata Session Modes

## About Differing Transaction Semantics in Different Session Modes

You can perform transaction processing in any of the following session modes:

- ANSI
- Teradata
- 2PC

## About Default Session Modes

Teradata session mode ensures compatibility with legacy applications, although it introduces a few caveats.

- When you upgrade an existing 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.

## Changing Session Modes

To change the mode for a session using client software based on the CLIV2 API, do any of the following:

| Client Software | USE These Commands or Options                                                                                                                                                                                                                   | Session mode That You Switch To |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| BTEQ            | <code>.[SET] SESSION TRANSACTION ANSI</code>                                                                                                                                                                                                    | ANSI                            |
|                 | <code>.[SET] SESSION TRANSACTION BTET</code>                                                                                                                                                                                                    | Teradata                        |
|                 | Log off the current Teradata session before you enter this BTEQ command. You must submit this command before you log onto a Teradata session.<br>See <i>Basic Teradata® Query Reference</i> , B035-2414 for more detail on using BTEQ commands. |                                 |
| Preprocessor2   | <code>TRANSACT(ANSI)</code>                                                                                                                                                                                                                     | ANSI                            |
|                 | <code>TRANSACT(BTET)</code><br><code>TRANSACT(2PC)</code><br><code>TRANSACT(COMMIT)</code>                                                                                                                                                      | Teradata                        |
|                 | COMMIT is the default option. It follows Teradata transaction semantics with the exception that it permits the ANSI session mode-only COMMIT request to be used to terminate transactions.                                                      |                                 |

| Client Software | USE These Commands or Options                                                                                                                                                                                                                                                                                                                         | Session mode That You Switch To |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| CLIV2           | set tx_semantics = 'A'                                                                                                                                                                                                                                                                                                                                | ANSI                            |
|                 | set tx_semantics = 'T'                                                                                                                                                                                                                                                                                                                                | Teradata                        |
|                 | set tx_semantics = 'D'                                                                                                                                                                                                                                                                                                                                | Platform default                |
|                 | See the following manuals for more detail on setting the tx_semantics field. <ul style="list-style-type: none"> <li>• <i>Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems</i>, B035-2417</li> <li>• <i>Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems</i>, B035-2418</li> </ul> |                                 |

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 *Teradata Vantage™ - SQL Fundamentals*, B035-1141 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 commands.

- CLIV2 abort request
- TDP when the application terminates without proper session cleanup
- BTEQ with .ABORT

In both Teradata and ANSI session modes, implementation 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.

In ANSI session mode, request errors 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 request 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 ... |
|-------------------------------------------------------------------------------------|----------------------------------------|
| <ul style="list-style-type: none"> <li>• BEGIN TRANSACTION</li> <li>• BT</li> </ul> | ANSI                                   |

| These SQL statements ...                                                          | Are not valid in this session mode ...                                                                                                                                                                                                   |
|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• END TRANSACTION</li> <li>• ET</li> </ul> |                                                                                                                                                                                                                                          |
| COMMIT [WORK]                                                                     | Teradata<br>An exception is that you can submit COMMIT requests to terminate transactions from Preprocessor2 applications in Teradata session mode if you specified the Preprocessor2 TRANSACT (COMMIT) option to log onto the database. |

## Other Session Mode-Specific Features and Restrictions

The following additional session mode-specific features and restrictions apply to transaction processing:

- Session Pool Manager

The Session Pool Manager permits setting the Teradata or ANSI session mode for pooled sessions.

- Two-Phase Commit (2PC) Protocol

Vantage supports the 2PC protocol (see *Teradata Vantage™ - Database Introduction*, B035-1091 for a brief description of 2PC) for Teradata session mode, but not for ANSI session mode.

If you attempt to use the 2PC protocol while in ANSI session mode, the logon process aborts and a failure response is returned.

## Comparison of Transactions in ANSI and Teradata Session Modes

ANSI session mode is a state in which transaction processing semantics adhere to the rules defined by the ANSI SQL:2011 specification. Teradata (or BTET) session mode is a state in which transaction processing follows a set of rules defined by Teradata. Teradata session mode provides a means for conducting transaction processing by legacy applications.

In ANSI session mode, each transaction consists of one or more requests, each of which can consist of one or more SQL statements (see [Transactions, Requests, and Statements](#)) in ANSI session mode.

Multistatement requests are treated as a single atomic unit; either all the work done by all the statements in a multistatement request is committed or none of it is.

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.

| Rule                   | ANSI Session Mode                                                                                                                       | Teradata Session Mode                                                                                                                                                                                      |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Transaction initiation | A transaction initiates when no transaction is active and an SQL request is performed. A transaction is opened by one of the following. | A transaction initiates when no transaction is currently active and one of the following occurs: <ul style="list-style-type: none"> <li>• An SQL request is executed (an implicit transaction).</li> </ul> |

| Rule                    | ANSI Session Mode                                                                                                                                                                                                                                                                                                                                                                                        | Teradata Session Mode                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                         | <ul style="list-style-type: none"> <li>The first SQL request executed in a session.</li> <li>The next request performed in the session following the close of a transaction in ANSI session mode.</li> </ul> <p>The BEGIN TRANSACTION (BT) statement is not valid.</p>                                                                                                                                   | <ul style="list-style-type: none"> <li>There is a BEGIN TRANSACTION request (an explicit transaction). Unless initiated by BEGIN TRANSACTION (BT), the system treats each request as an implicit transaction (see <a href="#">Transactions, Requests, and Statements</a>).</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Transaction termination | <p>The transaction terminates with either a COMMIT [WORK] or ROLLBACK [WORK] (or ABORT) statement, or a failure, times out, logoff, or system restart.</p> <p>The END TRANSACTION statement is not valid.</p>                                                                                                                                                                                            | <ul style="list-style-type: none"> <li>An implicit transaction terminates at the end of the request, ROLLBACK or ABORT statement, a failure, times out, or a system restart.<br/>An implicit transaction terminates when it either completes successfully (Success response) or causes a Failure response. (See <i>Teradata Vantage™ - SQL Fundamentals</i>, B035-1141 for information about success, warning, error, and failure responses.)</li> <li>An explicit transaction terminates with an END TRANSACTION statement, ROLLBACK or ABORT statement, failure, times out, logoff, or system restart.<br/>In the case of nested transactions, it is not the first END TRANSACTION request encountered that terminates the transaction, but the last.<br/>The COMMIT request is not valid.<br/>You can mix both explicit and implicit transactions within the same script.<br/>When a transaction commits, Vantage discards its Transient Journal (see <a href="#">Using the Transient Journal to Roll Back Transactions</a>) and closes any open cursors.<br/>If a transaction is not successfully committed, either explicitly or by an END TRANSACTION statement, the database rolls back all your requests for the transaction.</li> </ul> |
| Cursors                 | Defaults to being positioned.                                                                                                                                                                                                                                                                                                                                                                            | Defaults to being not positioned.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Error behavior          | <p>Errors roll back only the request that causes them, not the entire transaction.</p> <p>See <i>Teradata Vantage™ - SQL Fundamentals</i>, B035-1141 for information about success, warning, error, and failure responses.</p> <p>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,</p> | <p>Not applicable.</p> <p>Something that would be an error in ANSI mode is treated as a failure in Teradata session mode.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

| Rule                                        | ANSI Session Mode                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | Teradata Session Mode                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|---------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                             | <p>they do not support the A property of ACID transactions (see <a href="#">ACID Properties of Transactions</a>) in all circumstances.</p> <p>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.</p> <p>Control of character truncation of trailing non-blank characters causes errors in ANSI session mode.</p> <p>Use the SUBSTRING function to prevent such errors. For more information, see <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i>, B035-1145.</p> <p>Locks placed by erroneous requests are not released and the Transient Journal is not deleted from the dictionary.</p> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Failure behavior                            | Failures roll back the entire transaction. The Transient Journal is first applied, then deleted from the dictionary, and then all locks are released.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | 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 request to force the rollback to occur in Teradata session mode. Failure responses roll back the entire transaction, not just the request that evokes them. Error responses do not occur, that is, any response that would be treated as an error in ANSI session mode is treated as a failure in Teradata session mode. |
| Default attribute for character comparisons | CASESPECIFIC                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | NOT CASESPECIFIC                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Default TRIM behavior                       | TRIM(BOTH FROM)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | TRIM(BOTH FROM)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Control of character truncation errors      | Error response for truncation problems with trailing non-pad characters.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | Truncation problems never cause failures; the truncation is silently applied.                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Default table type semantics                | MULTISET<br>This means that duplicate rows are allowed when updating or inserting rows into tables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | SET<br>This means that duplicate rows are not allowed when updating or inserting rows into tables.                                                                                                                                                                                                                                                                                                                                                                                                             |
| DDL statement placement                     | Must be the only statement in the last request in the transaction, other than an optional COMMIT statement.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | Must be the only statement in the last request in the transaction, other                                                                                                                                                                                                                                                                                                                                                                                                                                       |

| Rule                                                | ANSI Session Mode                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Teradata Session Mode                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-----------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                     | <p>If you attempt to perform another DDL request before you issue a COMMIT request, the database returns an Error for the non-valid request, but does not roll back the transaction.</p> <p>Although they are technically DCL statements, the database treats the DATABASE and SET SESSION statements as DDL statements for the purposes of handling transactions.</p>                                                                                                                                                                                                                                                                | <p>than the END TRANSACTION for an explicit transaction.</p> <p>If you attempt to perform another DDL request before you commit the transaction, the system returns a failure response for the non-valid request, and then rolls back the transaction.</p> <p>Although they are technically DCL statements, the database treats the DATABASE and SET SESSION statements as DDL statements for the purposes of handling transactions.</p>                                                             |
| Statements performed through a logon startup string | Not applicable.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | <p>Must follow standard rules for the following:</p> <ul style="list-style-type: none"> <li>• Transaction definition</li> <li>• Transaction termination</li> <li>• DDL request placement</li> </ul>                                                                                                                                                                                                                                                                                                  |
| Logoff behavior                                     | If you log off prior to committing your work, then the system rolls back all your transaction requests.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Two-phase commit (2PC)                              | <p>Not supported.</p> <p><i>Two-Phase Commit</i> is 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.</p> <p>For more information, see <i>Teradata Vantage™ - Database Introduction</i>, B035-1091.</p>                                                                                                                                                                                                                                                                                                                            | <p>Supported.</p> <p><i>Two-Phase Commit</i> is 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.</p> <p>For more information, see <i>Teradata Vantage™ - Database Introduction</i>, B035-1091.</p>                                                                                                                                                                                               |
| Locks                                               | <p>The most exclusive locks (READ, WRITE, EXCLUSIVE) are retained at the highest level (rowkey, rowhash, partition, table, and so on) and are not released until a transaction is committed or rolled back.</p> <p>The database checks the number of control blocks used for Rowhash locks in a transaction and aborts it if the number exceeds the threshold set for the DBS Control field MaxRowHashBlocksPercent.</p> <p>For more information, see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</p> <p>Any locks that are placed for a request that is rolled back are not released because of an error response.</p> | <p>The most exclusive locks (READ, WRITE, EXCLUSIVE) are retained at the highest level (rowkey, rowhash, partition, table, and so on) and are not released until a transaction is committed.</p> <p>The database checks the number of control blocks used for rowhash locks in a transaction and aborts it if the number exceeds the threshold set for the DBS Control field MaxRowHashBlocksPercent.</p> <p>For more information, see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</p> |

## ANSI Session Mode Transaction Processing Case Studies

This topic presents several case studies concerning ANSI session mode transaction semantics. Also see [Teradata Session Mode Transaction Processing Case Studies](#)).

### Successful ANSI Mode Transaction Example

The following transaction is an example of a successful ANSI mode transaction:

|                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                               |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>INSERT INTO employee SELECT * FROM employee; *** Insert completed. 26 rows added.</pre>                                                                                                                           | <p>This single request begins an ANSI mode explicit transaction.</p> <p>WRITE locks are held on employee.</p>                                                                                                                                                                                                 |
| <pre>UPDATE employee SET department_number = 400 WHERE department_number = 401 ;DELETE FROM employee WHERE department_number = 401; *** Update completed. 7 rows changes. *** Delete completed. No rows removed.</pre> | <p>This request is composed of two statements.</p> <p>WRITE locks are still held on employee.</p>                                                                                                                                                                                                             |
| <pre>SELECT * FROM employee WHERE department_number = 401; *** Query completed. No rows found.</pre>                                                                                                                   | <p>This single statement request is a check to ensure that all employees in department_number 401 were deleted in the previous request.</p> <p>Notice that even though this is a SELECT request, which only requires a READ (or ACCESS) lock, the transaction continues to hold a WRITE lock on employee.</p> |
| <pre>COMMIT; *** COMMIT done. *** Total elapsed time was 1 second.</pre>                                                                                                                                               | <p>This request terminates the transaction by committing all changes.</p> <p>All locks are released and the Transient Journal is dropped from the dictionary.</p> <p>The next request entered begins another ANSI transaction.</p>                                                                            |

### ANSI Session Mode Error and Failure Responses

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:

|                                                                                                               |                                                                        |
|---------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| <pre>INSERT INTO employee SELECT * FROM customer_service.employee; *** Insert completed. 26 rows added.</pre> | <p>This INSERT ... SELECT request begins an ANSI mode transaction.</p> |
|---------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|

|                                                                                                                |                                                                                                                                                                                                                                               |
|----------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>SELECT * FRM employee; *** Error 3706 Syntax error; SELECT * must have a FROM clause.</pre>               | <p>Syntexer problems are errors, not failures in an open transaction.</p> <p>The inserts from the previous request remain, and all locks continue to be in force.</p>                                                                         |
| <pre>SELECT * FROM employee; *** Query completed. 26 rows found. 9 columns returned.</pre>                     | <p>Correcting the SELECT syntax produces a successful request that indicates the inserted rows from the first request remain in place.</p> <p>The transaction is still not committed.</p>                                                     |
| <pre>SELECT * FROM employee WHERE emp_num = 1010; *** Error 5628 Column emp_num not found in Employee.</pre>   | <p>Resolver problems are neither always errors, nor always failures. In this case, the problem evokes an Error response, not a Failure response.</p> <p>The inserts from the first request remain, and all locks continue to be in force.</p> |
| <pre>SELECT * FROM employee; *** Query completed. 26 rows found. 9 columns returned.</pre>                     | <p>Repeating the selection of all columns from the table proves that the inserted rows from the first request remain.</p> <p>The transaction is still not committed.</p>                                                                      |
| <pre>COMMIT;</pre>                                                                                             | <p>The transaction and the rows inserted in the first request are committed.</p>                                                                                                                                                              |
| <pre>SELECT * FRM employee; *** Failure 3706 Syntax error; SELECT * must have a FROM clause.</pre>             | <p>A failure response is returned since there is no open transaction.</p> <p>Note that above, an error response is returned, since a transaction was successfully opened.</p>                                                                 |
| <pre>INSERT INTO employee SELECT * FROM customer_service.employee; *** Insert completed. 26 rows added.</pre>  | <p>This INSERT ... SELECT request begins an ANSI mode transaction.</p>                                                                                                                                                                        |
| <pre>CREATE TABLE tbl_1 (   col_1,   col_2 INTEGER); *** Error 3739 The user must give a data type for .</pre> | <p>Syntexer problems are errors, not failures in an open transaction.</p> <p>The inserts from the previous request remain, and all locks continue to be in force.</p>                                                                         |
| <pre>SELECT * FROM employee; *** Query completed. 26 rows found. 9 columns returned.</pre>                     | <p>Selecting all columns from the table proves that the inserted rows from the first request remain.</p> <p>The transaction is still not committed.</p>                                                                                       |
| <pre>CREATE TABLE tbl_1 (   col_1 INTEGER,</pre>                                                               | <p>This resolver problem evokes a Failure response, which terminates the transaction.</p>                                                                                                                                                     |



|                                                                           |                                                                                                                                                    |
|---------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>col_2 INTEGER); *** Failure 3802 Table 'tbl_1' already exists.</pre> | The transaction rolls back and the rows inserted into employee after the commit above are deleted.                                                 |
| <pre>SELECT * FROM employee; *** Query completed. No rows found.</pre>    | This request begins an ANSI mode transaction. Selecting all columns from the table proves that the previously inserted rows have all been deleted. |

## Placing DDL Requests Within ANSI Mode Transactions

Like Teradata mode transactions, there can only be 1 DDL request in a transaction, and it must be the last sequential request other than a COMMIT 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 request or a ROLLBACK/ABORT request.

|                                                                                                                                                              |                                                                                                                                                                                                                                 |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>CREATE TABLE table_3 ( col_1 INTEGER); *** Table has been created.</pre>                                                                                | This single-statement DDL request begins an ANSI mode transaction. No further requests are valid within the boundaries of the current transaction.                                                                              |
| <pre>SHOW TABLE table_3; *** Error 3722 Only a COMMIT WORK or null statement is legal after a DDL Statement.</pre>                                           | This single-statement 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. The transaction does not roll back. |
| <pre>COMMIT; *** COMMIT done.</pre>                                                                                                                          | A COMMIT request commits the transaction, which is now complete.                                                                                                                                                                |
| <pre>SHOW TABLE table_3; CREATE MULTISET TABLE DB.table_3 , NO FALLBACK, NO BEFORE JOURNAL, NO AFTER JOURNAL ( col_1 INTEGER) PRIMARY INDEX ( col_1 );</pre> | A new ANSI 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 multistatement request and macro, 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 request, it can either create a Transient Journal entry for each row deleted from a table or create only one Transient Journal entry for the entire transaction.

The following DELETE is a single request followed by another request to commit the 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 multistatement request contains the same two statements as the transaction above, but because they are packaged as a multistatement request, both statements are seen by the system together. The system does not write a Transient Journal entry for each row deleted from the table, but instead writes a single Transient Journal entry. The multistatement request's performance is very good.

```

DELETE FROM table_1
;COMMIT;

```

In the first case, the system knows what the next request is and that it needs to be prepared to roll back the transaction. In the second case, the system can see that the delete is to be committed, and therefore the system knows that it does not need to roll back the transaction.

## Teradata Session Mode Transaction Processing Case Studies

This topic presents several case studies concerning Teradata session mode transaction semantics. Also see [ANSI Session Mode Transaction Processing Case Studies](#)).

### Failed Teradata Session Mode Transaction Example

|                                                                                                                                                                           |                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>BTEQ -- Enter your DBC/SQL request or BTEQ command: BEGIN TRANSACTION; *** Begin transaction accepted.</pre>                                                         | Beginning of an explicit transaction.                                                                                                                          |
| <pre>BTEQ -- Enter your DBC/SQL request or BTEQ command: INSERT INTO employee SELECT * FROM customer_service.employee; *** Insert completed. 26 rows added.</pre>         | Single request.<br>WRITE locks held.                                                                                                                           |
| <pre>BTEQ -- Enter your DBC/SQL request or BTEQ command: SELECT * FRM employee WHERE empnum = 401; *** Failure 3706 Syntax error; SELECT * must have a FROM clause.</pre> | Invalid syntax is a failure.<br>Transaction rolled back.<br>All previous requests in the transaction<br>are also rolled back.<br>All locks released.           |
| <pre>BTEQ -- Enter your DBC/SQL request or BTEQ command: SELECT * FROM employee; *** Query completed. No rows found.</pre>                                                | Single request.<br>Implicit transaction.                                                                                                                       |
| <pre>BTEQ -- Enter your DBC/SQL request or BTEQ command: END TRANSACTION; *** Failure 3510 Too many END TRANSACTION statements.</pre>                                     | Request to end the transaction causes<br>a failure response because the<br>transaction begun with the BEGIN<br>TRANSACTION request had already<br>rolled back. |

### Teradata Session Mode Requests

|                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>DELETE FROM table_1 WHERE PI_col=2; INSERT INTO table_1 VALUES (2,3,4);</pre> | <p>These 3 statements are separate requests. They are also implicit transactions. The implications of this are as follows:</p> <ul style="list-style-type: none"> <li>• The requests are performed serially in the order they are received.</li> <li>• Their locks are applied and released separately.</li> <li>• The success or failure of each has no effect on the success or failure of the others.</li> </ul> |
|------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                                                                                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| UPDATE table_1<br>SET col_3=4;                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| BEGIN TRANSACTION;<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>END TRANSACTION; | <p>These 5 statements are separate requests within a single explicit transaction. The implications of this are as follows.</p> <ul style="list-style-type: none"> <li>• The requests are performed serially in the order they are specified in the transaction.</li> <li>• 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 statement, ABORT statement, failure, logoff, or system restart rolls back the work.</li> <li>• The success or failure of each has a direct effect on the success or failure of the others.</li> </ul> |
| 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;                                           | <p>These 3 statements form a single multistatement request. They are also implicitly a single transaction. The implications of this are as follows:</p> <ul style="list-style-type: none"> <li>• The most restrictive lock held by the transaction, a table-level WRITE lock, is applied to <i>table_1</i>.</li> <li>• The work done by the transaction is atomic: either it is all committed or it is all rolled back.</li> </ul>                                                                                                                                                                                                                               |
| 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>);          | <p>This macro contains 3 separate requests. Because they are contained within the same macro, they behave identically to a multistatement request that contains the same three requests in the same order.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| EXEC mac_1;                                                                                                                                                  | <p>The result of executing the macro is atomic in exactly the same way its equivalent multistatement request above is atomic.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| EXPLAIN EXEC mac_1<br>EXPLAIN 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;             | <p>The EXPLAIN reports generated for these 2 requests are identical.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

### Mixing DDL and DML Statements Within a Multistatement Request

You cannot mix DDL and DML statements within the same macro or multistatement request in Teradata session mode. An attempt to perform such a request results in a failure response.

For example, the following multistatement request fails because it mixes DML (2 SELECT requests) with DDL (a CREATE TABLE request):

```

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 results in the same failure at the time that you attempt to create the macro.

```

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 action statement 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 Session Mode DELETE Performance for Different Transaction Structures

Depending on how you structure a transaction that contains a DELETE request, it can either create a Transient Journal entry for each row deleted from a table or create only one Transient Journal entry for the entire transaction.

The following DELETE request is a single implicit transaction. It does not write a Transient Journal entry for each deleted row, so its performance is quite good.

```
DELETE FROM table_1;
```

The following explicit transaction contains only BEGIN TRANSACTION and END TRANSACTION requests in addition to the DELETE request. Because of the way it is structured, the transaction 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 requests as the previous transaction, but because they are packaged as a multistatement request, they are treated as an implicit transaction. Vantage 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;
```

In the first case, the system knows what the next request is and that it needs to be prepared to roll back the transaction. In the second case, the system can see that the delete is to be committed, and therefore the system knows that it does not need to roll back the transaction.

## Rollback Processing

This section describes rollback (abort) processing for Teradata and ANSI session modes.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details about the ABORT and ROLLBACK statements.

### Rollback Processing in ANSI Session Mode

ANSI only recognizes termination of a transaction by the performance of a COMMIT [WORK] or ABORT/ ROLLBACK [WORK] request 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, which means that a request in which this happens is not atomic (for more information, see [ACID Properties of Transactions](#)). The Lock Manager does not release locks placed for a rolled back request.

The entire transaction is rolled back when the current request is in one of the following states:

- ABORT or ROLLBACK
- Deadlocked.
- An 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.

## Rollback Processing in Teradata Session Mode

For a transaction, either all submitted requests are performed, or none are. Another way of stating this is to say that all transactions in Teradata session mode are atomic (for more information, see [ACID Properties of Transactions](#)).

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 requests in the transaction.
3. Erases any partially accumulated results (spools).

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.
- The transaction may fail due to the accumulation of too many locks.
- If a failure, time out, ABORT/ROLLBACK, logoff, or system restart occurs, rollback of the work already performed by the transaction may delay releasing locks and the availability of the locked objects.
- If the system must restart during the transaction, rollback of the work already performed by the transaction might delay post-restart system availability.

## Application-Initiated Asynchronous Aborts

An application-initiated asynchronous abort causes full transaction rollback in both ANSI and Teradata session modes. The term *application* in this case refers to a component of the database management system, whether client-based or server-based, and not to user-written applications.

This can occur in several ways, for example, by means of a CLIV2 abort request, or by the TDP when the application terminates without proper session cleanup, or by using a Teradata tool such as Teradata Studio™.

# Database Locks, Two-Phase Locking, and Serializability

## Database Locks Defined

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 database lock is the identity of the database object it is locking, the identity of the transaction holding it, and its level and severity.

You can think of the level and severity of a database lock as guarantees made to the transaction, assuring it that the objects for which it has requested locks are isolated to the desired level from changes made by other concurrently running transactions on those objects.

Database locks can be placed on the individual row partitions of a row-partitioned table, as well as on either the entire table, individual rowhash values within individual row partitions, or on individual row hash values.

## Two-Phase Locking Defined

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.

The phases are as follows:

- 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.

In practice, the shrinking phase occurs at the point that the transaction commits or finishes rolling back; at this point, all the database locks are released.

The principal concurrency problems that 2PL prevents are elaborated by the classic problems of transaction processing, usually named as follows:

- The lost update problem.
- The uncommitted dependency problem.
- The inconsistent analysis problem.
- The dirty read problem.
- The deadlock problem (see [Deadlock](#) and [Proxy Locks](#)).

The following additional problems can also occur:

- Increased system overhead to administer locking.
- Decreased concurrency.

The impact of the lower concurrency that locks introduce is reduced greatly in a system like Vantage that supports multiple levels of locking granularity.

Consult any standard transaction processing textbook for more information about these concurrency problems.



## Serializability Defined

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).

Serializability violations can occur for some DML operations for temporal tables. See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for details and how you can work around it.

Vantage ensures serializability for nontemporal transactions as long as the current isolation level for the session is SERIALIZABLE (see [ACID Properties of Transactions](#) and the SET SESSION CHARACTERISTICS statement in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for details).

For example, suppose *table\_A* is a checking accounts table and table B is a savings accounts table. Suppose one transaction, *Tx<sub>1</sub>*, 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<sub>2</sub>*, 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<sub>1</sub>* 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<sub>1</sub>* has deposited the \$400.00 withdrawn from the checking account into the savings account.

Without two-phase locking, *Tx<sub>2</sub>* 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.

The condition that 2PL ensures is serializability. When serializability is in force for nontemporal transactions, 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 is sufficient, but not necessary, to ensure serializability of nontemporal transactions.

The term *serializable* was originally a synonym for the ACID property known as Isolation (see [ACID Properties of Transactions](#) and the SET SESSION CHARACTERISTICS statement in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144), but the usage of the term isolation in the ANSI SQL standard has been broadened to include the concept of isolation levels, so it is no longer synonymous with serializable.

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 Defined

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. The ANSI SQL standard formalizes what it refers to as four isolation levels for transactions. To be precise, this section of the standard defines SERIALIZABLE 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.

The defined phenomena are dirty read, non-repeatable read, and phantom read.

The non-serializable isolation levels ANSI defines are as follows:

- Read Uncommitted
- Read Committed
- Repeatable Read

The following table shows the relationship between the different isolation levels and the different kinds of prohibited read phenomena.

| Isolation Level  | Dirty Read   | Non-Repeatable 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 |

Vantage does not support the isolation levels READ COMMITTED and REPEATABLE READ.

The READ UNCOMMITTED isolation level is implemented using an ACCESS level lock (see the ACCESS entry in the Lock Severity table in [Database Locking Levels and Severities](#) and the information about SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

Vantage does support a form of isolation called *load* isolation, which makes use of row versioning. Teradata permits concurrent reads on committed rows even though the underlying table is being modified. The load isolation phenomenon is compatible with the Read Committed isolation level, but it is not limited to a regular transaction boundary. The phenomenon persists until the load is committed and can occur only on load-isolated tables. A non-repeatable read or a phantom read can occur since a query that uses the LOAD COMMITTED locking modifier only acquires an ACCESS lock and thus allows a concurrent load-isolated write to occur. If a load commit happens in a concurrent session, then the reader transaction can see modified rows if the same query is issued again even within the same transaction. This is a non-repeatable read phenomenon. Additionally, a phantom read phenomenon can occur, wherein a repeated execution of the same query with search conditions after a concurrent load commit causes the different rows to be returned due to newly committed data.

For information about load isolation, see [Load Isolation](#).

## Changing the Transaction Isolation Level for Read-Only Operations

Sometimes you might be willing to give up a level of transaction isolation 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.

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 use 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 application phase (see *Teradata® MultiLoad Reference*, B035-2409 for details), using ACCESS locks to query the target table of the MultiLoad job during an application phase can produce extremely inaccurate result sets. In this case, the results probably would not provide even a reasonable impression of the table data.

Vantage provides methods for allowing the possibility 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 request modifier.<br>See <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 for details of the syntax and usage of this request modifier.                                                        |
| session                                                          | SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement.<br>See <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144 for details of the syntax and usage of this statement. |

Note that the global application of ACCESS locking for read operations when SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is set to READ UNCOMMITTED depends on the setting of the DBS Control field AccessLockForUncomRead.

When the field is set FALSE, SELECT operations within INSERT, DELETE, MERGE, and UPDATE requests set READ locks, while when the field is set TRUE, the same SELECT operations set ACCESS locks. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102 and the information about SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Retrievals With an ACCESS Lock

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, a situation that is referred to as a *dirty read*. 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 that a SELECT request 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 by another user, the column 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 might occur even if the data is changed only momentarily by a transaction that is later backed out.

The normal policy for ACCESS lock Read operations is that Vantage guarantees to return all rows that were not being updated at the time of the ACCESS lock Read operation to the requestor. For rows that are being updated, rows may be returned, possibly in an inconsistent state, or not returned.

## Considerations for Specifying LOCKING FOR ACCESS

A READ lock is normally placed on an object for a SELECT operation, which causes the request to be queued if the object is already locked for WRITE.

If an ad hoc query has no concern for data consistency, the LOCKING request modifier can be used to override the default READ lock with an ACCESS lock. For example:

```
LOCKING TABLE tablename FOR ACCESS
SELECT ...
FROM tablename ...;
```

Be aware that the effect of LOCKING FOR ACCESS is that of reading while writing, so dirty reads can occur with this lock. The best approach to specifying ACCESS locks is to use them only when you are interested in a broad, statistical snapshot of the data in question, not when you require precise results. On load-isolated tables, however, LOCKING FOR LOAD COMMITTED may be used to obtain committed data even while concurrent isolated writes occur simultaneously on the table.

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 request 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.

### Using the LOCKING Request Modifier: An 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 qualify\_accnt is defined as a secondary index, the following request could return the result that follows the request text:

```
LOCKING TABLE acct_rec FOR ACCESS
SELECT acct_no, qualify_accnt
FROM acct_rec
WHERE qualify_accnt = 1587;
Acct_No Qualify_Accnt
```

|       |       |
|-------|-------|
| ----- | ----- |
| 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. Note that for load isolated tables, you can avoid such inconsistency by using the `LOCKING FOR LOAD COMMITTED` modifier. Refer to [The LOCKING FOR LOAD COMMITTED Request Modifier](#) and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## Load Isolation

Teradata provides a table isolation property that enables you to read committed rows from tables while the tables are being loaded with data. Even if the data is changing while the table is being loaded, load isolation ensures that the most recent committed data is retrieved.

You can use the `WITH CONCURRENT ISOLATED LOADING` option of a `CREATE/ALTER TABLE` request to define tables as load isolated. Load-isolated tables record the most recent committed load ID values in the data dictionary. Read operations on load-isolated tables use the most recent committed load ID to isolate the rows that are committed from those that are not yet committed. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

To enable committed reads in concurrent sessions, rows that are being modified are logically deleted from load-isolated tables and new rows with the modified values are inserted when rows are updated. If a row is not yet committed, then the data that is read is the data that was true and final in the table prior to the subsequent data change. To enable concurrent index-based reads, indexes (including join indexes) also maintain the commit property of the row. A join index table is marked as a load-isolated table if any of the referenced base tables is a load-isolated table.

You can use the `LOAD COMMITTED` locking modifier to read committed rows in load-isolated tables without blocking load-isolated modifications.

For more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Modifications on load-isolated tables are classified as either concurrent or nonconcurrent. Concurrent load-isolated modifications (CLDIs) are modifications that permit concurrent reader sessions to select load-committed rows. Nonconcurrent load-isolated modifications (NCLDIs) are modifications that block the reader sessions from selecting committed rows. NCLDI modifications performed using SQL use `EXCLUSIVE` locks instead of `WRITE` locks. NCLDI modifications that are performed using a utility such as FastLoad or MultiLoad, or a load operator using TPT or the TPT update operator that does not use `MLOADX` may use an `EXCLUSIVE` lock or a `WRITE` lock. An acquired `WRITE` lock during the utility-based load operation may be upgraded to an `EXCLUSIVE` lock for load-isolated target tables, which can result in blocking concurrent read sessions even if they are being performed using an `ACCESS` lock.

Load operations are started implicitly, in a transaction. A load operation is committed at the time the transaction is committed. The session performing this transaction is the load session.

### **Load Processing for Load-isolated Tables**

For load-isolated tables, a load operation is limited to a single transaction.

All rows modified during a load operation are considered load-uncommitted rows and the table is considered to be in load state until the load operation is committed.

The first committed load-isolated DML SQL that you issue on a table begins the load on the load-isolated table. The table is marked for load, associated with both the session and the transaction. When the transaction commits, the load operation is committed. If transaction is rolled back, the load operation is rolled back.

If an underlying join index on the load-isolated table is modified, then the join index is also marked as being loaded. As part of the transaction commit, such tables are load committed with the following actions:

- DBC.TVM rows for the tables in load are updated with the corresponding CurrentLoadID property value, incremented by 1.
- During the commit step processing of the transaction commit, the following events take place:
  - The table header row for the tables is updated to mark load completion.
  - Load-isolated spoil steps are generated for the TVM and the table header of the load-isolated table to update the dictionary cache with the updated CurrentLoadID value.

### **Load State Parameters for an Implicit Transaction-based Load-isolated Load**

The following load state information is recorded in the table header of the load-isolated table when the load is an implicit transaction-based load:

- Logical HostID and Session number of the session performing the load.
- NewLoadID value used in the load operation.

### **The LOCKING FOR LOAD COMMITTED Request Modifier**

The LOCKING request modifier permits a user to override the default lock that the system places on a database, table, view, or a row-hash. When users want to read load-isolated tables, they can use LOAD COMMITTED locking.

---

#### **Note:**

The LOAD COMMITTED lock mode is not a new type of lock. The lock displayed or reported by the Lock Manager for this clause is the existing ACCESS lock.

---

A LOAD COMMITTED lock mode causes the system to place an ACCESS lock if you specify it by using a NULL SQL request or if the referenced object is not used in the SQL request. If you specify a LOAD COMMITTED lock mode on a nonload-isolated table, this action also results in the system applying an ACCESS lock.



LOAD COMMITTED lock mode behavior is similar to the ACCESS lock locking behavior. The primary difference is in the data that it selects from a load-isolated table when the table is referenced in an SQL request.

- If the table is a load-isolated table and it is not being loaded by the session, then load-committed data is returned from the table. This indicates that the system has applied a load-committed condition to the table being read.
- If the table is a load-isolated table and it is being loaded by the session, then the uncommitted data is also returned from the table. This indicates that the system has applied a load-uncommitted condition to the table being read.

If a lock cannot be upgraded, the locking modifier is ignored. A LOCKING FOR ACCESS modifier on a view that defines a LOAD COMMITTED lock mode is ignored.

The following table describes the behavior with and without an explicit LOCKING modifier being specified for the transaction isolation level of a table. If the session isolation level is SERIALIZABLE, then read operations on load-isolated tables block concurrent write operations. The recommended method for load-isolated tables is to specify the LOAD COMMITTED locking modifier so that the session does not block concurrent writes but still selects load-committed rows from the table.

| Isolation Level  | Locking Modifier               | DBS Control Field 54 setting (AccessLockForUncomRead) | Read Table Source in Modification statement | Read Table Non-Modification statement |
|------------------|--------------------------------|-------------------------------------------------------|---------------------------------------------|---------------------------------------|
| SERIALIZABLE     | Not specified                  | Not applicable                                        | READ load committed condition               | READ load committed condition         |
| SERIALIZABLE     | Locking src for access         | Not applicable                                        | ACCESS load uncommitted condition           | ACCESS load uncommitted condition     |
| SERIALIZABLE     | Locking src for load committed | Not applicable                                        | ACCESS load committed condition             | ACCESS load committed condition       |
| READ UNCOMMITTED | Not specified                  | FALSE (Default)                                       | READ load committed condition               | ACCESS load uncommitted condition     |
| READ UNCOMMITTED | Locking src for load committed | Not applicable                                        | ACCESS load uncommitted condition           | ACCESS load uncommitted condition     |
| READ UNCOMMITTED | Not specified                  | TRUE                                                  | ACCESS load uncommitted condition           | ACCESS load uncommitted condition     |



| Isolation Level  | Locking Modifier               | DBS Control Field 54 setting (AccessLockForUncomRead) | Read Table Source in Modification statement | Read Table Non-Modification statement |
|------------------|--------------------------------|-------------------------------------------------------|---------------------------------------------|---------------------------------------|
| READ UNCOMMITTED | Locking src for load committed | Not applicable                                        | ACCESS load committed condition             | ACCESS load committed condition       |
| READ UNCOMMITTED | Locking src for read           | Not applicable                                        | READ load committed condition               | READ load committed condition         |

For information about the LOCKING LOAD COMMITTED request modifier, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## Modifying Load-isolated Tables

You can use INSERT, UPDATE, MERGE, and DELETE statements on load-isolated tables with the WITH CONCURRENT ISOLATED LOADING or WITH NO CONCURRENT ISOLATED LOADING syntax. If you specify WITH NO CONCURRENT ISOLATED LOADING, then the modification is performed as a nonconcurrent, load-isolated modification (NCLDI).

A CLDI modification is a modification on a load-isolated table that is isolated from a concurrent load-committed-based read operation. The modification is performed while isolating the change by saving the version of the row being modified or deleted.

A NCLDI modification is a modification on a load-isolated table that does not permit a concurrent read operation. It follows the same modification method that is used for an equivalent nonload-isolated table, but the lock acquired is an EXCLUSIVE lock and concurrent read operations are blocked.

If you explicitly use WITH CONCURRENT ISOLATED LOADING, then the SQL operation is a CLDI and is treated as load operation, which is useful if you want to override a session setting of FOR NO ISOLATED LOADING.

All DML statements in a session that disable load operations default to NCLDI modifications.

The following table describes how DML statement operations that do not contain explicit WITH CONCURRENT ISOLATED LOADING or WITH NO CONCURRENT ISOLATED LOADING syntax are handled by the system.

| Description of Situation or Syntax Used                                                                                                                                | Operation Type |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|
| The table is part of a load operation.                                                                                                                                 | CLDI DML       |
| The DML operation requires an all-AMP table-level/partition-level write lock.                                                                                          | CLDI DML       |
| The session disables concurrent load-isolated operations.                                                                                                              | NCLDI DML      |
| The DML operation is from a load utility such as FastLoad, MultiLoad, Teradata PT load operator, or Teradata PT update operator that does not use the MLOADX protocol. | NCLDI DML      |

| Description of Situation or Syntax Used                                                                     | Operation Type |
|-------------------------------------------------------------------------------------------------------------|----------------|
| The table is set to FOR NONE as its load-isolated DML level.                                                | NCLDI DML      |
| The table is set to FOR INSERT as DML level and the DML is not INSERT or MERGE-INTO with an INSERT portion. | NCLDI DML      |
| The DML operation does not require an all-AMP table- or partition-level WRITE lock.                         | NCLDI DML      |

## Locking Rules

The system always uses an EXCLUSIVE lock for NCLDI DML on a load-isolated table. A concurrent read operation on the table at the same lock level and involved AMPs is always blocked until the associated transaction releases the EXCLUSIVE lock.

For instances where loading is not disabled in the session, the following rules apply:

- The first CLDI DML on a table or join index sets the table or join index to an implicit transaction-based load operation.
- If the first CLDI DML on a load-isolated table or join index in the transaction is not all-AMP or is all AMP but not a table-level lock-based operation, then the operation is forced to acquire an all-AMP and table-level lock.
- You cannot use both CLDI and NCLDI DML on the same table in the same transaction. This rule means that all subsequent load-isolated types for the DMLs must be same as the load-isolated type of the first DML.
- An implicit load on a table or join index prevents any concurrent write operation on the table.

## Example of an Explicit NCLDI Modification

This example illustrates an explicit NCLDI modification:

```
EXPLAIN UPDATE WITH NO ISOLATED LOADING t1 SET c2 = c2 + 1;
*** Help information returned. 12 rows.
*** Total elapsed time was 1 second.
```

Explanation

- 
- 1) First, we lock db1.t1 for exclusive use on a reserved rowHash to prevent global deadlock.
  - 2) Next, we lock db1.t1 for exclusive use.
  - 3) We do an all-AMPs UPDATE (nonconcurrent load isolated) from db1.t1 (Load Uncommitted) by way of an all-rows scan with a condition of `("(db1.t1.TD_ROWLOADID_DEL = 0) AND ((1=1))")`. The size is estimated with low confidence to be 4 rows. The estimated time for this step is 0.07 seconds.
  - 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.  
The total estimated time is 0.07 seconds.

### Example of an Implicit Load Operation

This example illustrates an implicit load operation in an implicit transaction. Notice the new explain element, concurrent load isolated, which indicates that UPDATE versions the rows during modifications. Notice also that the “Begin Isolated Load” starts the load operation, and the “End Isolated Load” step ends the load operation. The TVM is also updated to reflect the committed load ID.

```
EXPLAIN UPDATE t1 SET c2 = c2 + 1;
*** Help information returned. 18 rows.
*** Total elapsed time was 1 second.
Explanation

1) First, we lock db1.t1 for write on a reserved rowHash to prevent
 global deadlock.
2) Next, we lock db1.t1 for write.
3) We Begin Isolated Load on buck.T1.
4) We execute the following steps in parallel.
 1) We do an all-AMPs UPDATE (concurrent load isolated) from
 db1.t1 (Load Uncommitted) by way of an all-rows scan with a
 condition of "(db1.t1.TD_ROWLOADID_DEL = 0) AND ((1=1))".
 The size is estimated with low confidence to be 4 rows. The
 estimated time for this step is 0.07 seconds.
 2) We lock DBC.TVM for write on a RowHash, and then we do a
 single-AMP UPDATE from DBC.TVM by way of the unique primary
 index "Field_1 = '00000204'XB, Field_2 = 'T1'" with no
 residual conditions.
5) We End Isolated Load.
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.
```

### Example of an Implicit NCLDI Modification

This examples illustrates an implicit NCLDI modification and assumes that no other operation on t1 is taking place and t1 is not already in an explicit load state:

```
BEGIN TRANSACTION ;
EXPLAIN UPDATE t1 SET c2 = C2 +1 WHERE c1= 5;
*** Help information returned. 6 rows.
*** Total elapsed time was 1 second.
Explanation

```

```

1) First, we do a single-AMP UPDATE (nonconcurrent load isolated)
 from db1.t1 (Load Uncommitted) by way of the primary index
 "db1.t1.c1 = 5" with a residual condition of (
 "(db1.t1.TD_ROWLOADID_DEL = 0) AND ((1=1))"). The size is
 estimated with low confidence to be 2 rows. The estimated time
 for this step is 0.06 seconds.
-> No rows are returned to the user as the result of statement 1.
 The total estimated time is 0.06 seconds.

```

## CLDI and NCLDI Modifications

All existing error logging rules apply to each of the qualified rows that the CLDI DML modifies, with the following exceptions:

- The data row that results from a modification of a child non-load-isolated table using MLOADX update is rolled back when an RI violation occurs. The relevant information is logged in the error table. If the RI error results from an MLOADX update to a child load-isolated table, the error is logged, but the transaction is rolled back rather than the offending row.
- The data row that results from a modification of a non-load-isolated table using MLOADX Update is rolled back when a uniqueness violation occurs. The relevant information is logged in the error table. If the uniqueness violation error results from an MLOADX update to a load-isolated table, the error continues to be logged, but the transaction is rolled back rather than the offending row.
- In general, when errors occur during the modified row processing of CLDI updates that are performed by using MLOADX with error logging enabled, data rows are not rolled back. The associated transaction is rolled back instead.

All NCLDI modifications follow the rules for existing modifications on a non-load-isolated table.

For CLDI INSERTs, the row to be inserted is inserted. For CLDI DELETEs, if the row to be deleted is a row inserted in the same load operation, then the row is deleted. Otherwise, the row to be deleted is marked and then the row is updated in the table. A CLDI Update on the row is performed as follows:

- If the row to be updated is not a newly inserted row in the load, then a CLDI DELETE is performed on the existing row and a CLDI INSERT is performed on the modified row with the updated values.
- If the row to be updated is a row inserted in the current load operation, then the same row is modified with the updated values.

## Lock Manager

Any number of users and applications can simultaneously access data stored in a database.

The Lock Manager imposes concurrency control on Vantage by managing the locks on the database objects 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 exceptions of global deadlock prevention and detection, locks are not managed globally, but locally by each individual AMP.

For information about global deadlock detection, see [Deadlock](#).

While the Parser defines the locks for a request automatically, you can upgrade, and in some cases, downgrade locks explicitly by using the SQL LOCKING request modifier. For more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## Locking Considerations

Vantage 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.

Users have no control over when locks are released because of the way two-phase locking works (see [Database Locks, Two-Phase Locking, and Serializability](#)).

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 partitions or 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
- A partition of the table
- 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. The Transaction Manager aborts any transaction that requests a lock when Lock Manager resources are exhausted. In such cases, you can disable rowhash-level locking for DDL requests. There is also a per AMP limit on the number of locks that can be placed at a time.

If an application begins a transaction and then performs a large number of single-row updates without closing that transaction, at least one AMP lock table eventually fills to its maximum capacity. This typically occurs for, but is not limited to, transactions in ANSI session mode.

Such an AMP lock table overflow also affects other transactions because no request can begin without first acquiring one or more locks on its underlying database objects. After a while, transactions that are unable to acquire any locks abort, and can even cause the system to crash.

You can control the maximum number of rowhash-level locks that the Lock Manager allows for each transaction using the DBS Control field MaxRowHashBlocksPercent (for more information, see *Teradata Vantage™ - Database Utilities*, B035-1102).

The default for this field is 50 percent of the total number of control blocks that an AMP lock table can support. You can set the value for MaxRowHashBlocksPercent to up to 100% to accommodate the number of rowhash-level locks that your system workloads require.

Vantage automatically aborts a transaction when the number of rowhash-level locks it acquires exceeds the threshold defined by `MaxRowHashBlocksPercent`. No other transactions are affected when this occurs. This control over the number of rowhash-level locks held by a transaction applies only to ANSI session mode and explicit Teradata session mode transactions. Implicit Teradata mode transactions are not affected because each implicit Teradata mode request is a transaction in itself, and the system releases its locks immediately after it commits or rolls back.

You can review all the active locks and determine which other user locks are blocking your transactions using the Lock Display (lokdisp) utility (see *Teradata Vantage™ - Database Utilities*, B035-1102) or the Lock Viewer Viewpoint portlet.

### Once Placed, Locks Are Not Released Until A Transaction Completes

A database lock placed as part of a transaction is held during processing of the transaction and are not released until one of the following events occurs:

- The transaction commits.
- The transaction aborts and has completed its rollback.

ABORT/ROLLBACK statements, asynchronous aborts, failures, time outs, log offs, and system restarts can cause a transaction to abort.

This lock release occurs regardless of when you receive the response to a request because the spool might exist after the end of the transaction. Each of these actions also drops the Transient Journal and closes any open cursors.

During system restart, only update transactions that were in progress at the time of the crash need to be aborted and rolled back. WRITE and EXCLUSIVE locks remain in place for those transactions until they are rolled back.

### Using Multistatement Requests to Minimize How Long Locks Are Held

When possible, you should group individual requests that access the same set of tables or views together using multistatement requests. For example, suppose you want to insert a row into a table, update another row in the same table, and then delete a third row from that table.

You could do this using the following individual requests that insert a row into `cust_rate`, update an existing row in `cust_rate`, delete a third row from `cust_rate`, execute a macro that does something to `cust_rate`, and then update the `cust_rate` row that was just touched by the `init_cust` macro:

```
INSERT INTO cust_rate
VALUES (123, "GOOD");

UPDATE cust_rate
SET cust_rating='FAIR'
WHERE cust_id=456;

DELETE FROM cust_rate
WHERE cust_id=789;
```

```
EXECUTE init_cust (9999);

UPDATE cust_rate
SET cust_rating='FAIR'
WHERE cust_id=9999;
```

In this case, you are submitting only one SQL statement per request, which is inefficient for minimizing request blocking.

If you were to instead submit a multistatement request that executed the identical SQL, Vantage can impose the same row-hash-level WRITE lock on *cust\_rate* just twice, freeing the table for faster access by other concurrently running transactions. The multistatement requests used to do that would be optimally sequenced in the following order:

```
INSERT INTO cust_rate
VALUES (123, 'GOOD')
; UPDATE cust_rate
 SET cust_rating='FAIR'
 WHERE cust_id=456
; DELETE FROM cust_rate
 WHERE cust_id=789;

EXECUTE init_cust (9999)
; UPDATE cust_rate
 SET cust_rating='FAIR'
 WHERE cust_id=9999;
```

---

**Note:**

It is possible to combine these two multistatement requests into an even more efficient explicit multiple request transaction that only sets one row-hash-level WRITE lock on *cust\_rate*.

---

```
BEGIN TRANSACTION
; INSERT INTO cust_rate
 VALUES (123, 'GOOD')
; UPDATE cust_rate
 SET cust_rating='FAIR'
 WHERE cust_id=456
; DELETE FROM cust_rate
 WHERE cust_id=789;

EXECUTE init_cust (9999)
; UPDATE cust_rate
```

```

SET cust_rating='FAIR'
WHERE cust_id=9999;
; END TRANSACTION;

```

## Database Locking Levels and Severities

Database locks have two dimensions: level and severity. 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 rowkey-level lock.

The selection of lock granularity is always a trade-off between the conflicting demands of concurrency and overhead. Concurrency increases as the choice of locking level becomes increasingly granular. Exerting a rowhash- or rowkey-level lock permits more users to access a given table than exerting a table-level lock on the same table. This is why Vantage provides multiple levels of locking granularity.

The severity of a lock refers to its degree of restrictiveness or exclusivity, such as a WRITE lock being more restrictive than an ACCESS lock, or an EXCLUSIVE lock being more restrictive than a READ lock. See also [Default Lock Assignments and Lock Upgradeability](#).

### About 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 coarsest granularity and rowkey-level locks having the 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

See [Proxy Locks](#) for a description of a special category of table-level locking.

- View
- Partition
- RowHash
- RowKey (Partition and RowHash)

| Lock Level ... | What is Locked                                                                                                                                                                                                                             |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Database       | all rows of all tables in the specified database and their associated secondary index subtables.                                                                                                                                           |
| Table          | all rows in the specified base table and in any secondary index and fallback subtables associated with it.                                                                                                                                 |
| View           | all underlying tables accessed by the specified view.                                                                                                                                                                                      |
| Partition      | the primary and fallback copy of rows in a partition for the specified table or single-table view. The table must be row partitioned.<br>This lock permits other users to access the data in the table that are not in the same partition. |



| Lock Level ...              | What is Locked                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PartitionRange              | <p>the primary and fallback copy of rows in a range of partitions for the specified table or single-table view. The table must be row partitioned.</p> <p>This lock permits other users to access the data in the table that are outside the specified partition range.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| RowHash                     | <p>the specified primary or fallback copy of rows sharing the same row hash value for the specified table or single-table view. For a row-partitioned table, this lock level applies to the row hash value for all partitions.</p> <p>The rowhash-level lock permits other users to access the data in the table that do not have the same rowhash.</p> <p>The rowhash-level lock applies to a set of rows that shares the same hash code. It does not necessarily lock only a single row, since multiple rows may have the same rowhash.</p> <ul style="list-style-type: none"> <li>• A rowhash-level lock is applied whenever a non-row-partitioned table is accessed by using a unique primary index (UPI) or a nonunique primary index (NUPI).</li> <li>• For an update or delete that accesses a data row by using a unique secondary index (USI), the appropriate rowhash of the USI subtable is locked, as well as the indexed data rowhash or rowkey.</li> <li>• Rowhash locks on a table's nonunique secondary index (NUSI) subtables are usually not needed. First, a query or DML that uses a NUSI access path locks the whole table. Second, DML that does not lock the whole table uses task locks rather than rowhash locks on any NUSI subtables that require index maintenance.</li> </ul> |
| RowHash in a PartitionRange | <p>the specified primary or fallback copy of rows sharing the same row hash value for the specified table or single-table view in a range of partitions. The table must be row partitioned.</p> <p>This lock permits other users to access other data in the table that do not have the same rowhash or are outside the specified partition range.</p> <p>The rowhash-level lock applies to a set of rows that shares the same hash code. It does not necessarily lock only one row since multiple rows may have the same rowhash in the same partition or in more than one partition in the partition range.</p> <p>This lock level is not used on rows in a USI or NUSI index subtable, as these subtables are never partitioned.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| RowKey                      | <p>the specified primary or fallback copy of rows sharing the same rowkey (partition and row hash value) for the specified table or single-table view. The table must be row partitioned.</p> <p>A rowkey-level lock permits other users to access other data in the table that do not have the same rowhash or partition value.</p> <p>A rowkey-level lock applies to a set of rows that shares the same partition and rowhash. It does not necessarily lock only one row since there could be multiple rows with the same rowhash in a partition.</p> <ul style="list-style-type: none"> <li>• A rowkey-level lock is applied whenever a row-partitioned table with a primary index (UPI or NUPI) is accessed by specifying the primary index and partitioning column values.</li> <li>• The rowkey-level lock is not used on rows in a USI or NUSI subtable, as these subtables are never partitioned.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                       |

The locking level determines whether other users can access the target object.

Locking severities and locking levels 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.

## About Locking Severity

The available lock severities, from most restrictive to least restrictive, are described in the following table.

| Lock Severity                                                                                          | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Most Restrictive                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| EXCLUSIVE                                                                                              | <p>EXCLUSIVE locks are placed only on a database or table when the object is undergoing structural changes (for example, a column is being created or dropped).</p> <p>You can also place an EXCLUSIVE lock explicitly using the LOCKING request modifier (see <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146).</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| WRITE                                                                                                  | <p>Placed in response to an INSERT, UPDATE, or DELETE request.</p> <p>A WRITE lock restricts access by other users (except for applications that are not concerned with data consistency and choose to override the automatically applied WRITE lock by specifying a less restrictive ACCESS lock).</p> <p>You can also place this lock explicitly using the LOCKING request modifier (see <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146).</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| READ                                                                                                   | <p>A READ lock is placed in response to a SELECT request and restricts access by users who require EXCLUSIVE or WRITE locks.</p> <p>Several users can hold READ locks on a resource, during which the system permits no modification of that resource. READ locks ensure consistency during READ operations such as those that occur during a SELECT statement.</p> <p>You can also place the READ lock explicitly using the LOCKING request modifier (see <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146).</p>                                                                                                                                                                                                                                                                                                                                                                                                |
| The CHECKSUM and ACCESS locking severities are all at the same level in the restrictiveness hierarchy. |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| CHECKSUM                                                                                               | <p>Placed in response to a user-defined LOCKING FOR CHECKSUM modifier (see <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146) when using cursors in embedded SQL.</p> <p>CHECKSUM locking is identical to ACCESS locking except that it adds checksums to the rows of a spool 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.</p> <p>See also <a href="#">Cursor Locking Modes</a>, <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146, and <i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i>, B035-1148.</p>                                                                                                                                                                                                                                                               |
| ACCESS                                                                                                 | <p>Placed in response to a user-defined LOCKING FOR ACCESS modifier (see <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146), or by setting the session default isolation level to READ UNCOMMITTED using the SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement (see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144).</p> <p>Permits a user to have a form of read access to an object that might already be locked for READ or WRITE. An ACCESS lock does not restrict access by another user except when an EXCLUSIVE lock is required; therefore it is sometimes referred to as a <i>dirty READ lock</i>.</p> <p>The global application of ACCESS locking for read operations when SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is set to READ UNCOMMITTED depends on the setting of the DBS Control field AccessLockForUncomRead.</p> |

| Lock Severity     | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                   | <p>When the parameter is set FALSE, SELECT operations within INSERT, DELETE, MERGE, and UPDATE requests set READ locks, while when the parameter is set TRUE, the same SELECT operations set ACCESS locks.</p> <p>A user requesting an ACCESS lock disregards all data consistency issues. Because ACCESS and WRITE locks are compatible, the data might be undergoing updates while the user who requested the access is reading it. Therefore, any query that requests an ACCESS lock might return incorrect or inconsistent results.</p> <p>An ACCESS lock is also placed in response to a user-defined LOAD COMMITTED locking modifier (see <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146).</p> <p>For information about designating tables for ISOLATED LOADING, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</p> <p>For a load-isolated table, this modifier allows users to read from committed rows in tables, even while the table is being loaded with data. If the table that is being read is not a load-isolated table, this severity results in an ACCESS lock.</p> <p>For information about load isolation, see <a href="#">Load Isolation</a>.</p> |
| Least Restrictive |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

## Compatibility Among Locking Severities

The Teradata Lock Manager controls the interaction of following types of lock, when placed at specific levels:

- ACCESS.
- CHECKSUM.
- READ.
- WRITE.
- EXCLUSIVE.

The following notation is used to describe the locking severity compatibilities.

| Notation                        | Definition                                         |
|---------------------------------|----------------------------------------------------|
| <i>Q</i>                        | A locking queue associated with a database object. |
| <i>L</i>                        | 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 requests perform a locking operation before they access any database objects.

| IF lock(<object>,<lock requested>) is ... | THEN ...                                                                                                                                                                                                                    |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| queued for any lock in <i>L</i>           | <p>the transaction is placed in <i>Q</i> and waits there as long as lock(&lt;object&gt;,&lt;lock requested&gt;) is queued.</p> <p>A request in this state is said to be blocked (see <a href="#">Blocked Requests</a>).</p> |

| IF lock(<object>,<lock requested>) is ... | THEN ...                                                                                                    |
|-------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| 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 table on the following page 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 CHECKSUM                                | READ                                                                                                                                 | WRITE                                                                                                                                | EXCLUSIVE                                                                                                                            |
| ACCESS CHECKSUM            | Lock Granted          | Lock Granted                                   | Lock Granted                                                                                                                         | Lock Granted                                                                                                                         | Request Queued<br>If you specify a LOCKING FOR NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing. |
| READ                       | Lock Granted          | Lock Granted                                   | Lock Granted                                                                                                                         | Request Queued<br>If you specify a LOCKING FOR NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing. | Request Queued<br>If you specify a LOCKING FOR NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing. |
| WRITE                      | Lock Granted          | Lock Granted                                   | Request Queued<br>If you specify a LOCKING FOR NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing. | Request Queued<br>If you specify a LOCKING FOR NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing. | Request Queued<br>If you specify a LOCKING FOR NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing. |
| EXCLUSIVE                  | Lock Granted          | Request Queued<br>If you specify a LOCKING FOR | Request Queued<br>If you specify a LOCKING                                                                                           | Request Queued<br>If you specify a LOCKING FOR                                                                                       | Request Queued<br>If you specify a LOCKING FOR NOWAIT request                                                                        |

| Severity of Requested Lock | Severity of Held Lock |                                                                                       |                                                                                           |                                                                                       |                                                                        |
|----------------------------|-----------------------|---------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|------------------------------------------------------------------------|
|                            | None                  | ACCESS CHECKSUM                                                                       | READ                                                                                      | WRITE                                                                                 | EXCLUSIVE                                                              |
|                            |                       | NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing. | FOR NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing. | NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing. | modifier, the transaction aborts if it is blocked instead of queueing. |

Because other client utilities such as BTEQ, FastExport, FastLoad, MultiLoad, Teradata Parallel Data Pump, and Teradata Parallel Transporter use standard database locks, the interactions of those locking severities with those of other database locks are identical. See the following manuals for details of the locks set by those utilities:

- *Basic Teradata® Query Reference*, B035-2414
- *Teradata® FastExport Reference*, B035-2410
- *Teradata® FastLoad Reference*, B035-2411
- *Teradata® MultiLoad Reference*, B035-2409
- *Teradata® Parallel Data Pump Reference*, B035-3021
- *Teradata® Parallel Transporter Reference*, B035-2436

A queued request is in an I/O wait state and is said to be blocked (see [Blocked Requests](#)).

A WRITE or EXCLUSIVE lock on a database, table, or view restricts all requests or transactions except the one holding the lock from accessing data within the domain of that object.

Because a lock on an entire database can restrict access to a large quantity of data, the Parser 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 Parser attempts to lock dictionary tables at the rowhash or rowkey (partition and rowhash) level whenever possible.

For information about how load isolation affects compatibility among locking severities, see [Load Isolation](#).

### Using the NOWAIT Option for the SQL LOCKING Request Modifier

When you specify the NOWAIT option for the SQL LOCKING request modifier, Vantage aborts a transaction that makes a lock request that cannot be fulfilled immediately. For details on how to use the NOWAIT option with the LOCKING request modifier, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Vantage uses a slight variation of this code internally to avoid blocking on DDL operations. Instead of aborting the request, the system instead downgrades rowhash lock severities from READ to ACCESS. See [DDL and DCL Requests, Dictionary Access, and Locks](#).

## AMP-Based Utilities and Logging

Following are the utilities related to logging:

- Lock Viewer Viewpoint portlet

This portlet produces a report of miscellaneous database lock delay information that you can use to detect blocked transactions and global deadlocks. Vantage extracts the data reported by the Lock Viewer portlet from various DBQL transaction logs.

## Client Utility Locks

The Teradata Tools and Utilities programs that perform tasks such as bulk data loads (FastLoad, MultiLoad, BTEQ), bulk data exports (FastExport, BTEQ), stream data loads (Teradata Parallel Data Pump), both data loads and data exports (Teradata Parallel Transporter) place locks at different levels and severities on database resources. See [About Locking Levels](#) and [About Locking Severity](#) for details of locking levels and severities.

### Locks Placed on Database Resources by Teradata Tools and Utilities

The client utilities place the following basic types of locks on database resources:

- Standard database locks

All of the client data loading and exporting utilities use standard database transaction locks (see [Database Locks, Two-Phase Locking, and Serializability](#)).

For details about when the various database locks that client utilities place and release on resources in the database, see the following Teradata Tools and Utilities manuals:

- *Basic Teradata® Query Reference*, B035-2414
- *Teradata® FastExport Reference*, B035-2410

Teradata Parallel Transporter uses the FastExport protocol for its EXPORT operator (see *Teradata® Parallel Transporter Reference*, B035-2436 for details).

- *Teradata® FastLoad Reference*, B035-2411

Teradata Parallel Transporter uses the FastLoad protocol for its LOAD operator (see *Teradata® Parallel Transporter Reference*, B035-2436 for details).

- *Teradata® MultiLoad Reference*, B035-2409

Teradata Parallel Transporter uses the MultiLoad protocol for its UPDATE operator (see *Teradata® Parallel Transporter Reference*, B035-2436 for details).

- *Teradata® Parallel Data Pump Reference*, B035-3021

Teradata Parallel Transporter uses the Teradata Parallel Transporter protocol for its STREAM operator (see *Teradata® Parallel Transporter Reference*, B035-2436 for details).

- *Teradata® Parallel Transporter Reference*, B035-2436

## Default Lock Assignments and Lock Upgradeability

Vantage assigns locking levels and severities to SQL requests by default.

When necessary, Vantage 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 Vantage commits the transaction.

### Note:

For load-isolated tables, a WRITE lock is used for concurrent load-isolated modifications and an EXCLUSIVE lock is used for nonconcurrent load-isolated modifications. For details about concurrent and nonconcurrent load-isolated modifications, see [Load Isolation](#).

### About Default Lock Assignments

The following table lists some of the default lock assignments for various SQL requests and their access strategies.

| SQL Statement                                       | Access Type    |                                                                                                           | Default Lock Type Assigned                                                                                                 |
|-----------------------------------------------------|----------------|-----------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
|                                                     | UPI or USI     | NUSI or FTS                                                                                               |                                                                                                                            |
| CREATE DATABASE<br>DROP DATABASE<br>MODIFY DATABASE | Not applicable | Database                                                                                                  | EXCLUSIVE                                                                                                                  |
| CREATE TABLE<br>DROP TABLE<br>ALTER TABLE           | Not applicable | Table                                                                                                     | EXCLUSIVE                                                                                                                  |
| DELETE                                              | row hash       | Table                                                                                                     | WRITE                                                                                                                      |
| INSERT                                              | row hash       | Not applicable                                                                                            | WRITE                                                                                                                      |
| MERGE                                               | row hash       | <ul style="list-style-type: none"> <li>• Not applicable for INSERT</li> <li>• Table for UPDATE</li> </ul> | WRITE                                                                                                                      |
| SELECT                                              | row hash       | Table                                                                                                     | READ<br>If a SELECT operation is part of a DDL operation, the system might downgrade a rowhash READ lock to an ACCESS lock |



| SQL Statement         | Access Type |             | Default Lock Type Assigned                                                                                                                                                                                                    |
|-----------------------|-------------|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                       | UPI or USI  | NUSI or FTS |                                                                                                                                                                                                                               |
|                       |             |             | to avoid blocking. For more information, see <a href="#">DDL and DCL Requests, Dictionary Access, and Locks</a> .                                                                                                             |
| SELECT<br>AND CONSUME | Row hash    | Row hash    | WRITE<br>Vantage 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. |
| UPDATE                | row hash    | Table       | WRITE                                                                                                                                                                                                                         |

Load-isolated tables lock defaults differ somewhat from standard defaults. Nonconcurrent load-isolated modifications that use INSERT/DELETE/UPDATE/MERGE statements block the reader sessions from selecting committed rows. Nonconcurrent load-isolated modifications performed using SQL use EXCLUSIVE locks instead of WRITE locks.

For more information, see [Load Isolation](#).

### Changing Lock Assignments Using the LOCKING Request Modifier

Depending on the assigned lock and the individual SQL request, you can change some default lock assignments using the LOCKING request modifier (for details, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146). You can upgrade any lower severity lock to a higher severity lock, but the only downgrade permitted is from a READ lock to an ACCESS lock.

The following table provides a summary of the allowable changes for database locks.

| A change from this default assignment lock ...                                                                                                                                                                                                                                                       | TO this user-specified lock ... | IS ...                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|-----------------------|
| ACCESS<br>Even though a CHECKSUM lock is essentially identical to an ACCESS lock, you can only specify CHECKSUM locks explicitly using the LOCKING request modifier; you cannot upgrade them to a higher severity lock. This is because Vantage never specifies CHECKSUM as a default lock severity. | ACCESS                          | redundant, but valid. |
| READ                                                                                                                                                                                                                                                                                                 | READ                            |                       |
| WRITE                                                                                                                                                                                                                                                                                                | WRITE                           |                       |
| EXCLUSIVE                                                                                                                                                                                                                                                                                            | EXCLUSIVE                       |                       |
| ACCESS                                                                                                                                                                                                                                                                                               | READ                            | a valid upgrade.      |
|                                                                                                                                                                                                                                                                                                      | WRITE                           |                       |



| A change from this default assignment lock ... | TO this user-specified lock ... | IS ...             |
|------------------------------------------------|---------------------------------|--------------------|
|                                                | EXCLUSIVE                       |                    |
| READ                                           | WRITE                           |                    |
|                                                | EXCLUSIVE                       |                    |
| WRITE                                          | EXCLUSIVE                       |                    |
| READ                                           | ACCESS                          | a valid downgrade. |

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 rowhash, view, and table database object levels:

| This LOCKING request modifier severity specification ... | Is available for this SQL DML statement ...                                                                                       |
|----------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| EXCLUSIVE                                                | <ul style="list-style-type: none"> <li>• DELETE</li> <li>• INSERT</li> <li>• MERGE</li> <li>• SELECT</li> <li>• UPDATE</li> </ul> |
| WRITE                                                    | <ul style="list-style-type: none"> <li>• SELECT</li> <li>• SELECT AND CONSUME</li> </ul>                                          |
| READ                                                     | SELECT                                                                                                                            |
| ACCESS                                                   | SELECT                                                                                                                            |

The reason you can only specify the LOCKING FOR EXCLUSIVE modifier for DELETE, INSERT, MERGE, and UPDATE requests 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.

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 *Teradata Vantage™ - Database Utilities*, B035-1102 for details on how to use Query Session).

### **Guidelines for Changing Default Lock Assignments and Changing Intratransaction Request Orders to Maximize Concurrency**

The following set of guidelines lists some rules of thumb for maximizing concurrency with your database transactions.

- Use ACCESS locks in place of READ locks whenever an application can tolerate dirty reads.

If you have load-isolated tables, however, you can use the LOAD COMMITTED locking modifier to read committed rows without being blocked and without blocking the concurrent isolated modifications. Using load-isolated tables allows you to obtain committed reads instead of dirty reads. For more information about load-isolated tables, see [Load Isolation](#).

- A SELECT request that requires a READ lock on a table cannot run concurrently with an executing CREATE INDEX or ALTER TABLE ... FALLBACK request for the same table.

Instead, specify a READ lock for the CREATE INDEX or ALTER TABLE request to permit concurrency (for details, see the information about the LOCKING request modifier in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).

- If the CREATE INDEX or ALTER TABLE ... FALLBACK LOCKING request modifier specifies WRITE (or if there is no LOCKING request modifier specified), then specify an ACCESS lock in a LOCKING request modifier on your SELECT request to permit concurrency (for details, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).

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. For details, see the information about the LOCKING request modifier in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- Be aware that when running long transactions concurrently with CREATE INDEX or ALTER TABLE ... FALLBACK, the CREATE INDEX or ALTER TABLE request might not complete until the long running transactions have completed.
- To avoid the chance of deadlocks with other DML transactions or DDL requests, 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 requests as early as possible in a transaction to avoid conflicts with other database resources. This is to minimize the likelihood of a situation where a

SELECT AND CONSUME TOP 1 request would enter a delayed state while holding locks on resources that might be needed by other requests.

## Blocked Requests

A request that is waiting in a lock queue is considered to be blocked (see [Compatibility Among Locking Severities](#)). A consume mode SELECT request 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 [Definition of an Explicit Transaction](#).

If you suspect that a request is blocked, you can use the Query Session utility (see *Teradata Vantage™ - Database Utilities*, B035-1102) 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 Deadlocking Are Different

When the Lock Manager places a request in a lock queue, it is known that the request will execute as soon as it arrives at the head of the queue. Blocked requests do not time out; they remain in the queue as long as it takes to reach its head, at which time they are granted the locks they are waiting for and execute.

The only requests the Lock Manager never enqueues are those explicitly specified with a LOCKING request modifier and the NOWAIT option (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146). 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.

A deadlock is a lock contention situation that cannot be solved without intervention of some kind (see [Deadlock](#)). For Vantage, the intervention is accomplished by aborting the younger request in the deadlock.

Note that a retryable error is reported for a request that is aborted by a deadlock.

### Dealing with Multirequest Transactions that Require Lock on Same Table

When several requests that compete for the same table are submitted as separate, single-request transactions, Vantage 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.

## Resolving Blocked Requests

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.

## Dealing with Multirequest Transactions

Explicit multirequest transactions should also be reviewed for any scheduling concerns.

When competing locks are needed by multiple requests in a single transaction, Vantage automatically upgrades the locks for each request in turn, until the transaction is completed.

This handling protects an active transaction from being interrupted by new arrivals. However, a blocked queue can still result if the active transaction has many requests or demands excessive processing time.

For example, consider the following scenario:

| Stage | This transaction number ... | Process                    |                             |                                                                                                                                           |
|-------|-----------------------------|----------------------------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
|       |                             | Has this request ...       | That requires this lock ... | With this result ...                                                                                                                      |
| 1     | 1                           | SELECT ...<br>FROM table_a | READ                        | The READ lock on table_a is granted and the SELECT request begins processing.                                                             |
| 2     | 2                           | INSERT<br>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                           |                            |                             | Processing of the SELECT request from Transaction 1 completes.                                                                            |
|       |                             | INSERT<br>INTO table_a ... | WRITE                       | The READ lock on table_a is upgraded to a WRITE lock for the Transaction 1 INSERT request.                                                |

| Stage | Process                     |                      |                             |                                                                                                         |
|-------|-----------------------------|----------------------|-----------------------------|---------------------------------------------------------------------------------------------------------|
|       | This transaction number ... | Has this request ... | That requires this lock ... | With this result ...                                                                                    |
|       |                             |                      |                             | Transaction 2 remains queued and inactive, waiting for its WRITE lock request on table_a to be granted. |

## Proxy Locks

Proxy locks provide a mechanism for queueing locks to avoid the global deadlocks that can otherwise occur when Teradata places a lock on each of the AMPs in a parallel system (see [Deadlock](#)).

When you make an all-AMP request for a READ, WRITE, or EXCLUSIVE lock, or for an ACCESS lock on a load-isolated table (see [Load Isolation](#)), the system automatically places a proxy lock on a single AMP before it places the all-AMP lock. Think of the proxy lock as an intention lock, that is, it indicates that the request intends to place a lock on each of the AMPs.

### About Proxy Locks

A proxy lock enables sequential locking on database objects that span multiple AMPs in a parallel database architecture. Without a proxy lock, 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 in different sequential orders at the various AMPs holding the table rows. Each request then locks the rows that belong to that table on different AMPs, which creates a deadlock situation.

For example, suppose a request from user\_1 locks table rows on AMP 3, while user\_2 locks the table rows on AMP 4 first. When the user\_1 request attempts to lock table rows on AMP 4, or when the user\_2 request attempts to lock table rows on AMP 3, a global deadlock occurs. A proxy lock prevents such deadlocks from occurring.

Proxy locking has the following properties:

- Each table has a system-assigned table ID that can be associated with a unique hash value. This hash value identifies a single AMP, called the gatekeeper AMP, on which to place the proxy lock for this table.
- The table ID hash values are evenly distributed across the AMPs so that proxy locks are not set on one AMP. However, for the same table, the same AMP is always used.
- An all-AMP step for a READ, WRITE, or EXCLUSIVE lock (or an ACCESS lock for load-isolated tables) is always preceded by a single-AMP step that is sent to the relevant gatekeeper AMP to place a proxy lock.
- The gatekeeper AMP places a rowhash lock on the table by using a reserved hash code value of 0xFFFFFFFF. This reserved hash value is a value that cannot be generated by the hashing function.
- For a row-partitioned table, a proxy lock may be placed on a partition with a reserved internal partition number of 0xFFFFFFFFFFFFFFFF. This reserved internal partition number is a value that cannot be generated for actual partitions. The reserved partition is used to prevent more than one writer to a table.

## Table-Level Proxy Locking

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 rowhash lock on the table using the reserved hash value.
4. Because the table is not currently locked, the user\_1 request obtains the requested reserved rowhash lock and proceeds to obtain a lock on each of the AMPs.
5. Meanwhile, user\_2 submits an all-AMP request for the same table.
6. The request-originating PE sends a message to the gatekeeper AMP for the table.
7. The gatekeeper AMP attempts to place a rowhash lock on the table using the reserved hash value.
8. Because user\_1 already has the reserved rowhash locked for the table, the request from user\_2 must wait in a queue until the request submitted by user\_1 releases its locks on the table. Because the user\_2 request was the next in sequence to request a proxy lock on the table, it is next in the queue to lock the table for processing.

Vantage handles all-AMP lock requests as follows:

1. The PE that processes an all-AMPs request uses the table ID hash value to determine the gatekeeper AMP where the proxy lock is to be set for the table.
2. The first request to place a proxy lock acquires locks on the table across all AMPs.

The following example is an EXPLAIN report for a simple SELECT from the nonpartitioned table named t4.

```
EXPLAIN SELECT * FROM t4;
Explanation
```

- ```
-----
1) First, we  lock PLS.t4 for read on a reserved rowHash  to prevent
   global deadlock.
2) Next, we  lock PLS.t4 for read.
3) We do an all-AMPs RETRIEVE step from PLS.t4 by way of an all-rows
   scan with no residual conditions into Spool 1 (group_amps), which
   is built locally on the AMPs. The size of Spool 1 is estimated
   with low confidence to be 8 rows (344 bytes). 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.
```

Step 1 is a single-AMP lock step that is sent to the gatekeeper AMP for table t4. The gatekeeper AMP is determined by the hash value of the table ID of table t4. The gatekeeper AMP places a rowhash lock using the reserved rowhash on table t4.

Step 2 is an all-AMP lock step that places a table-level lock on table t4 on each AMP.

In the above example, step 1 is a typical proxy lock that is associated with a subsequent table-level lock on the actual table in step 2. This form is always used for a single table. If multiple tables need to be locked by proxy, Vantage places a separate single-AMP proxy lock for each of them before placing the table-level locks. Note that unlike the proxy locks that are placed in individual steps, the table-level locks on multiple tables are all placed in one all-AMP step.

The following examples use a row-partitioned table named t5, which was created as follows:

```
CREATE TABLE t5 (a INTEGER, b INTEGER)
PRIMARY INDEX (a)
PARTITION BY RANGE_N (b BETWEEN 1 AND 100 EACH 1);
```

The following example is an EXPLAIN report for a simple select from table t5. The relevant EXPLAIN text is highlighted in boldface type.

```
EXPLAIN SELECT * FROM t5;
Explanation
-----
1) First, we lock PLS.t5 for read on a reserved rowHash in all
   partitions to prevent global deadlock.
2) Next, we lock PLS.t5 for read.
3) We do an all-AMPs RETRIEVE step from PLS.t5 by way of an all-rows
   scan with no residual conditions into Spool 1 (group_amps), which
   is built locally on the AMPs. The size of Spool 1 is estimated
   with low confidence to be 8 rows (344 bytes). 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.
```

Step 1 is a single-AMP lock step that is sent to the gatekeeper AMP for table t5. The gatekeeper AMP is determined by the hash value of the table ID of table t5. The gatekeeper AMP places a rowhash lock in all partitions using the reserved rowhash on table t5.

Step 2 is an all-AMP lock step that places a table-level lock on table t5 on each AMP.

The following example is an EXPLAIN report for a select from table t5 with a condition on the partitioning column such that only a single partition needs to be accessed.

```
EXPLAIN SELECT * FROM t5 WHERE b = 4;
Explanation
-----
1) First, we lock PLS.t5 for read on a reserved rowHash in a single
   partition to prevent global deadlock.
```

- 2) Next, we lock PLS.t5 for read on a single partition.
 - 3) We do an all-AMPs RETRIEVE step from a single partition of PLS.t5 with a condition of ("PLS.t5.B = 4") with a residual condition of ("PLS.t5.B = 4") 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 (43 bytes). 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.

Step 1 is a single-AMP lock step that is sent to the gatekeeper AMP for table t5. The gatekeeper AMP is determined by the hash value of the table ID of table t5. The gatekeeper AMP places a rowhash lock in a single partition using the reserved rowhash on table t5.

Step 2 is an all-AMP lock step that places a partition lock on table t5 on each AMP.

The following example is an EXPLAIN report for a select from table t5 with a condition that limits access to a range of partitions.

```
EXPLAIN SELECT * FROM t5 WHERE b BETWEEN 3 AND 5;
```

Explanation

-
- 1) First, we lock PLS.t5 for read on a reserved rowHash in all partitions to prevent global deadlock.
 - 2) Next, we lock PLS.t5 for read.
 - 3) We do an all-AMPs RETRIEVE step from 3 partitions of PLS.t5 with a condition of ("(PLS.t5.B <= 5) AND (PLS.t5.B >= 3)") 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 (86 bytes). 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.

Step 1 is a single-AMP lock step that is sent to the gatekeeper AMP for table t5. The gatekeeper AMP is determined by the hash value of the table ID of table t5. The gatekeeper AMP places a rowhash lock in all partitions (since more than one partition must be locked) using the reserved rowhash on table t5.

Step 2 is an all-AMP lock step that places a table-level lock on table t5 on each AMP.

Pseudo Table Locks

You can think of a pseudo table as an alias for the physical table it represents. (Note that the word pseudo modifies the table, and not the lock.) Pseudo tables provide a mechanism for queueing data dictionary rowhash locks to avoid the global deadlocks that can otherwise occur when Teradata places locks on data dictionary rows (see [Deadlock](#)).

When DDL needs to lock a rowhash for both the primary and fallback of a dictionary table, the system automatically places a rowhash-level pseudo table lock. Think of a rowhash-level pseudo table lock as an intention lock, that is, it indicates that the request intends to place a rowhash lock on both the primary and fallback of a dictionary table.

Pseudo table locks enable the sequential locking of rowhashes on both primary and fallback rows that are on different AMPs in a parallel database architecture. Without pseudo table locking, if multiple users simultaneously submit DDL requests that need to place a rowhash lock on both the primary and the fallback of a data dictionary table, a deadlock can occur. If multiple requests are sent in parallel and need to lock the same rowhash of a dictionary table, they are likely to arrive at the primary and fallback AMPs holding the rowhash in different sequential orders. One request may obtain the rowhash lock on the primary AMP first and the other on a fallback AMP first, thus creating a deadlock.

Pseudo table locks are placed on various data dictionary tables at the rowhash-level for the following DDL statements:

- CREATE HASH INDEX
- CREATE JOIN INDEX
- CREATE TABLE
- DROP HASH INDEX
- DROP JOIN INDEX
- DROP MACRO
- DROP PROCEDURE
- DROP TABLE
- DROP VIEW

It would be more accurate to refer to these locks as pseudo rowhash locks rather than rowhash-level pseudo table locks, but the EXPLAIN phrase text does not follow this convention, and this description follows the terminology used by the EXPLAIN phrase text (see [EXPLAIN Request Modifier Phrase Terminology](#)).

Rowhash-Level Pseudo Table Locking

This set of scenarios uses simple CREATE TABLE and DROP TABLE requests to demonstrate how Vantage uses rowhash-level pseudo table locking.

The following example is an EXPLAIN report for a simple CREATE TABLE definition for table t4.

```
EXPLAIN CREATE TABLE t4 (a INTEGER, b INTEGER);
Explanation
```

- ```

```
- 1) First, we lock DB1.T4 in TD\_MAP1 for exclusive use.
  - 2) Next, we lock a distinct DBC."pseudo table" in TD\_DATADITIONARYMAP for read on a RowHash for deadlock prevention, we lock a distinct DBC."pseudo table" in TD\_DATADITIONARYMAP for write on a RowHash for deadlock prevention, and we lock a distinct DBC."pseudo table" in TD\_DATADITIONARYMAP for write on a RowHash for deadlock prevention.
  - 3) We lock DBC.AccessRights in TD\_DATADITIONARYMAP for write on a reserved RowHash in a single partition to prevent global deadlock.
  - 4) We lock DBC.DBase in TD\_DATADITIONARYMAP for read on a RowHash, we lock DBC.Maps in TD\_DATADITIONARYMAP for read on a RowHash, we lock DBC.TVFields in TD\_DATADITIONARYMAP for write on a RowHash, we lock DBC.Indexes in TD\_DATADITIONARYMAP for write on a RowHash, we lock DBC.TVM in TD\_DATADITIONARYMAP for write on a RowHash, and we lock DBC.AccessRights in TD\_DATADITIONARYMAP for write on a single partition.
  - 5) We execute the following steps in parallel.
    - 1) We do a single-AMP ABORT TEST step from DBC.DBase by way of the unique primary index "Field\_1 = 'DB1'" with a residual condition of ("'00001904'XB= DBC.DBase.Field\_2").
    - 2) We do a single-AMP ABORT TEST step from DBC.TVM by way of the unique primary index "Field\_1 = '00001904'XB, Field\_2 = 'T4'".
    - 3) We do an INSERT step into DBC.Indexes (no lock required).
    - 4) We do an INSERT step into DBC.TVFields (no lock required).
    - 5) We do an INSERT step into DBC.TVFields (no lock required).
    - 6) We do a two-AMP ABORT TEST step in TD\_DATADITIONARYMAP from DBC.Maps by way of unique index # 4 "Field\_3 = 1025" with no residual conditions.
    - 7) We do an INSERT step into DBC.TVM (no lock required).
    - 8) We INSERT default rights to DBC.AccessRights for DB1.T4.
  - 6) We create the table header in TD\_MAP1.
  - 7) 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.

CREATE TABLE does not place a proxy lock before it applies the table-level lock for the table in step 1. That is unnecessary because the table does not exist until it has been created, so no other user could be trying to access it.

Step 2 places one pseudo lock for a rowhash in the pseudo table on one AMP. Every DDL request for the same rowhash must then go through this pseudo table, so deadlock does not occur from attempt to lock both the primary and fallback copies. Step 2 is an all-AMP lock step. Each AMP is passed the same list of rowhash locks and only the active primary AMP of the rowhash places the lock.

Step 3 is a proxy lock for AccessRights on a single partition.

Step 4 places rowhash locks on the actual DBase, Maps, TVFields, Indexes, TVM, and AccessRights dictionary tables. Step 4 is an all-AMP step and only the primary and fallback AMPs that own the rowhashes place locks.

Notice that the number of rowhash-level locks placed is fewer in step 2 than in step 4. The EXPLAIN text does not indicate which rowhash-level lock in step 4 corresponds to the lock in the step 2. Furthermore, the rowhash-level locks in step 2 are not placed in exactly the same order as in step 4, based on matching READ and WRITE locks, because of the way the locks are sorted, which is in table/rowhash order. For step 2, the table is the same for all the row hashes, but in step 4 the table is different for each row hash.

Step 2 places fewer rowhash-level locks than step 4 because DBC.TVFields and DBC.Indexes have the same primary indexes, so the rowhash is the same for both tables and only one rowhash-level lock is required on the pseudo table. But in step 4, two actual rowhash-level locks must be placed because they are for separate dictionary tables.

The following example is an EXPLAIN report for a simple DROP TABLE request where no privileges have been granted explicitly to a user or database on the table:

```
EXPLAIN DROP TABLE t4;
```

```
Explanation
```

- 
- 1) First, we lock DB1.t4 in TD\_MAP1 for exclusive use on a reserved RowHash to prevent global deadlock.
  - 2) Next, we lock DB1.t4 in TD\_MAP1 for exclusive use.
  - 3) We lock a distinct DBC."pseudo table" in TD\_DATADictionaryMAP for read on a RowHash for deadlock prevention, we lock a distinct DBC."pseudo table" in TD\_DATADictionaryMAP for write on a RowHash for deadlock prevention, we lock a distinct DBC."pseudo table" in TD\_DATADictionaryMAP for write on a RowHash for deadlock prevention, we lock a distinct DBC."pseudo table" in TD\_DATADictionaryMAP for write on a RowHash for deadlock prevention, and we lock a distinct DBC."pseudo table" in TD\_DATADictionaryMAP for write on a RowHash for deadlock prevention.
  - 4) We lock DBC.AccessRights in TD\_DATADictionaryMAP for write on a reserved RowHash in a single partition to prevent global deadlock.
  - 5) We lock DBC.DBase in TD\_DATADictionaryMAP for read on a RowHash, we lock DBC.TVFields in TD\_DATADictionaryMAP for write on a RowHash, we lock DBC.Indexes in TD\_DATADictionaryMAP for write on a RowHash, we lock DBC.TVM in TD\_DATADictionaryMAP for write on a RowHash, we lock DBC.DBCAssociation in TD\_DATADictionaryMAP for write on a RowHash, we lock DBC.RCEvent in TD\_DATADictionaryMAP for write on a RowHash, we lock DBC.Dependency in TD\_DATADictionaryMAP for write on a RowHash, we lock

DBC.ObjectUsage in TD\_DATADITIONARYMAP for write on a RowHash,  
and we lock DBC.AccessRights in TD\_DATADITIONARYMAP for write on  
a single partition.

- 6) We drop the table header and the data in the table DB1.t4.
- 7) We execute the following steps in parallel.
  - 1) We do a single-AMP ABORT TEST step from DBC.DBase by way of the unique primary index "Field\_1 = 'DB1'" with a residual condition of ("'00001904'XB= DBC.DBase.Field\_2").
  - 2) We do a single-AMP ABORT TEST step from DBC.TVM by way of the unique primary index "Field\_1 = '00001904'XB, Field\_2 = 'T4'" with a residual condition of ("'0000470D0000'XB= DBC.TVM.Field\_5").
  - 3) We do a single-AMP ABORT TEST step from DBC.TVM by way of the unique primary index "Field\_1 = '00001904'XB, Field\_2 = 'T4'" with a residual condition of ("DBC.TVM.Field\_33 > 0").
- 8) We lock DBC.StatsTbl in TD\_DATADITIONARYMAP for write on a RowHash.
- 9) We execute the following steps in parallel.
  - 1) We do a single-AMP DELETE step in TD\_DATADITIONARYMAP from DBC.StatsTbl by way of the primary index "{LeftTable}.Field\_2 = '0000470D0000'XB" with no residual conditions.
  - 2) We do a single-AMP DELETE step in TD\_DATADITIONARYMAP from DBC.TVFields by way of the primary index "Field\_1 = '0000470D0000'XB" with no residual conditions.
  - 3) We do a single-AMP DELETE step in TD\_DATADITIONARYMAP from DBC.Indexes by way of the primary index "Field\_1 = '0000470D0000'XB" with no residual conditions.
  - 4) We do a single-AMP DELETE step in TD\_DATADITIONARYMAP from DBC.DBCAssociation by way of the primary index "Field\_1 = '0000470D0000'XB" with no residual conditions.
  - 5) We do a single-AMP DELETE step in TD\_DATADITIONARYMAP from DBC.TVM by way of the unique primary index "Field\_1 = '00001904'XB, Field\_2 = 'T4'" with no residual conditions.
  - 6) We do a single-AMP DELETE step in TD\_DATADITIONARYMAP from a single partition of DBC.AccessRights by way of the primary index "Field\_1 = '00001904'XB, Field\_2 = '00001904'XB, Field\_3 = '0000470D0000'XB" with a residual condition of ("DBC.AccessRights.Field\_3 = '0000470D0000'XB") (no lock required).
  - 7) We do a single-AMP DELETE step in TD\_DATADITIONARYMAP from DBC.Dependency by way of the primary index "Field\_1 = '0000470D0000'XB" with no residual conditions.
  - 8) We do a single-AMP DELETE step in TD\_DATADITIONARYMAP from DBC.ObjectUsage by way of the primary index "Field\_1 =

```
'00001904'XB, Field_2 = '0000470D0000'XB" with no residual
conditions.
```

```
9) We do an INSERT step into DBC.RCEvent.
```

```
10) We spoil the statistics cache for the table, view or query.
```

```
11) We end logging on DB1.t4.
```

```
12) We spoil the parser's dictionary cache for the table.
```

```
13) 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.
```

Unlike the CREATE TABLE request, multiple users might be trying to access or DROP table t4, so it is necessary to coordinate them using a proxy lock, that is, locking a reserved rowhash in step 1, followed by the lock of table t4 in step 2.

Step 3 places one pseudo lock for a rowhash in the pseudo table on one AMP and every DDL request for the same rowhash must then go through this pseudo table, so deadlock does not occur from attempts to lock both the primary and fallback copies. Step 2 is an all-AMP lock step. Each AMP is passed the same list of rowhash locks and only the active primary AMP of the rowhash places the lock.

Step 4 is a proxy lock for AccessRights on a single partition.

Step 5 places rowhash-level locks on the actual DBase, TVFields, and so on dictionary tables. This is an all-AMPs step, but for a particular rowhash that is mentioned in the step, only the primary and fallback AMPs that own that rowhash place a lock for that rowhash, even though the step is assigned to all AMPs.

For this request, the number of rowhash-level locks placed in step 3 is fewer than the number placed in step 5. The EXPLAIN text does not indicate which rowhash-level lock in step 5 corresponds to the lock in the step 3. Furthermore, the rowhash-level locks in step 3 are not placed in exactly the same order as in step 5 based on matching READ and WRITE locks, because of the way the locks are sorted, which is in table/rowhash order.

Step 3 places fewer rowhash-level locks than step 5 because DBC.TVFields and DBC.Indexes have the same primary indexes, so the rowhash is the same for both tables and only one rowhash-level lock is required on the pseudo table. But two actual rowhash-level locks must be placed in step 5 because they are for separate dictionary tables.

For both the CREATE TABLE and DROP TABLE cases, the locking becomes increasingly more complicated if there are referential integrity constraints defined for the table because of the additional parent and child and dictionary tables that must be locked. Options that involve such things as journals and other dictionary tables that must be locked complicate the picture still further.

## Deadlock

One problem that can occur with two-phase locking (see [Two-Phased Locking Defined](#)) is a situation in which two requests each lock a different database resource that the other request needs to continue processing its transaction. For example, consider the situation where the first transaction has locked table row B and the second transaction has locked table row A. If the first transaction now needs to lock table

row A before it can continue, and the second transaction needs to lock table row B before it can continue, the two transactions have created a deadlock. The transactions that are neutralized by this outcome are in a simultaneous wait state, as each transaction waits for the other transaction to release a locked resource before it can continue.

This is a different situation than the case in which individual requests are waiting for a lock to be released. Note that database lock requests do not time out, so a request waits in the lock queue until it is granted the lock it needs. 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.

For example, a local deadlock occurs when two transactions that concurrently hold READ locks for the same data on the same AMP 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 AMP1, and transaction\_b places a WRITE lock on object\_y on AMP2. Both of these lock requests are granted.

Now, if transaction\_b attempts to place a write lock on object\_x on AMP1, it is blocked. In a similar manner, if transaction\_a places a WRITE lock on object\_y on AMP2, it is also blocked. Neither transaction can complete.

If the DBQL Lock Logger is enabled, then whenever deadlocks occur, they are logged in the XML lock log table, DBQLXMLLockTbl, and in the main log table, DBQLLogTbl. You can use the Viewpoint DBQL Lock Logger portlet to report deadlocks whenever they occur. For information about how to use this portlet, see *Teradata® Viewpoint User Guide*, B035-2206.

## About Deadlocks

Each request in a transaction places its own locks as it runs and continues to hold the locks it acquires until the transaction either commits or rolls back.

The disadvantage of this is that it can result in deadlocks when multiple concurrent accesses against the same database objects occur. The application-level solution to this potential problem is to apply all the explicit database- and table-level locks you need in a single request at the beginning of a transaction.

## Deadlock Detection

Vantage supports deadlock detection and handling at the following levels:

- AMP-local

A LOCAL deadlock is contention between threads within an AMP, localized within one vproc. The AMP-local deadlock detection software checks for deadlocks at a fixed 30-second interval.

- Global

A GLOBAL deadlock is the contention between threads over two or more different AMPs.

Global deadlock detection runs at a user-defined period set by the DBS Control utility using the

DeadLockTimeOut parameter. 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.

Use the Viewpoint Lock Logger portlet to detect blocked transactions and global deadlocks. For information about how to use this utility portlet, see *Teradata® Viewpoint User Guide*, B035-2206.

You can also use the XMLPLAN output from DBQL logging to investigate deadlock situations. For more information, see *Teradata Vantage™ - Database Administration*, B035-1093.

When the system detects a deadlock, it terminates the transaction and then rolls back the most recently initiated transaction of the two.

## Detecting and Handling Deadlock for Client Utility Locks

Because most client utilities use database locking, the global deadlock detector knows when they become trapped in a deadlock. (The global deadlock detector does not detect HUT deadlocks.) To resolve the deadlock, the Lock Manager aborts the deadlocked transaction, rolls back its updates, and returns a retryable 2631 error to the requesting application or utility.

The following table summarizes what happens to a transaction submitted from a client utility or application when a retryable 2631 error occurs, and then provides a very basic solution for the problem.

| IF the client utility or application is ... | AND ...                 | THEN ...                                                                                                                                                                                                                                                                                                                                                          |
|---------------------------------------------|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BTEQ                                        | .SET<br>RETRY is active | even though the entire transaction has been rolled back, BTEQ retries only the failed request and any subsequent requests in the transaction, not the entire transaction.<br>If this occurs, the updates made by requests in the transaction prior to the failed request are lost, as is the contextual information indicating that a transaction is in progress. |
| anything else                               |                         | you must code the application to.<br>1. Check for error code 2631.<br>2. If error code 2631 is detected, then resubmit the entire aborted transaction.                                                                                                                                                                                                            |

For more information about retryable error 2631, see *Teradata Vantage™ - Database Messages*, B035-1096.

## Minimizing Deadlock

### Minimizing Deadlock by Specifying LOCKING Request Modifier

For particular types of transactions, or for very large or urgent applications, you can reduce or prevent the chance of a deadlock by specifying the LOCKING request modifier with your SQL DML requests (for details, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).

You can use the LOCKING request modifier 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 reduce the chance of a deadlock during an update when multiple transactions select and then update the same row.
- Applying a higher or lower severity of lock than that normally applied.

## Using LOCKING Modifier to Enhance Concurrency

To make more efficient use of system resources, you can use the LOCKING request modifier in a transaction to decrease or increase the restrictiveness of locks that would otherwise be placed by the system during the processing of transaction requests.

The following list contains some important facts about the LOCKING request modifier.

- The LOCKING request modifier is not ANSI SQL-2008-compliant.
- The LOCKING request modifier can precede any SQL request or it can be used alone, without modifying an SQL request. The LOCKING clause is typically used as a modifier, in which case it precedes the request. For example, the following request is valid.

```
LOCKING TABLE tablename FOR READ;
```

- More than one LOCKING request modifier can be specified in the same request.
- The CREATE VIEW, REPLACE VIEW, CREATE RECURSIVE VIEW, and REPLACE RECURSIVE VIEW statements allow the LOCKING request modifier to be specified as part of the view definition.
- You can use the LOCKING request modifier to specify the mode of lock to be placed on a database, table, or rowhash before a request is processed. You can specify LOCKING with the NOWAIT option to abort a transaction if a lock cannot be granted immediately.
- The LOCKING request modifier 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 request is executed. Note that locks for rowhash operations are not documented by EXPLAIN reports.
- If you have load-isolated tables, you can use the LOAD COMMITTED locking modifier to read committed rows without being blocked and without blocking the concurrent isolated modifications. For more information about load-isolated tables, see [Load Isolation](#).

The following example uses the LOCKING request modifier to maximize concurrency:

```
LOCKING TABLE table_a FOR READ
LOCKING TABLE table_b FOR READ
SELECT ...
FROM table_a, table_b
WHERE ...;
LOCKING TABLE table_name FOR WRITE
;
UPDATE ...;
```



## Upgrading Locks Dynamically and Deadlock

When the Optimizer decides to use a primary or unique secondary index to process a SELECT request, the system applies a READ lock on the row hash value.

If the same transaction contains a subsequent DML request based on the same index value, the system 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 request followed by an UPDATE request, as follows. This example assumes the user is operating in Teradata session mode.

| User | SQL Text                                                                                                                              |
|------|---------------------------------------------------------------------------------------------------------------------------------------|
| A    | <pre>BEGIN TRANSACTION;   SELECT y   FROM table_a   WHERE pi = 1;   UPDATE table_a   SET y = 0   WHERE pi = 1; END TRANSACTION;</pre> |
| B    | <pre>BEGIN TRANSACTION;   SELECT z   FROM table_a   WHERE pi = 1;   UPDATE table_a   SET z = 0   WHERE pi = 1; END TRANSACTION;</pre> |

In this case, user\_a and user\_b are allowed to access the rows sharing the same row hash value simultaneously for READ during SELECT processing.

When the user\_a UPDATE request requires a WRITE lock on the table\_a row hash value, the upgrade request is queued, waiting for the user\_b READ lock on table\_a to be released.

However, the user\_b READ lock cannot be released because the user\_b UPDATE request also requires a WRITE lock on the row hash value, and that request is queued waiting for the user\_a Read lock to be released.

You can avoid such deadlocks by preceding the transaction with a LOCKING ROW FOR WRITE or LOCKING ROW FOR EXCLUSIVE phrase as appropriate.

---

### Note:

This phrase does not override any locks already held on the target table.

---

LOCKING ROW is appropriate only for single-table select requests that use a primary index or unique secondary index constraint, as shown in the following example:

| User | SQL Text                                                                                                                                                        |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| A    | <pre>BEGIN TRANSACTION;   LOCKING ROW FOR WRITE   SELECT y   FROM table_a   WHERE USI = 1;   UPDATE table_a   SET y = 0   WHERE USI = 1; END TRANSACTION;</pre> |
| B    | <pre>BEGIN TRANSACTION;   LOCKING ROW FOR WRITE   SELECT z   FROM table_a   WHERE USI = 1;   UPDATE table_a   SET z = 0   WHERE USI = 1; END TRANSACTION;</pre> |

In this example, the user\_a request for a rowhash-level WRITE lock is granted, which blocks the user\_b request for a WRITE lock on that row hash value. The user\_b transaction is queued until the user\_a lock is released.

The user\_a lock is held until the entire transaction is complete. Thus, the user\_b LOCKING ROW FOR request is granted only after the user\_a END TRANSACTION request has been processed.

### Using LOCKING Request Modifier with NOWAIT Option

If your request cannot wait in a lock queue, you can specify the LOCKING request modifier with the NOWAIT option.

NOWAIT specifies that the entire transaction, even in ANSI session mode, is to be aborted if the system cannot place the necessary lock on the target object immediately upon receipt of a request.

This situation is treated as a failure. 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.

### Specifying LOCKING Request Modifier in View Definitions

The LOCKING request modifier can be specified in CREATE VIEW, REPLACE VIEW, CREATE RECURSIVE VIEW, and REPLACE RECURSIVE VIEW definitions. For example, the view can downgrade READ locks to ACCESS locks. Thus, the ad hoc user need not worry about specifying a locking request modifier and accidentally impacting transaction processing, and users can modify base table data without impacting any existing report-oriented queries.

## Preventing Deadlocks When Using BTEQ

If you use BTEQ to submit a transaction, the database reports the deadlock abort to BTEQ. BTEQ resubmits only the request that caused the failure (the default behavior), not the complete transaction. Because this can result in partially committed transactions, you must take care when writing a BTEQ script to ensure that the transaction is one request. For example, a statement in BTEQ ends with a semicolon (;) as the last non-blank character in the line. Thus, BTEQ sees the following example as two requests:

```
sel * from x;
sel * from y;
```

However, if you write these same statements in the following way, BTEQ sees them as only one request:

```
sel * from x
; sel * from y;
```

Suppose you use BEGIN TRANSACTION and submit separate requests, as shown by the following:

```
BEGIN TRANSACTION
INSERT x (1, 2) ;
INSERT x (3, 4) ;
INSERT x (5, 6) ;
```

If deadlock occurs on the third insert, the transaction is rolled back and, if retry is enabled for BTEQ, the third insert is resubmitted as an implicit transaction. If the third insert is then followed by END TRANSACTION, a failure occurs, but the third insert was already committed.

## Example of a Transaction Without Deadlock

This example demonstrates how to optimize concurrency.

### Table Definition for the Example

Assume the following table definition:

```
CREATE TABLE table_1 (
 column_1 INTEGER,
 column_2 INTEGER)
PRIMARY INDEX (column_1);
```

### Problem Transactions for the Example

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 Without Deadlock

Assume these steps are taken in the following order.

1. Transaction 1 places a table-level READ lock on table\_1.
2. Transaction 2 also places an table-level READ lock on table \_1.
3. Both transactions access table\_1 at the same time and run concurrently.
4. Transaction 1 builds the fallback concurrently with the SELECT request in the transaction 2, but does not complete until after transaction 2 completes and releases its lock on table\_1. Then transaction 1 can upgrade its lock to EXCLUSIVE to complete the ALTER.

### Example of a Transaction With Deadlock

The following example demonstrates 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;                                                        |

| Transaction Number | Request Number | SQL Text                                                                  |
|--------------------|----------------|---------------------------------------------------------------------------|
|                    | 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:

- 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).
- Request 3 places an READ lock on table\_1.
- Transaction 1 and request 3 can run concurrently when granted access to table\_1.
- Request 3 finishes but does not release the READ lock on table\_1 until the end of the transaction.
- Request 4 attempts to place a WRITE rowhash-level lock on table\_1.  
Its lock request is blocked because of the READ lock placed on table\_1 by transaction 1.
- Transaction 1 needs to upgrade its lock from READ to EXCLUSIVE.  
Its lock request is blocked because of the WRITE rowhash-level lock placed by request 4 in transaction 2.
- At this point there is a deadlock situation: request 4 in transaction 2 is waiting for transaction 1 to release its lock, and transaction 1 is blocked by request 4.

## Resolving the Problem

To avoid this deadlock, change your SQL in either of the following ways:

- Add the modifier LOCKING table\_1 FOR WRITE to transaction 2 as follows.

```
LOCKING table_1 FOR WRITE
BEGIN TRANSACTION;
SELECT *
FROM table_1;
UPDATE table_1
SET column_1 = value_1
```

```
WHERE column_1 = value_2;
END TRANSACTION;
```

- Remove the LOCKING request modifier from transaction 1.

```
CREATE INDEX (column_3, column_4) ON table_1;
```

## Example of Two Serial Transactions

This example shows 2 transactions. The first transaction begins its operations 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:**

The SELECT request does not recognize the index being created by the CREATE INDEX request.

## Resolving the Problem

To eliminate concurrency, change the coding of transaction 1 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 transaction 1 blocks the table-level READ lock request on table\_1 in transaction 2.

## DDL and DCL Requests, Dictionary Access, and Locks

The execution of a DDL or DCL request causes the data dictionary to be updated and appropriate locks to be placed on system tables while that request is processing.

### Optimizing the Locking Granularity for Data Dictionary Access

To improve concurrency, DDL and DCL processing use the finest locking granularity that is practical and delay placing their locks for as long as possible. Depending on the dictionary table, the system sometimes downgrades rowhash READ lock requests made 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 rowhash READs are not blocked, then they use the standard READ locks.

The following dictionary views and tables are affected by this locking downgrade on a blocked READ lock request:

- DBC.AccLogRuleTbl
- DBC.ConstraintNames
- DBC.Indexes
- DBC.TableConstraints
- DBC.TextTbl
- DBC.TriggersV
- DBC.TVFields
- DBC.TVM
- DBC.UDFInfo

The only SQL statements eligible for a dictionary access READ lock-to-ACCESS lock downgrade upon being otherwise blocked are the following:

- SELECT
- HELP COLUMN
- HELP CONSTRAINT
- HELP INDEX
- HELP STATISTICS
- SHOW FUNCTION/HASH INDEX/JOIN INDEX/MACRO/METHOD/PROCEDURE/TABLE/TRIGGER/TYPE/VIEW

These are system-initiated lock downgrades: you cannot specify them using the LOCKING request modifier. For more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## DML Requests and Locks

When it processes DML requests, Vantage 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 that the system applies for DML requests are listed in the following table.

### Note:

When any of the modifications in the table are performed as nonconcurrent isolated load operations on a load-isolated table, the Lock Manager sets an EXCLUSIVE lock rather than what is listed in the table.

| DML Request        | Updated Columns | Selection Criteria | Object Locked | Locking Severity                                                                                                                                                          |
|--------------------|-----------------|--------------------|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SELECT             | Not applicable  | UPI or USI         | Row hash      | READ<br>If a join or hash index includes the specified columns, the Lock Manager sets a READ lock on the entire join or hash index and does not access the base table.    |
|                    |                 |                    | Set of rows   | READ<br>If a join or hash index includes the specified columns, the Lock Manager sets a READ lock on a set of join or hash index rows and does not access the base table. |
|                    |                 | Any other          | Table         | READ<br>When the SELECT is a tactical query, the Lock Manager downgrades the lock request from READ to ACCESS if the operation would otherwise be blocked.                |
| SELECT AND CONSUME | Not applicable  | Any                | Row hash      | WRITE                                                                                                                                                                     |



| DML Request                                                  | Updated Columns      | Selection Criteria | Object Locked | Locking Severity                                                                                                                                                 |
|--------------------------------------------------------------|----------------------|--------------------|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DELETE<br>On tables without a hash or join index             | Not applicable       | UPI or USI         | Row hash      | WRITE                                                                                                                                                            |
|                                                              |                      | NUPI               | Set of rows   |                                                                                                                                                                  |
|                                                              |                      | Any other          | Table         |                                                                                                                                                                  |
| DELETE<br>On tables with a hash or join index                | Not applicable       | UPI or USI         | Row hash      | WRITE<br>Deletes on a table with a hash or join index also require a WRITE lock on the index table plus READ locks on other tables associated with a join index. |
|                                                              |                      | NUPI               | Row hash      |                                                                                                                                                                  |
|                                                              |                      | Any other          | Table         |                                                                                                                                                                  |
| INSERT ...<br>SELECT                                         | Not applicable       | Select table       |               |                                                                                                                                                                  |
|                                                              |                      | UPI or USI         | Row hash      | READ<br>Inserts on a table with a hash or join index also require a WRITE lock on the index table plus READ locks on other tables associated with a join index.  |
|                                                              |                      | NUPI               | Row hash      |                                                                                                                                                                  |
|                                                              |                      | Any other          | Table         |                                                                                                                                                                  |
|                                                              |                      |                    | Insert table  | WRITE                                                                                                                                                            |
| INSERT<br>[VALUES]<br>On tables without a hash or join index | Not applicable       | Not applicable     | Primary row   | WRITE                                                                                                                                                            |
| INSERT<br>[VALUES]<br>On tables with a hash or join index    | Not applicable       | Not applicable     | Primary row   | WRITE<br>Inserts on a table with a hash or join index also require a WRITE lock on the index table plus READ locks on other tables associated with a join index  |
| UPDATE<br>On tables without a hash or join index             | Neither UPI nor USI  | UPI or USI         | Row hash      | WRITE<br>Updates on a table that does not have a join or hash index require WRITE locks on the table.                                                            |
|                                                              | Neither NUPI nor USI | NUPI               | Set of rows   |                                                                                                                                                                  |
|                                                              |                      | Any other          | Table         |                                                                                                                                                                  |
|                                                              | USI                  | USI                | Table         |                                                                                                                                                                  |
| UPDATE<br>On tables with a hash or join index                | Neither UPI nor USI  | UPI or USI         | Row hash      | WRITE<br>Updates on a table with a join or hash index require WRITE locks on the table                                                                           |

| DML Request    | Updated Columns      | Selection Criteria | Object Locked | Locking Severity                                                                                                                                                                                                                                        |
|----------------|----------------------|--------------------|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                | Neither NUPI nor USI | NUPI               | Set of rows   | and the hash or join index and READ locks on other tables associated with it if and only if the modified base table columns are also defined for the hash or join index.                                                                                |
|                |                      | Any other          | Table         |                                                                                                                                                                                                                                                         |
|                | USI                  | USI                | Table         |                                                                                                                                                                                                                                                         |
| MERGE (Update) | Neither UPI nor USI  | UPI or USI         | Row hash      | WRITE<br>MERGE places a matching rowhash lock (a rowhash-level lock based on the primary index value specified by the MATCH condition) when performing the MERGE results in an update.                                                                  |
|                | Neither NUPI nor USI | NUPI               | Set of rows   |                                                                                                                                                                                                                                                         |
|                |                      | Any other          | Table         |                                                                                                                                                                                                                                                         |
|                | USI                  | USI                | Table         |                                                                                                                                                                                                                                                         |
| MERGE (Insert) | Not applicable       | Not applicable     | Primary row   | WRITE<br>MERGE inserts on a table with a join or hash index require WRITE locks on the hash or join index and READ locks on other tables associated with it if and only if the modified base table columns are also defined for the hash or join index. |

## Locking Issues With Consume Mode SELECT Queries on a Queue Table

### About Special Locking Issues Raised by Consume Mode SELECT Requests

Several locking issues apply to consume mode SELECT operations. This topic introduces the most important locking issues with SELECT requests made against queue tables.

The following bulleted list documents the more important locking issues for consume mode SELECT requests:

- The default lock assignment for a consume mode SELECT operation against a queue table has WRITE severity at the rowhash level.  
You cannot lower the severity of this lock assignment.
- The default lock assignment for a consume mode SELECT operation against the target table of an INSERT ... SELECT AND CONSUME operation has WRITE severity at the table level.

Although you can specify a LOCKING request modifier, it has no effect on the behavior of the SELECT AND CONSUME operation: the system still retrieves rows in the same order and the query enters a delay state when the table is empty.

- The system does not grant a rowhash-level WRITE lock on a queue table for a consume mode SELECT operation unless there is at least one row in the table to be consumed. This is unlike all other SQL requests, where the system grants locks at the time the it receives the request.

As soon as a row is inserted into the subject queue table, the following things happen:

- The system grants the WRITE lock on the first row hash in the queue.
  - Transaction processing resumes.
  - Rows are consumed until the queue is empty.
- You can include only one consume mode SELECT request in a multirequest transaction in Teradata session mode unless you place all such consume mode SELECT requests between explicit BEGIN TRANSACTION and END TRANSACTION requests.

For example, the following Teradata session mode 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;
```

The following consume mode SELECT requests are also valid in Teradata session mode, but each is an individual implicit transaction (see [Transactions, Requests, and Statements](#)).

```
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 requests in ANSI session mode except to remember that you must terminate ANSI session mode transactions by submitting either a COMMIT request or a ROLLBACK request.

SELECT AND CONSUME requests are the only SELECT requests where it matters whether you terminate the transaction with a COMMIT request or a ROLLBACK request.

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;
```

## NOTICE

Every request you submit in ANSI session mode is treated as part of the same transaction until you submit an explicit COMMIT or ROLLBACK request.

If you submit a ROLLBACK request, 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 requests 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 requests.
  - Coding any SELECT AND CONSUME requests within explicit transactions.
  - Coding large numbers of SELECT AND CONSUME requests within a transaction, especially if there are also DELETE and UPDATE operations made on the same queue table as SELECT AND CONSUME requests.

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 request as the only request in a transaction.

- Vantage aborts any transaction in which the limit on the number of SELECT AND CONSUME requests, 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.

- Vantage uses the Teradata dynamic workload management software to manage all deferred requests (transactions in a delayed state) against a queue table. The Teradata dynamic workload software client application software does not need to be enabled to be used by the system to manage delayed consume mode requests.

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 dynamic workload management software to manage not just delayed consume mode queries, but all queries, optimally.

- The system returns a failure response 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.

## Cursor Locking Modes

### Rules for Cursor Locking

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 request performed when its associated cursor is opened causes Vantage to use either a table-level or rowhash-level lock by default, depending on the constraint clause of the SELECT request.

You can explicitly change this locking level by using the LOCKING request modifier. The LOCKING request modifier is supported for updatable cursors, and the CHECKSUM locking severity is designed especially for use with LOCKING.

## Interactions Between Cursors and Locks

When it opens a cursor, Vantage generates a response spool. This spool identifies each data row that is a source for the response data in the spool row.

Vantage uses this identifier 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 Vantage ...                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ACCESS                         | <p>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.</p> <p>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.</p>                                                                                                                                                                                                                                                                                                        |
| CHECKSUM                       | <p>inserts a checksum into each row of the spool.</p> <p>This provides a mechanism for ensuring that the rows in the spool have not been modified by another user or session at the time an update is being made through the cursor.</p> <p>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.</p> <p>The CHECKSUM option uses ACCESS severity locks. CHECKSUM locking differs in that it also provides the checksums of the spool rows.</p> |

## Simple Example of Cursor Locking

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);
```

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for more complex examples of locking and cursors.

## Locking Issues With Tactical Queries

### Special Locking Issues Raised by Tactical Queries

Tactical queries deliver better response times and throughput using rowhash-level locks whenever possible. Rowhash-level 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

Rowhash-level 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 *Teradata Vantage™ - Database Design*, B035-1094 for information about how to design your databases in ways that facilitate mixed tactical and decision support query workloads.

### Group AMP Locking Considerations for Tactical Queries

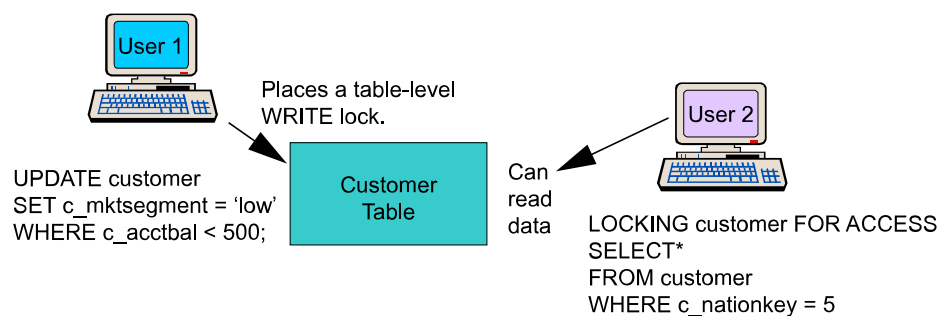
Group AMP operations use a series of rowhash-level locks, one for each of the rows touched by the query. See *Teradata Vantage™ - Database Design*, B035-1094 for more information about group AMP operations.

One of the major benefits of group AMP processing is that it significantly reduces the need to apply table-level locks. Table-level locking, because it is at a higher level than rowhash-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 requests and reading by another is occurring on 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.

### Rowhash-Level or Table-Level ACCESS Locks for Tactical Queries

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.

## Escalating Lock Enhancements Automatically

Even if you explicitly specify a rowhash-level ACCESS lock, the Optimizer automatically converts it to a table-level lock if the query plan requires an all-AMPs operation and there is only one table referenced in the query. Always check the EXPLAIN report to verify that row hash or, when appropriate, table-level ACCESS locks are issued for tactical queries.

The following EXPLAIN report for an all-AMPs query shows that the Optimizer applies a table-level ACCESS lock even though the LOCKING request modifier explicitly requests a rowhash-level ACCESS lock.

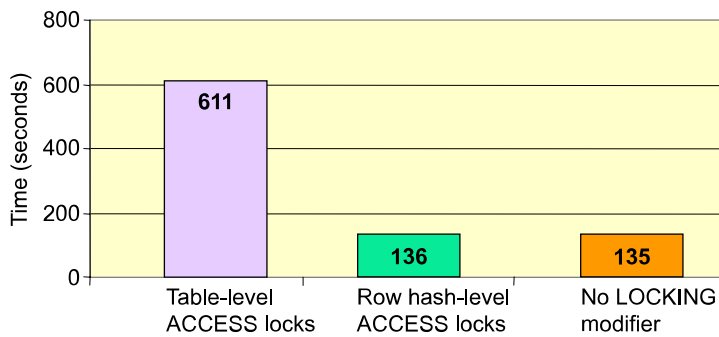
```

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 request, 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 request.

The following graph illustrates the cost of using table-level ACCESS locks compared to rowhash-level ACCESS locks when the request itself only performs single-AMP operations.



The elapsed time represents the total time to perform 100 000 single-row SELECT requests using 20 sessions. When table-level ACCESS locks were specified for the request, total execution time for this workload increased by a factor of 5. The response times for the variables labeled rowhash-level ACCESS locks and NO LOCKING modifier are almost identical because when NO LOCKING modifier was specified, the Optimizer applied a rowhash-level READ lock in the background. This lock has the same overhead as the row-level ACCESS lock.

Some request tools make it difficult to specify an ACCESS lock modifier. In spite of this, you can enforce explicit ACCESS locking by querying views and placing the appropriate LOCKING request modifier in their view definitions.

### Rowhash-Level ACCESS Locks, Join Indexes, and Tactical Queries

The Optimizer propagates rowhash-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 rowhash-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 rowhash-level ACCESS lock to the join index row hash, not to the base table row hash.

## Rowhash-Level ACCESS Locks, Group AMP Steps, and Tactical Queries

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 rowhash-level locks on each row hash in the AMP group. The same performance-enhancing lock-level substitution also occurs when you explicitly request rowhash-level locking by specifying a LOCKING ROW FOR ACCESS modifier with your SQL request, as demonstrated by the EXPLAIN report generated for the following SELECT request. Notice the rowhash-level locks with ACCESS severity being applied in step 2 to the rows of both tables that are accessed by the request.

```
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, Joins, and Tactical Queries

When a query for which a rowhash-level ACCESS lock has been requested also includes a join operation, the Optimizer applies the rowhash-level ACCESS lock request to both tables if both are eligible.

The Optimizer plan generated for the following request 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
merge join, with a join condition of ("CAB.lineitem.L_ORDERKEY =
CAB.ordertbl.O_ORDERKEY"). The input tables CAB.ordertbl and
CAB.lineitem will not be cached in memory, but CAB.ordertbl
is eligible for synchronized scanning. The result goes into
Spool 1(one-amp), which is built locally on that AMP. The
size of Spool 1 is estimated with low confidence to be 35
rows. The estimated time for this step is 0.03 seconds.

```

## Rowhash-Level ACCESS Locks in Tactical Queries Are Compatible With Aggregates

If aggregation can be performed as a single-AMP operation, the Optimizer honors explicit rowhash-level ACCESS lock requests. An example might be the following request, 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 rowhash-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 requests. If this is a problem at your site, you can instead enforce ACCESS locking by placing the LOCKING request modifier in views through which the queries access the base tables.

### Locking for Tactical Query Updates

A simple update request that specifies a primary index value for the table cues the Optimizer to apply a rowhash-level WRITE lock to process the update. Rowhash-level WRITE or rowhash-level READ locks are not reported in the EXPLAIN text. This request updates the unindexed orders table column o\_orderpriority by accessing a single row using the UPI defined on o\_orderkey. Only a single AMP and a rowhash-level lock are used to process this request.

```
EXPLAIN
UPDATE orders
SET o_orderpriority = 5
WHERE o_orderkey = 39256
```

#### Explanation

- 
- 1) First, we do a **single-AMP UPDATE** from CAB.orders by way of the unique primary index "CAB.orders.O\_ORDERKEY = 39256" with no residual conditions.  
-> No rows are returned to the user as the result of statement 1.

### Locking for Complex Tactical Query Updates

The following query updates the same unindexed 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 high-performing single-AMP merge update using rowhash-level locking, as you can see in Step 1 of the EXPLAIN report for the following UPDATE request:

```
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.

# Query Capture Facility

This section briefly describes the query capture database (QCD) that supports the various Database Query Analysis tools such as the Query Capture Facility.

It also provides the definitions for the QCD tables. The physical implementation of QCD is presented here so you can create and perform your own queries and applications against the information it stores.

For information about using the Query Capture Facility, see *Teradata Vantage™ - Database Administration*, B035-1093.

For information about the various query analysis tools and SQL statements that are relevant to the Query Capture Facility, see the following manuals:

- *Teradata® Viewpoint User Guide*, B035-2206
- *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
  - COLLECT STATISTICS (QCD Form)
  - DROP STATISTICS (QCD Form)
  - DUMP EXPLAIN
  - INITIATE INDEX ANALYSIS
  - INITIATE PARTITION ANALYSIS
  - INSERT EXPLAIN
  - RESTART INDEX ANALYSIS
- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
  - BEGIN QUERY CAPTURE

## Quick Functional Overview of the Query Capture Facility

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 *Teradata Vantage™ - Database Administration*, B035-1093.

### 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

For more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## QCD Physical Model

Note the following facts about the physical implementation of QCD databases.

- 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.

See *Teradata Vantage™ - Database Design*, B035-1094 for information about physical capacity planning for your QCDs.

## Applications of QCF and QCD

Vantage supports the following applications of QCF and QCD:

- 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.
- 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.”

## QCD Table Definitions

This section provides the definitions for the QCD tables.

## AnalysisLog

### Function of AnalysisLog

AnalysisLog maintains the checkpoint information that is recorded when you enable the CHECKPOINT option for an INITIATE INDEX ANALYSIS request.

### AnalysisLog Table Definition

The following CREATE TABLE request defines the AnalysisLog table:

```
CREATE TABLE AnalysisLog (
 WorkLoadID INTEGER NOT NULL,
 RecommendationID INTEGER NOT NULL,
```

```

IndexNameTag VARCHAR(128) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
SeqNumber SMALLINT NOT NULL,
Cflag CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC NOT NULL,
UserName VARCHAR(128) 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 for AnalysisLog

The following table defines the AnalysisLog table attributes:

| Attribute        | Definition                                                                                                                                                                                                                                |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WorkloadID       | <ul style="list-style-type: none"> <li>Uniquely identifies the workload.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                  |
| RecommendationID | Uniquely identifies the index recommendation generated for the specific index analysis.                                                                                                                                                   |
| IndexNameTag     | <ul style="list-style-type: none"> <li>User-specified name for the index analysis.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                        |
| SeqNumber        | A sequence number to identify individual rows that belong to a multirow AnalysisStatus.<br>The sequence begins at 1.                                                                                                                      |
| Cflag            | Used with multirow AnalysisStatus. <ul style="list-style-type: none"> <li>If the row is the last row in the sequence, then Cflag is set to F.</li> <li>If the row is not the last row in the sequence, then Cflag is set to T.</li> </ul> |
| UserName         | Name of the user performing the index analysis.                                                                                                                                                                                           |
| AnalysisStatus   | Identifies the status of the index analysis.                                                                                                                                                                                              |
| StartTime        | Timestamp for the beginning of the index analysis.                                                                                                                                                                                        |
| UpdateTime       | Timestamp for the moment the AnalysisStatus column is most recently updated.                                                                                                                                                              |

## AnalysisStmts

Supports combined use of the CHECKPOINT and TIMELIMIT options for the INITIATE INDEX ANALYSIS and RESTART INDEX ANALYSIS statements, particularly in the context of submitting an INITIATE

INDEX ANALYSIS request that is halted for some reason, followed by one or more RESTART INDEX ANALYSIS requests.

The table provides the following information to support generating index recommendations in this scenario:

- Identifies a group of logically related INITIATE INDEX ANALYSIS and RESTART INDEX ANALYSIS requests performed as a related sequence.
- Provides the sequence number for a given RESTART INDEX ANALYSIS request within the group.
- Provides the SQL text for the INITIATE INDEX ANALYSIS or RESTART INDEX ANALYSIS request used to produce a given set of index recommendations.

### AnalysisStmts Table Definition

The following CREATE TABLE request defines the AnalysisStmts table:

```
CREATE SET TABLE QCD.AnalysisStmts, NO FALLBACK, NO BEFORE JOURNAL,
 NO AFTER JOURNAL,
 CHECKSUM=DEFAULT
 DEFAULT MERGEBLOCKRATIO(
WorkLoadID INTEGER NOT NULL,
IndexNameTag VARCHAR(128) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
RecommendationID INTEGER NOT NULL,
IASeqNumber INTEGER NOT NULL, TimeOfAnalysis TIMESTAMP(6) NOT NULL,
IAStmtText VARCHAR(20000) CHARACTER SET UNICODE
 NOT CASESPECIFIC)
PRIMARY INDEX (WorkLoadID);
```

### Attribute Definitions for AnalysisStmts

The following table defines the AnalysisStmts table attributes.

| Attribute        | Definition                                                                                                           |
|------------------|----------------------------------------------------------------------------------------------------------------------|
| WorkLoadID       | <ul style="list-style-type: none"> <li>• Uniquely identifies the workload.</li> <li>• NUPI for the table.</li> </ul> |
| IndexNameTag     | User-specified name for the index analysis.                                                                          |
| RecommendationID | Uniquely identifies a set of index recommendations.                                                                  |
| IASeqNumber      | System-assigned sequence number for this index analysis.                                                             |
| TimeOfAnalysis   | The timestamp of this index analysis                                                                                 |
| IAStmtText       | The SQL text for the request being analyzed.                                                                         |

## Related Information

See the following topics for further information related to the AnalysisStmts table.

- [IndexRecommendations](#)
- [SeqNumber](#)
- INITIATE INDEX ANALYSIS and RESTART INDEX ANALYSIS in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146

## DataDemographics

Contains table demographic information collected by COLLECT DEMOGRAPHICS.

### DataDemographics Table Definition

The following CREATE TABLE request defines the DataDemographics table:

```
CREATE TABLE DataDemographics (
 MachineName VARCHAR(30) CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC NOT NULL,
 TableName VARCHAR(128) CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC NOT NULL,
 DatabaseName VARCHAR(128) 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) CHARACTER SET LATIN NOT CASESPECIFIC,
 RowCount DECIMAL(18,0) NOT NULL,
 AvgRowSize INTEGER NOT NULL,
 QueryID INTEGER,
 IndexName VARCHAR(128) CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC,
 DemographicsID INTEGER)
PRIMARY INDEX (MachineName, TableName, DatabaseName);
```

### Attribute Definitions for DataDemographics

The following table defines the DataDemographics table attributes:

| Attribute   | Definition                                                                                                                                    |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| MachineName | <ul style="list-style-type: none"> <li>• The name of the system to which TableName belongs.</li> <li>• Partial NUPI for the table.</li> </ul> |

| Attribute      | Definition                                                                                                                                                                                                                                                                                                               |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TableName      | <ul style="list-style-type: none"> <li>The name of the table on which demographic data has been collected.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                                                               |
| DatabaseName   | <ul style="list-style-type: none"> <li>The name of the containing database for TableName.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                                                                                |
| DBSize         | The size in KB of data blocks in TableName.                                                                                                                                                                                                                                                                              |
| CollectedTime  | The timestamp value when the data demographics were collected.                                                                                                                                                                                                                                                           |
| AMPNumber      | The AMP Vproc number to which the row information pertains.                                                                                                                                                                                                                                                              |
| ClusterNumber  | The number of the cluster to which AMPNumber belongs.                                                                                                                                                                                                                                                                    |
| SubTableID     | The unique identifier for the subtable in which the data the row describes is stored.                                                                                                                                                                                                                                    |
| SubTableType   | The subtable data in text format.                                                                                                                                                                                                                                                                                        |
| RowCount       | The cardinality of the subtable.                                                                                                                                                                                                                                                                                         |
| AvgRowSize     | The average size of a row in the subtable.                                                                                                                                                                                                                                                                               |
| QueryID        | <p>The value for QueryID depends on how the demographics are captured.</p> <ul style="list-style-type: none"> <li>If demographics are captured by a COLLECT DEMOGRAPHICS request, QueryID is null.</li> <li>If demographics are captured by an INSERT EXPLAIN request, QueryID is the unique ID of the query.</li> </ul> |
| IndexName      | A comma-separated list of the names of the primary and secondary indexes for the table. If an index has no name, then it is represented in this column by a comma-separated list of the names of the columns that compose it.                                                                                            |
| DemographicsID | <p>Set to 1 if the demographics are captured by a COLLECT DEMOGRAPHICS or INSERT EXPLAIN AND DEMOGRAPHICS statement.</p> <p>1 indicates that the capture is on the system on which the row is inserted.</p> <p>DemographicsID has different values if the demographics are imported rather than captured directly.</p>   |

## 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.TableSizeVX.

## Field

### Function of Field

Captures all the columns used or referenced in a captured query plan.

## Field Table Definition

The following CREATE TABLE request defines the Field table:

```
CREATE TABLE Field (
 FieldID INTEGER NOT NULL,
 RelationKey INTEGER NOT NULL,
 Name VARCHAR(128) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 FldAlias VARCHAR(128) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 QueryID INTEGER NOT NULL,
 ValueAccessFrequency INTEGER,
 JoinAccessFrequency INTEGER,
 RangeAccessFrequency INTEGER,
 ChangeRate INTEGER,
 DataLength INTEGER,
 StatsKind CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC,
 NumNulls FLOAT,
 NumIntervals INTEGER,
 MinValue VARCHAR(512) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 ModeValue VARCHAR(512) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 ModeFreq FLOAT,
 TotalValues FLOAT,
 TotalRows FLOAT)
PRIMARY INDEX(RelationKey)
UNIQUE INDEX USK_FieldID_RelationKey (FieldID, RelationKey);
```

## Attribute Definitions for Field

The following table defines the Field table attributes:

| Attribute   | Definition                                                                                                                                                                                                            |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FieldID     | <ul style="list-style-type: none"> <li>• Unique column identifier within a specific table.</li> <li>• Partial USI for the table.</li> </ul>                                                                           |
| RelationKey | <ul style="list-style-type: none"> <li>• Unique identifier for the relation in which the column is defined within a certain database.</li> <li>• NUPI for the table.</li> <li>• Partial USI for the table.</li> </ul> |
| Name        | The name of the captured column.                                                                                                                                                                                      |

| Attribute            | Definition                                                                                                                                                                                                                                                  |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FldAlias             | <ul style="list-style-type: none"> <li>Alias name for the column.</li> <li>Null if none exists.</li> </ul>                                                                                                                                                  |
| QueryID              | The unique ID for the query.                                                                                                                                                                                                                                |
| ValueAccessFrequency | The number of times the column is used with an equality condition.                                                                                                                                                                                          |
| JoinAccessFrequency  | The number of times the column is used in a join condition.                                                                                                                                                                                                 |
| RangeAccessFrequency | The number of times the column is used in a range condition.                                                                                                                                                                                                |
| ChangeRate           | <p>The change rating value for the column.</p> <ul style="list-style-type: none"> <li>If the columns is modified, ChangeRate is set to 1.</li> <li>If the column is not modified, ChangeRate is set to 0.</li> </ul>                                        |
| DataLength           | The maximum length of the column.                                                                                                                                                                                                                           |
| StatsKind            | <p>Defines whether the statistics are collected from the dictionary or from a QCD. The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available for the column or index.</p> |
| NumNulls             | <p>Number of nulls for the column or index.</p> <p>The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available.</p>                                                         |
| NumIntervals         | <p>Number of intervals for the column or index.</p> <p>The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available.</p>                                                     |
| MinValue             | <p>Minimum value for the interval.</p> <p>The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available.</p>                                                                  |
| ModeValue            | <p>Modal value for the interval.</p> <p>The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available.</p>                                                                    |
| ModeFreq             | <p>Number of occurrences of the modal value in the interval.</p> <p>The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available.</p>                                        |
| TotalValues          | <p>Number of unique values for the column or index in the table.</p> <p>The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available.</p>                                    |
| TotalRows            | <p>Cardinality of the table.</p> <p>The data for this column is retrieved from interval 0 of the available statistics. The column is set to null if no statistics are available.</p>                                                                        |

## Index\_Field

### Function of Index\_Field

Captures all the index columns used in the query plan.

### Index\_Field Table Definition

The following CREATE TABLE request 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 for Index\_Field

The following table defines the Index\_Field table attributes:

| Attribute   | Definition                                                                                                                                                                                                           |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RelationKey | <ul style="list-style-type: none"> <li>• Unique identifier for the relation in which the captured index columns are defined.</li> <li>• Partial NUPI for the table.</li> <li>• Partial USI for the table.</li> </ul> |
| IndexNum    | <ul style="list-style-type: none"> <li>• Unique identifier for the captured index.</li> <li>• Partial NUPI for the table.</li> <li>• Partial USI for the table.</li> </ul>                                           |
| FieldID     | <ul style="list-style-type: none"> <li>• Unique column identifier within a specific table.</li> <li>• Partial USI for the table.</li> </ul>                                                                          |

## IndexColumns

### Function of IndexColumns

Captures the columns that form the index identified by IndexID during index analysis using INITIATE INDEX ANALYSIS.

### IndexColumns Table Definition

The following CREATE TABLE request 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 VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL)
PRIMARY INDEX (RecommendationID, TableID, IndexID);
```

### Attribute Definitions for IndexColumns

The following table defines the IndexColumns table attributes:

| Attribute        | Definition                                                                                                                                                                                                           |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WorkloadID       | Uniquely identifies the workload analyzed to create this secondary index recommendation.                                                                                                                             |
| RecommendationID | <ul style="list-style-type: none"> <li>Uniquely identifies the recommendation ID in the IndexRecommendations table.</li> <li>Partial NUPI for the table.</li> </ul>                                                  |
| TableID          | <ul style="list-style-type: none"> <li>Uniquely identifies the table ID in the IndexRecommendations table.</li> <li>Partial NUPI for the table.</li> </ul>                                                           |
| IndexID          | <ul style="list-style-type: none"> <li>Uniquely identifies the index in the IndexRecommendations table.</li> <li>Partial NUPI for the table.</li> </ul>                                                              |
| ColumnName       | <p>The name of the column in the index.</p> <p>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.</p> |

## IndexMaintenance

### Function of IndexMaintenance

IndexMaintenance stores the estimated costs incurred by the INSERT, UPDATE, MERGE, or DELETE requests in maintaining the recommended indexes stored in IndexRecommendations.

IndexMaintenance contains one row for each SQL request-index combination where maintenance costs are required for the index. You can query this table directly to retrieve information about the estimated cost of maintaining the indexes for a given workload.

### IndexMaintenance Table Definition

The following CREATE TABLE statement defines the IndexMaintenance table:

```

CREATE TABLE IndexMaintenance (
 WorkloadID INTEGER NOT NULL,
 RecommendationID INTEGER NOT NULL,
 IndexNameTag VARCHAR128 CHARACTER SET UNICODE NOT CASESPECIFIC
 NOT NULL,
 SQLStatementID INTEGER NOT NULL,
 SecondaryIndexID INTEGER DEFAULT NULL,
 BaseTableID BYTE(6) NOT NULL,
 BaseTableName VARCHAR128 CHARACTER SET UNICODE NOT CASESPECIFIC
 NOT NULL,
 DatabaseName VARCHAR128 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 for IndexMaintenance

The following table defines the IndexMaintenance table attributes:

| Attribute        | Description                                                                                                                                                                                                                   |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WorkloadID       | <ul style="list-style-type: none"> <li>Uniquely identifies the workload analyzed to create the attribute data.</li> <li>NUPI for the table.</li> </ul>                                                                        |
| 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.                                                                                                                                                                               |
| BaseTableName    | The name for the base table referenced by BaseTableID.                                                                                                                                                                        |
| JINumber         | <p>The sequentially assigned number for the recommended join index in the IndexRecommendations table.</p> <p>This column is applicable only if the value for the IndexType column corresponds to a valid join index type.</p> |
| SQLStatementID   | 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.                                                                                 |
| SecondaryIndexID | <p>The IndexID of the recommended secondary index in the IndexRecommendations table.</p> <p>This column is applicable only if the value for the IndexType column corresponds to a valid secondary index type.</p>             |
| IndexType        | <p>A number that identifies the type of index recommended.</p> <p>Each unique index type is associated with its own IndexTypeText.</p>                                                                                        |

| Attribute     | Description                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|               | <p>The valid IndexType codes and their corresponding IndexTypeText strings are as follows:</p> <ul style="list-style-type: none"> <li>• If 1, IndexTypeText is USI.</li> <li>• If 2, IndexTypeText is VOSI.</li> <li>• If 3, IndexTypeText is HOSI.</li> <li>• If 4, IndexTypeText is NUSI.</li> <li>• If 5, IndexTypeText is JI.</li> <li>• If 6, IndexTypeText is JIAGG.</li> </ul>                                                                    |
| IndexTypeText | <p>The textual representation of IndexType.</p> <p>The valid IndexTypeText strings and their meanings are as follows:</p> <ul style="list-style-type: none"> <li>• HOSI is a hash-ordered secondary index.</li> <li>• JI is a simple join index.</li> <li>• JIAGG is an aggregate join index.</li> <li>• NUSI is a nonunique secondary index.</li> <li>• USI is a unique secondary index.</li> <li>• VOSI is a value-ordered secondary index.</li> </ul> |
| MaintCosts    | The estimated cost in milliseconds to update the recommended index structures.                                                                                                                                                                                                                                                                                                                                                                           |

## IndexRecommendations

### Function of IndexRecommendations

Contains information about the index recommendations generated by INITIATE INDEX ANALYSIS.

IndexRecommendations records the options specified during index analysis for later retrieval.

### IndexRecommendations Table Definition

The following CREATE TABLE request defines the IndexRecommendations table:

```
CREATE TABLE IndexRecommendations(
 WorkLoadID INTEGER NOT NULL,
 UserName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 TimeOfAnalysis TIMESTAMP(0) NOT NULL,
 RecommendationID INTEGER NOT NULL,
 QueryID INTEGER NOT NULL,
 IndexID INTEGER,
 IndexNameTag VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 TableName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
```

```

DatabaseName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
TableID BYTE(6) NOT NULL,
IndexType INTEGER,
IndexTypeText VARCHAR(30) CHARACTER SET LATIN
 NOT CASESPECIFIC,
StatisticsInfo VARBYTE(16383),
OriginalCost FLOAT,
NewCost FLOAT,
SpaceEstimate FLOAT,
TimeEstimate FLOAT,
DropFlag CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC,
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) CHARACTER SET LATIN
 NOT CASESPECIFIC,
IndexesPerTable SMALLINT DEFAULT NULL,
SearchSpaceSize SMALLINT,
ChangeRateThreshold BYTEINT,
ColumnPerIndex SMALLINT,
ColumnsPerJoinIndex SMALLINT DEFAULT NULL,
IndexMaintMode BYTEINT DEFAULT NULL,
JINumber INTEGER DEFAULT NULL,
JITableName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC DEFAULT NULL,
TimeLimitExceeded CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC)
TimeLimit INTEGER
PRIMARY INDEX (WorkloadID);

```

### Attribute Definitions for IndexRecommendations

The following table defines the IndexRecommendations table attributes:

| Attribute      | Definition                                                                                                       |
|----------------|------------------------------------------------------------------------------------------------------------------|
| WorkloadID     | <ul style="list-style-type: none"> <li>Uniquely identifies the workload.</li> <li>NUPI for the table.</li> </ul> |
| UserName       | Name of the user performing the index analysis.                                                                  |
| TimeOfAnalysis | The timestamp when the index recommendations were analyzed.                                                      |

| Attribute        | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                  | You can compare this column 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 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 <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 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. <ul style="list-style-type: none"> <li>• 1 indicates the IndexTypeText is a Unique Secondary Index (USI).</li> <li>• 2 indicates the IndexTypeText is a Value-Ordered Secondary Index (VOSI).</li> <li>• 3 indicates the IndexTypeText is a Hash-Ordered Secondary Index (HOSI).</li> <li>• 4 indicates the IndexTypeText is a Nonunique Secondary Index (NUSI).</li> <li>• 5 indicates the IndexTypeText is a Simple Join Index (JI).</li> <li>• 6 indicates the IndexTypeText is an Aggregate Join Index (JIAGG).</li> </ul> |
| IndexTypeText    | The textual representation of IndexType.<br>The valid IndexTypeText strings and their meanings are as follows: <ul style="list-style-type: none"> <li>• HOSI is a Hash-Ordered Secondary Index.</li> <li>• JI is a Simple Join Index.</li> <li>• JIAGG is an Aggregate Join Index.</li> <li>• NUSI is a Nonunique Secondary Index.</li> <li>• USI is a Unique Secondary Index.</li> <li>• VOSI is a Value-Ordered Secondary Index.</li> </ul>                                                                                                                                                                                                              |
| StatisticsInfo   | The statistics, if any, used to make the index recommendations.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| OriginalCost     | The estimated cost of the query in milliseconds before implementing the recommended indexes.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| NewCost          | The estimated cost of the query in milliseconds after implementing the recommended indexes.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| SpaceEstimate    | The estimated space in bytes the recommended index occupies when created.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| TimeEstimate     | The estimated time in milliseconds required to implement the index recommendation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

| Attribute           | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DropFlag            | Identifies whether the specified index is to be added or dropped. <ul style="list-style-type: none"> <li>• N means the index is not recommended for dropping.</li> <li>• Y means the index is recommended for dropping.</li> </ul>                                                                                                                                                                                                            |
| IndexDDL            | The DDL text of the CREATE INDEX, DROP INDEX, CREATE JOIN INDEX, or DROP JOIN INDEX request for the index.                                                                                                                                                                                                                                                                                                                                    |
| StatsDDL            | The DDL text of the COLLECT STATISTICS (QCD form) requests used for the analysis of the index.                                                                                                                                                                                                                                                                                                                                                |
| Remarks             | Provides details on the analysis involved in making the index recommendation.                                                                                                                                                                                                                                                                                                                                                                 |
| AnalysisData        | Reserved for future use.                                                                                                                                                                                                                                                                                                                                                                                                                      |
| IndexesPerTable     | The limit on the number of indexes on a given table as specified by the INITIATE INDEX ANALYSIS statement used to start this analysis.                                                                                                                                                                                                                                                                                                        |
| 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 INITIATE INDEX ANALYSIS SET <i>boundary_option</i> specification.                                                                                                                                                                                                                                             |
| IndexMaintMode      | The integer value used during analysis to control how estimated index maintenance costs are used.<br>The value is specified using the INITIATE INDEX ANALYSIS SET <i>boundary_option</i> specification.                                                                                                                                                                                                                                       |
| JINumber            | An integer value sequence number that identifies the recommended Join Index table for a given index analysis.<br>The column 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: <i>JI_RecommendationID_BaseTableName_JINumber</i> , where <i>JI</i> is a literal string and the values for <i>RecommendationID</i> , <i>BaseTableName</i> , and <i>JINumber</i> are those used in the eponymously named columns of the row, converted to character format where necessary. |
| TimeLimitExceeded   | Indicates whether or not the recommendation was generated by an INITIATE INDEX ANALYSIS request whose specified time limit expired. <ul style="list-style-type: none"> <li>• If F, the partition analysis was not interrupted because the specified INITIATE INDEX ANALYSIS time limit was exceeded.</li> </ul>                                                                                                                               |

| Attribute | Definition                                                                                                                                                                                                                                                              |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|           | <p>This is the default.</p> <ul style="list-style-type: none"> <li>If T, the partition analysis was interrupted because the specified INITIATE INDEX ANALYSIS time limit was exceeded.</li> </ul> <p>This means the final recommendations might have been affected.</p> |
| TimeLimit | A user-specified time limit, in whole number of minutes, for the duration of the analysis.                                                                                                                                                                              |

## IndexTable

### Function of IndexTable

Describes all indexes on the tables specified by the query.

### IndexTable Table Definition

The following CREATE TABLE request defines IndexTable:

```
CREATE TABLE IndexTable (
 IndexNum INTEGER NOT NULL,
 RelationKey INTEGER NOT NULL,
 OrderBy CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC
 NOT NULL,
 AccessInfo CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC
 NOT NULL,
 Field1Only CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC
 NOT NULL,
 RangeConstraint CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC
 NOT NULL,
 IndexFlag CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC,
 IndexName VARCHAR128 CHARACTER SET UNICODE NOT CASESPECIFIC,
 IndexType CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC,
 UniqueFlag CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC,
 IndexKind CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC,
 NumNulls FLOAT,
 NumIntervals INTEGER,
 MinValue VARCHAR(512) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 ModeValue VARCHAR(512) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 ModeFreq FLOAT,
 TotalValues FLOAT,
 TotalRows FLOAT)
```

```
PRIMARY INDEX (RelationKey)
UNIQUE INDEX USK_IdxNum_RelationKey (IndexNum, RelationKey);
```

### Attribute Definitions for IndexTable

The following table defines the IndexTable table attributes:

| Attribute       | Description                                                                                                                                                                                                                                                                                                                                                                                                               |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| IndexNum        | <ul style="list-style-type: none"> <li>• Unique identifier for the captured index.</li> <li>• Partial USI for the table.</li> </ul>                                                                                                                                                                                                                                                                                       |
| RelationKey     | <ul style="list-style-type: none"> <li>• Unique identifier for the relation in which the captured index is defined.</li> <li>• NUPI for the table.</li> <li>• Partial USI for the table.</li> </ul>                                                                                                                                                                                                                       |
| OrderBy         | <p>Defines whether the index has an associated ORDER BY clause.</p> <ul style="list-style-type: none"> <li>• If F, the index has no ORDER BY clause.</li> <li>• If T, the index has an ORDER BY clause.</li> </ul>                                                                                                                                                                                                        |
| AccessInfo      | <p>Specifies if the index is a covering index, bit map, or neither.</p> <ul style="list-style-type: none"> <li>• B is access using a bit map.</li> <li>• C is access using a covering index.</li> <li>• N is access neither by a bit map nor by a covering index.</li> <li>• P is access using a primary index.</li> </ul>                                                                                                |
| Field1Only      | <p>Defines whether the index is a join index and Field1 is the only part needed.</p> <ul style="list-style-type: none"> <li>• If F, this is not a join index requiring Field1 only. This generally means that either the index is not a join index or the index is a compressed join index.</li> <li>• If T, this is a join index requiring Field1 only.<br/>This means that the join index is not compressed.</li> </ul> |
| RangeConstraint | <p>Flag for value-ordered indexes that have a range constraint used by the query plan.</p> <ul style="list-style-type: none"> <li>• If F, 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.</li> <li>• If T, there is a range constraint on the value-ordered index used in the plan.</li> </ul>                                                          |
| IndexFlag       | <p>Flag indicating whether the index was used in the query plan.</p> <ul style="list-style-type: none"> <li>• If F, the index was not used in the query plan.</li> <li>• If T, the index was used in the query plan.</li> </ul> <p>IndexFlag is also set to T if a subset of the row partitions of a row-partitioned primary index is accessed because of row partition elimination.</p>                                  |
| IndexName       | <ul style="list-style-type: none"> <li>• The name of the index if it has one.</li> <li>• Null if this is not a named index.</li> </ul>                                                                                                                                                                                                                                                                                    |
| IndexType       | <p>A code indicating the type of the index.</p> <ul style="list-style-type: none"> <li>• H is a Hash-ordered secondary covering index.</li> <li>• J is a join index.</li> <li>• K is a primary key.</li> </ul>                                                                                                                                                                                                            |



| Attribute    | Description                                                                                                                                                                                                                                                                                                                                                |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <ul style="list-style-type: none"> <li>• N is a hash index.</li> <li>• O is a value-ordered secondary covering index.</li> <li>• P is a nonpartitioned primary index.</li> <li>• Q is a row-partitioned primary index.</li> <li>• S is a secondary index.</li> <li>• U is a unique constraint.</li> <li>• V is a value-ordered secondary index.</li> </ul> |
| UniqueFlag   | Code indicating whether the index is unique or nonunique. <ul style="list-style-type: none"> <li>• If F, the index is nonunique.</li> <li>• If T, the index is unique.</li> </ul>                                                                                                                                                                          |
| IndexKind    | Code indicating whether the index is permanent or simulated. <ul style="list-style-type: none"> <li>• P is a permanent index.</li> <li>• S is a simulated index.</li> </ul>                                                                                                                                                                                |
| NumNulls     | The number of nulls in the index.                                                                                                                                                                                                                                                                                                                          |
| NumIntervals | The number of intervals in the index statistics.                                                                                                                                                                                                                                                                                                           |
| MinValue     | The minimum value of the index.<br>This is obtained from statistical histogram interval 0 for the index.                                                                                                                                                                                                                                                   |
| ModeValue    | The value of the index that occurs the most in the table.<br>This is obtained from statistical histogram interval 0 for the index.                                                                                                                                                                                                                         |
| ModeFreq     | The number of times the modal value occurs in the index.                                                                                                                                                                                                                                                                                                   |
| TotalValues  | The total number of values in the index other than the modal value.                                                                                                                                                                                                                                                                                        |
| TotalRows    | The cardinality of the table.                                                                                                                                                                                                                                                                                                                              |

## JoinIndexColumns

### Function of JoinIndexColumns

Captures the columns that form the join index identified by JINumber during index analysis using INITIATE INDEX ANALYSIS.

### JoinIndexColumns Table Definition

The following CREATE TABLE request defines JoinIndexColumns:

```
CREATE TABLE JoinIndexColumns (
 WorkLoadID INTEGER NOT NULL,
 RecommendationID INTEGER NOT NULL,
 TableID BYTE(6) NOT NULL,
```

```

JINumber INTEGER NOT NULL,
ColumnName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC DEFAULT NULL,
AliasName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC DEFAULT NULL,
Field1Flag CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC DEFAULT NULL,
Field2Flag CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC DEFAULT NULL,
RowIDFlag CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC DEFAULT NULL,
AggregateFunc BYTEINT DEFAULT NULL,
PrimaryIndexPosition BYTEINT DEFAULT NULL,
GroupByPosition BYTEINT DEFAULT NULL)
PRIMARY INDEX (RecommendationID, TableID, JINumber);

```

### Attribute Definitions for JoinIndexColumns

The following table defines the JoinIndexColumns table attributes:

| Attribute        | Description                                                                                                                                                                                                                                                                                      |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WorkLoadID       | Uniquely identifies the workload analyzed to create this join index recommendation.                                                                                                                                                                                                              |
| RecommendationID | <ul style="list-style-type: none"> <li>Uniquely identifies a set of index recommendations in the IndexRecommendations table.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                     |
| TableID          | <ul style="list-style-type: none"> <li>The unique internal identifier in the IndexRecommendations table for the base table on which the join index is defined.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                   |
| JINumber         | <ul style="list-style-type: none"> <li>The system-assigned sequence number for the join index in the IndexRecommendations table.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                 |
| ColumnName       | Name of the join index column.                                                                                                                                                                                                                                                                   |
| AliasName        | The correlation name assigned to a column or aggregate function in the join index definition.                                                                                                                                                                                                    |
| Field1Flag       | Indicates whether the column is part of the column_1 (uncompressed columns) select list in the join index definition. <ul style="list-style-type: none"> <li>If F, the column is not part of the column_1 select list.</li> <li>If T, the column is part of the column_1 select list.</li> </ul> |
| Field2Flag       | Indicates whether the column is part of the column_2 (compressed columns) select list in the join index definition. <ul style="list-style-type: none"> <li>If F, the column is not part of the column_2 select list.</li> <li>If T, the column is part of the column_2 select list.</li> </ul>   |

| Attribute            | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RowIDFlag            | Indicates whether the value for the column is the reserved word ROWID. <ul style="list-style-type: none"> <li>• If F, the value for the column is not ROWID.</li> <li>• If T, the value for the column is ROWID.</li> </ul>                                                                                                                                                                                                                                                                                                                   |
| AggregateFunc        | Indicates whether an aggregate function is applied to ColumnName and, if so, the type of aggregation performed. <ul style="list-style-type: none"> <li>• If 0, there is no aggregation.</li> <li>• If 1, the column is used in an aggregate SUM operation.</li> <li>• If 2, the column is used in an aggregate COUNT operation.</li> <li>• If 3, the column is used in a COUNT(*) operation.</li> <li>• If 4, the column is used in an aggregate MIN operation.</li> <li>• If 5, the column is used in an aggregate MAX operation.</li> </ul> |
| PrimaryIndexPosition | Indicates whether ColumnName is a component of the primary index for the join index. <ul style="list-style-type: none"> <li>• If 0, the column is not part of the primary index definition for the join index.</li> <li>• If &gt;1, the column is part of the primary index definition for the join index.</li> </ul> The value represents the position of ColumnName within the primary index definition for the join index.                                                                                                                 |
| GroupByPosition      | Indicates whether ColumnName is a component of the GROUP BY clause in the join index definition. <ul style="list-style-type: none"> <li>• If 0, the column is not part of the GROUP BY specification for the join index.</li> <li>• If &gt;1, the column is part of the GROUP BY specification for the join index.</li> </ul> The value represents the position of ColumnName within the GROUP BY specification for the join index.                                                                                                           |

## Partition Recommendations

### Function of PartitionRecommendations

Captures the recommended partitioning expressions generated by an INITIATE PARTITION ANALYSIS statement and identified by a WorkLoadID value.

PartitionRecommendations contains one row for each combination of query and table row partitioning where the new Optimizer plan from the recommended row partitioning either benefits or degrades workload performance as a result of the new row partitioning.

If a particular row partitioning impacts more than one query, INITIATE PARTITION ANALYSIS writes multiple rows in the table to identify each affected query. The recommended PARTITION BY expression is stored as SQL text in the ExpressionText column.

### PartitionRecommendations Table Definition

The following CREATE TABLE request defines PartitionRecommendations:

```

CREATE SET TABLE qcd.partitionrecommendations (
 WorkLoadID INTEGER NOT NULL,
 UserName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 TimeOfAnalysis TIMESTAMP(6) NOT NULL,
 RecommendationID INTEGER NOT NULL,
 QueryID INTEGER NOT NULL,
 ResultNameTag VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 TableName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 DatabaseName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 TableID BYTE(6) NOT NULL,
 ExpressionText VARCHAR(10000) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 ExpressionType CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC,
 RecreateText VARCHAR(15000) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 OriginalCost FLOAT,
 NewCost FLOAT,
 SpaceEstimate FLOAT,
 TimeEstimate FLOAT,
 StatsDDL VARCHAR(10000) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 Remarks VARCHAR(1024) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 TimeLimitExceeded CHARACTER(1) CHARACTER SET LATIN NOT
 CASESPECIFIC
 AnalysisTimeLimit INTEGER,
 AnalysisData VARCHAR(2048) CHARACTER SET LATIN
 NOT CASESPECIFIC)
PRIMARY INDEX (WorkloadID);

```

### Attribute Definitions for PartitionRecommendations

The following table defines the PartitionRecommendations table attributes:

| Attribute  | Definition                                          |
|------------|-----------------------------------------------------|
| WorkLoadID | An identifier of the workload that was analyzed.    |
| UserName   | Name of the user performing the partition analysis. |

| Attribute         | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TimeOfAnalysis    | A timestamp recording the time at which this analysis completed.                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| RecommendationID  | An identifier for the set of row partitioning recommendations this recommendation belongs to.                                                                                                                                                                                                                                                                                                                                                                                                                              |
| QueryID           | Unique identifier for the query that benefits from this recommendation.                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| ResultNameTag     | User-assigned name for the set of recommendations identified by RecommendationID.                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| TableName         | Name of the table on which the row partitioning recommendation is being made.                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| DatabaseName      | Name of the database containing TableName.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| TableID           | Unique internal identifier for TableName.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| ExpressionText    | SQL text of the recommended PARTITION BY expression.                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| ExpressionType    | The form of partitioning used by ExpressionText.<br>If R, the partitioning expression is built from a RANGE_N function.                                                                                                                                                                                                                                                                                                                                                                                                    |
| RecreateText      | <p>SQL text of statements that will recreate the table with the recommended partitioning expression.<br/>The statements assume the table is populated and so require the current rows to be temporarily stored in another table.</p> <div style="border: 1px solid black; padding: 10px; margin-top: 10px;"> <p style="text-align: center; margin: 0;"><b>NOTICE</b></p> <p style="margin: 0;">Do not execute these SQL requests without first verifying that they are compliant with your system requirements.</p> </div> |
| OriginalCost      | The estimated cost in milliseconds to execute the query without the recommendation.                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| NewCost           | The estimated cost in milliseconds to execute the query with the recommendation.<br>If this value is higher than OriginalCost, the recommendation has negatively impacted this query.                                                                                                                                                                                                                                                                                                                                      |
| SpaceEstimate     | The additional incremental space, in bytes, needed to store the table with the recommended row partitioning.                                                                                                                                                                                                                                                                                                                                                                                                               |
| TimeEstimate      | Reserved for future use.<br>Estimated time, in milliseconds, to recreate the table with the recommended row partitioning.                                                                                                                                                                                                                                                                                                                                                                                                  |
| StatsDDL          | SQL text for collecting statistics on the recommended row partitioning expression.                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Remarks           | Additional details regarding the recommendation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| AnalysisTimeLimit | User-specified time limit, in whole minutes, for the duration of the analysis.                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| TimeLimitExceeded | <p>Indicates whether or not the recommendation was generated by an INITIATE PARTITION ANALYSIS request whose specified time limit expired.</p> <ul style="list-style-type: none"> <li>• If F, the row partition analysis was not interrupted because the specified INITIATE PARTITION ANALYSIS time limit was exceeded.</li> </ul>                                                                                                                                                                                         |

| Attribute    | Definition                                                                                                                                                                                                                                                                           |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <p>This is the default.</p> <ul style="list-style-type: none"> <li>If T, the row partition analysis was interrupted because the specified INITIATE PARTITION ANALYSIS time limit was exceeded.</li> </ul> <p>This means that the final recommendations might have been affected.</p> |
| AnalysisData | Reserved for future use.                                                                                                                                                                                                                                                             |

### Example: PartitionRecommendations

The following INITIATE PARTITION ANALYSIS request makes the entries in PartitionRecommendations reported by the query that follows it:

```

INITIATE PARTITION ANALYSIS ON recent_orders
FOR MyWorkload
IN MyQCD AS IPA_recent_orders;

SELECT ExpressionText
FROM MyQCD.PartitionRecommendations
WHERE ResultNameTag = 'IPA_recent_orders';

ExpressionText

PARTITION BY RANGE_N (order_date
 BETWEEN DATE '2004-01-01'
 AND DATE '2005-12-31'
 EACH INTERVAL '1' MONTH)

```

See also [RangePartExpr](#) for an example that displays the recommended partitioning expression component parts.

## Predicate

### Function of Predicate

Describes any index, join, or residual conditions applied for specific AMP steps in a query.

### Predicate Table Definition

The following CREATE TABLE request defines the Predicate table:

```

CREATE TABLE Predicate (
 PredicateID INTEGER NOT NULL,
 StepID INTEGER NOT NULL,

```

```

PredicateKind CHARACTER(1) NOT NULL,
PredicateText VARCHAR(2000) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL
Overflow CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC
Complete CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC)
UNIQUE PRIMARY INDEX PK_PredID (PredicateID);

```

## Attribute Definitions for Predicate

The following table defines the Predicate table attributes:

| Attribute     | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PredicateID   | <ul style="list-style-type: none"> <li>Unique identifier for the predicate.</li> <li>UPI for the table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| StepID        | Unique identifier for the AMP step.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| PredicateKind | <p>Describes the kind of predicate condition associated with this step.</p> <ul style="list-style-type: none"> <li>If A, the predicate is an additional join condition.</li> <li>If G, the predicate is a range constraint.<br/>Used for value-ordered relations.</li> <li>If I, the predicate is a condition associated with an index.</li> <li>If J, the predicate is a join condition.</li> <li>If L, the predicate is a condition on the left relation in a join.</li> <li>If Q, a row 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. Row partition elimination occurs prior to accessing the rows, so the condition applies only to rows retrieved from row partitions that were not eliminated.</li> <li>If R, the predicate is a condition on the right relation in a join.</li> <li>If S, the predicate is a source condition.<br/>This is a residual condition on the left or right table in a join or on a single-table retrieval. No row partition elimination occurs prior to accessing the rows.</li> </ul> |
| PredicateText | Full text of the predicate as it appears in the EXPLAIN report.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Overflow      | Contains a value of T if the predicate text is greater than 2000 characters. Otherwise, contains a value of F.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Complete      | Contains a value of T if the complete predicate text is saved in the predicate table or qryrelx table. If a LIMIT clause exists and the specified limit size is less than the actual predicate size, then this field contains a value of F.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

## Predicate\_Field

### Function of Predicate\_Field

Associates the list of columns specified in a captured predicate with the parent relation and predicate.

## Predicate\_Field Table Definition

The following CREATE TABLE request defines the Predicate\_Field table:

```
CREATE TABLE predicate_field (
 PredicateID INTEGER NOT NULL,
 RelationKey INTEGER NOT NULL,
 FieldID INTEGER NOT NULL)
PRIMARY INDEX (RelationKey);
```

## Attribute Definitions for Predicate\_Field

The following table defines the Predicate\_Field table attributes:

| Attribute   | Definition                                                                                                                                 |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| PredicateID | Uniquely identifies the predicate.                                                                                                         |
| RelationKey | <ul style="list-style-type: none"> <li>Unique identifier for the referenced table.</li> <li>NUPI for the Predicate_Field table.</li> </ul> |
| FieldID     | Unique identifier for the column.                                                                                                          |

# QryRelX

## Function of QryRelX

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.

## QryRelX Table Definition

The following CREATE TABLE request 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 for QryRelX

The following table defines the QryRelX table attributes:



| Attribute | Definition                                                                                                                                                                                                                                                                                                           |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RowType   | Defines whether the row handles text overflow from the Query table or from the Relation table. <ul style="list-style-type: none"> <li>• If Q, the text is overflow from the Query table.</li> <li>• If R, the text is overflow from the Relations table.</li> </ul>                                                  |
| KeyValue  | Defines the implicit primary key for QryRelX rows. <ul style="list-style-type: none"> <li>• If RowType is Q, then KeyValue is the QueryID.</li> <li>• If RowType is R, then KeyValue is the RelationKey.</li> </ul>                                                                                                  |
| 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 a unique SeqNumber value.<br>The value of 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 up to a total of 1 megabyte of overflow SQL query text.                                                                                                                                                       |

## Query

### Function of Query

Describes information about captured queries.

### Query Table Definition

The following CREATE TABLE request defines the Query table:

```
CREATE TABLE Query(
 QueryID INTEGER NOT NULL,
 UDB_Key INTEGER NOT NULL,
 MachName VARCHAR(30) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 NumAMPs INTEGER NOT NULL,
 NumPEs INTEGER NOT NULL,
 NumNodes INTEGER NOT NULL,
 ReleaseInfo VARCHAR(256) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 VersionInfo VARCHAR(256) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 PENum INTEGER NOT NULL,
 QueryText VARCHAR(20000) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 CaptureTimeStamp TIMESTAMP(0) NOT NULL,
 QueryName VARCHAR128 CHARACTER SET UNICODE
```

```

Frequency NOT CASESPECIFIC,
StatementTypes INTEGER NOT NULL,
 VARCHAR(120) CHARACTER SET LATIN
DefaultDBName NOT CASESPECIFIC NOT NULL,
 VARCHAR128 CHARACTER SET UNICODE
Overflow CHARACTER(1) CHARACTER SET LATIN,
 NOT CASESPECIFIC,
Complete CHARACTER(1) CHARACTER SET LATIN,
 NOT CASESPECIFIC,
ValidatedPlan CHARACTER(1) CHARACTER SET LATIN,
 NOT CASESPECIFIC,
ImportedPlan CHARACTER(1) CHARACTER SET LATIN,
 NOT CASESPECIFIC,
SessionTemporalQualifier VARCHAR(1024) CHARACTER SET LATIN
 NOT CASESPECIFIC,
TemporalTimestamp TIMESTAMP(0),
TotalCost FLOAT DEFAULT 0),
UNIQUE PRIMARY INDEX PK_QueryID (QueryID);

```

## Attribute Definitions for Query

The following table defines the Query table attributes:

| Attribute   | Definition                                                                                                                                                                                                                                                              |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| QueryID     | <ul style="list-style-type: none"> <li>Unique identifier for the query generated by the system when the query plan is captured.</li> <li>UPI for the table.</li> </ul>                                                                                                  |
| UDB_Key     | Identifier for the user who captured the plan by submitting either a DUMP EXPLAIN, INSERT EXPLAIN, or BEGIN QUERY CAPTURE statement (see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144 for descriptions of these statements). |
| MachName    | Name of the test machine on which the query plan is captured.                                                                                                                                                                                                           |
| NumAMPs     | Number of AMPs in the configuration.                                                                                                                                                                                                                                    |
| NumPEs      | Number of PEs in the configuration.                                                                                                                                                                                                                                     |
| NumNodes    | Number of nodes in the configuration.                                                                                                                                                                                                                                   |
| ReleaseInfo | Database release number under which the query was captured. The value is defined in DBC.DBCInfo.                                                                                                                                                                        |
| VersionInfo | Database version number under which the query was captured. The value is defined in DBC.DBCInfo.                                                                                                                                                                        |
| PENum       | Number of the parsing engine on which the query was processed.                                                                                                                                                                                                          |

| Attribute        | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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 <a href="#">QryRelX</a> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| CaptureTimeStamp | 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 that is used to create the row.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| StatementTypes   | A comma-separated list of 3-character statement type codes. <ul style="list-style-type: none"> <li>• ABT is an ABORT statement.</li> <li>• BTS is a BEGIN TRANSACTION statement.</li> <li>• DEL is a DELETE statement.</li> <li>• ETS is an END TRANSACTION statement.</li> <li>• INS is an INSERT statement.</li> <li>• MRG is a MERGE statement.</li> <li>• NUL is a null statement.</li> <li>• OTR is an other statement.</li> </ul> This code describes any statement type that is not described by the other 9 statement type codes. <ul style="list-style-type: none"> <li>• RET is a retrieve (SELECT) statement.</li> <li>• UPD is an UPDATE statement.</li> <li>• URT is an Update-Retrieve statement.</li> </ul> |
| DefaultDBName    | The name of the default database at the time the query plan is captured.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Overflow         | Specifies whether QueryText exceeds 20 000 characters.<br>If QueryText exceeds 20,000 characters, then all text beyond that boundary is truncated. <ul style="list-style-type: none"> <li>• Code F indicates that QueryText is &lt;= 20 000 characters.</li> <li>• Code T indicates that QueryText is &gt; 20 000 characters and has been truncated.</li> </ul>                                                                                                                                                                                                                                                                                                                                                            |
| Complete         | Identifies whether Query stores the complete query text or a truncated version. <ul style="list-style-type: none"> <li>• F indicates that the query text is truncated.</li> </ul> The upper boundary on the number of characters stored is controlled by the LIMIT clause of the DUMP EXPLAIN and INSERT EXPLAIN statements. For more information, see <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146.<br>If no limit is specified by DUMP EXPLAIN or INSERT EXPLAIN, then full query text is captured and stored. <ul style="list-style-type: none"> <li>• T indicates that the full query text is captured and stored.</li> </ul>                                                                 |

| Attribute                | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                          | If there is overflow (indicated when the value for the Overflow attribute is T), then it is stored in the QryRelX table (see <a href="#">QryRelX</a> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| ValidatedPlan            | Specifies whether the plan was captured in validation or non-validation mode. The value is always set to F during the capture. <ul style="list-style-type: none"> <li>• F indicates that the query plan was captured in non-validation mode.</li> <li>• T indicates that the query plan was captured in validation mode.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| ImportedPlan             | Specifies whether the query plan was captured on the current system or imported from another system. The system always sets the flag to F during the capture. <ul style="list-style-type: none"> <li>• F indicates that the query plan was captured on the current system.</li> <li>• T indicates that the query plan was imported to the current system from another system.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| TemporalTimestamp        | The timestamp value derived from evaluating the TEMPORAL_TIMESTAMP function at the time the query was captured. This column is only populated for requests that either reference temporal tables or that invoke the functions TEMPORAL_TIMESTAMP or TEMPORAL_DATE. The column is set null if the captured query does not reference temporal tables or invoke the TEMPORAL_TIMESTAMP or TEMPORAL_DATE functions. See <i>Teradata Vantage™ - Temporal Table Support</i> , B035-1182 for details about temporal timestamping.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| SessionTemporalQualifier | Specifies the temporal qualifier for the session specified by the most recent SET SESSION <i>temporal_qualifier</i> request. The valid session temporal qualifiers are indicated in the following list: <ul style="list-style-type: none"> <li>• ANSIQUALIFIER</li> <li>• AS OF <i>date_expression</i></li> <li>• AS OF <i>timestamp_expression</i></li> <li>• CURRENT TRANSACTIONTIME</li> <li>• CURRENT TRANSACTIONTIME AND CURRENT VALIDTIME</li> <li>• CURRENT VALIDTIME</li> <li>• CURRENT VALIDTIME AND CURRENT TRANSACTIONTIME</li> <li>• CURRENT VALIDTIME AND NONSEQUENCED TRANSACTIONTIME</li> <li>• CURRENT VALIDTIME AND TRANSACTIONTIME AS OF <i>timestamp_expression</i></li> <li>• NONSEQUENCED TRANSACTIONTIME</li> <li>• NONSEQUENCED TRANSACTIONTIME AND CURRENT VALIDTIME</li> <li>• NONSEQUENCED VALIDTIME</li> <li>• NONSEQUENCED VALIDTIME AND CURRENT TRANSACTIONTIME</li> <li>• NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME</li> <li>• NONSEQUENCED VALIDTIME AND TRANSACTION TIME AS OF <i>timestamp_expression</i></li> <li>• SEQUENCED VALIDTIME</li> <li>• SEQUENCED VALIDTIME AND CURRENT TRANSACTIONTIME</li> </ul> |

| Attribute | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|           | <ul style="list-style-type: none"> <li>• SEQUENCED VALIDTIME <i>period_of_applicability</i></li> <li>• SEQUENCED VALIDTIME <i>period_of_applicability</i> AND CURRENT TRANSACTIONTIME</li> <li>• SEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME</li> <li>• SEQUENCED VALIDTIME <i>period_of_applicability</i> AND NONSEQUENCED TRANSACTIONTIME</li> <li>• SEQUENCED VALIDTIME <i>period_of_applicability</i> AND TRANSACTIONTIME AS OF <i>timestamp_expression</i></li> <li>• TRANSACTIONTIME AS OF <i>timestamp_expression</i></li> <li>• VALIDTIME</li> <li>• VALIDTIME <i>period_of_applicability</i></li> <li>• VALIDTIME AND CURRENT TRANSACTIONTIME</li> <li>• VALIDTIME AND NONSEQUENCED TRANSACTIONTIME</li> <li>• VALIDTIME AND TRANSACTIONTIME AS OF <i>timestamp_expression</i></li> <li>• VALIDTIME AS OF <i>date_expression</i></li> <li>• VALIDTIME AS OF <i>timestamp_expression</i></li> <li>• VALIDTIME AS OF <i>date_expression</i> AND CURRENT TRANSACTIONTIME</li> <li>• VALIDTIME AS OF <i>timestamp_expression</i> AND CURRENT TRANSACTIONTIME</li> </ul> |
| TotalCost | The estimated total cost of performing this query, expressed in milliseconds.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

## QueryBandTbl

### Function of QueryBandTbl

Describes query band-related information about captured queries.

### QueryBandTbl Definition

The following CREATE TABLE request defines QueryBandTbl:

```
CREATE QCD.QueryBandTbl (
 ID INTEGER NOT NULL,
 SessionID INTEGER FORMAT '---,---,---,--9' NOT NULL,
 QueryBand VARCHAR(12304) CHARACTER SET UNICODE NOT CASESPECIFIC),
UNIQUE PRIMARY INDEX (ID);
```

### Attribute Definitions for QueryBandTbl

The following table defines the QueryBandTbl attributes:

| Attribute | Definition                                                                  |
|-----------|-----------------------------------------------------------------------------|
| ID        | Unique ID for the captured query and unique primary index for QueryBandTbl. |

| Attribute  | Definition                                                                                            |
|------------|-------------------------------------------------------------------------------------------------------|
| SessionID  | ID for the session from which the query was captured.                                                 |
| QueryBand  | Name of the query band used to capture the query.<br>If no query band is used, the attribute is null. |
| XMLDocType | The type of XML content stored in the Text column of the XMLQCD table.                                |

## QuerySteps

### Function of QuerySteps

Each row in the table lists the attributes of any step executed by the system corresponding to the request. If the step has more than one attribute, then multiple rows are stored for the step, and the row set is linked by means of its common StepID and QueryID values. In this case, the individual rows are distinguished by their row type codes.

### QuerySteps Table Definition

The following CREATE TABLE request 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,
 EstCUPUCost FLOAT,
 EstIOWCost FLOAT,
 EstNetworkCost FLOAT,
 EstHRCost FLOAT,
 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 for QuerySteps

The following table defines the QuerySteps table attributes:

| Attribute       | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| StepID          | <ul style="list-style-type: none"> <li>• Unique identifier for the step.</li> <li>• NUSI for the table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| QueryID         | <ul style="list-style-type: none"> <li>• Unique identifier for the query.</li> <li>• NUPI for the table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| StepNum         | <p>The number of the step whose text is reported by this QuerySteps row.</p> <ul style="list-style-type: none"> <li>• If the step is not performed in parallel, StepNum indicates the step number.</li> <li>• If the step is performed in parallel, StepNum indicates the number of the main step.</li> </ul>                                                                                                                                                                                                                                                        |
| ParallelStepNum | <p>The number of the parallel step whose text is reported by this QuerySteps row.</p> <ul style="list-style-type: none"> <li>• If the step is not performed in parallel, then ParallelStepNum is 0.</li> <li>• If the step is performed in parallel, then ParallelStepNum is the number of the parallel step.</li> </ul>                                                                                                                                                                                                                                             |
| StepText        | Stores text describing the step.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| RowType         | <p>Describes the type of detail this row characterizes.</p> <ul style="list-style-type: none"> <li>• If A, the row describes attributes for a query plan step beyond the first row for that step, which is coded with a RowType of G.</li> </ul> <p>The additional attributes are described by the StepAttributeType and StepAttributeValue columns. The remaining columns in an A row type are set null.</p> <ul style="list-style-type: none"> <li>• If G, the row describes the first row of step information for a particular step of the query plan.</li> </ul> |

| Attribute | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|           | All steps have one row of this type.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| StepKind  | <p>Describes the kind of step characterized by this row.</p> <ul style="list-style-type: none"> <li>• AB is an abort step.</li> <li>• AF is an AllRowsOneAMP In-Memory Hash Join Fly step.</li> <li>• AH is an AllRowsOneAMP In-Memory Hash Join step</li> <li>• BM is a bitmap step.</li> <li>• CE is a correlated inclusion merge join step.</li> <li>• CI is a correlated exclusion product join step.</li> <li>• CJ is a correlated join step. Mel comments: DR 178936 TEXT has never been reviewed or approved, in spite of multiple attempts</li> <li>• CP is a correlated exclusion merge join step.</li> <li>• DE is a delete step.</li> <li>• EF is an exclusion dynamic hash join step. Mel comments: DR 178936 TEXT has never been reviewed or approved, in spite of multiple attempts</li> <li>• EH is an exclusion hash join step.</li> <li>• EJ is an exists join step.</li> <li>• EM is an exclusion merge join step.</li> <li>• EP is an exclusion product join step.</li> <li>• FD is a flush database step.</li> <li>• HF is a dynamic hash join step.</li> <li>• HJ is a hash star join step.</li> <li>• HS is a hash join step.</li> <li>• IF is an inclusion dynamic hash join step. Mel comments: TEXT has never been reviewed or approved, in spite of multiple attempts</li> <li>• IH is an inclusion hash join step.</li> <li>• IJ is an intersect all join step.</li> <li>• IM is an inclusion merge join step.</li> <li>• IN is an insert step.</li> <li>• IP is an inclusion product join step.</li> <li>• LK is a lock step.</li> <li>• MD is a merge delete step.</li> <li>• MF is an in-memory hash fly join</li> <li>• MG is a merge step.</li> <li>• MH is an in-memory hash join</li> <li>• MI is a minus all join step.</li> <li>• MJ is a merge join step.</li> <li>• MR is a merge into step. Mel comments: DR 178936 TEXT has never been reviewed or approved, in spite of multiple attempts</li> <li>• MS is an other step.<br/>The MS code describes any type of step not described by the other StepKind codes.</li> <li>• MT is a materialize temporary table step.</li> <li>• MU is a merge update step.</li> <li>• NJ is a nested join step.</li> </ul> |



| Attribute      | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                | <ul style="list-style-type: none"> <li>• OJ describes a not exists join step.</li> <li>• PJ is a product join step.</li> <li>• RJ is a row ID join step.</li> <li>• SA is a sample step.</li> <li>• SO is a sort step.</li> <li>• SP is a spoil step.</li> <li>• SR is a single retrieve step.</li> <li>• ST is a statistics step.</li> <li>• SU is a sum retrieve step.</li> <li>• UP is an update step.</li> <li>• US is an upsert step.</li> </ul> |
| ParallelKind   | <p>Describes whether the step can be done in parallel with its preceding step or not.</p> <ul style="list-style-type: none"> <li>• B is a beginning parallel step.</li> <li>• E is an ending parallel step.</li> <li>• P is a parallel step and can be performed in parallel with the preceding step.</li> <li>• S is a non-parallel step that is performed sequentially and cannot be performed in parallel with the preceding step.</li> </ul>      |
| AMPUUsage      | <p>Describes how AMPs are used to process the step.</p> <ul style="list-style-type: none"> <li>• A is an all-AMPs operation.</li> <li>• G is a step that operates on a group of AMPs.</li> <li>• O is a step that is a one-AMP operation.</li> <li>• T is a step that is a two-AMPs operation.</li> </ul>                                                                                                                                             |
| TriggerType    | <p>Describes triggering associated with this step.</p> <ul style="list-style-type: none"> <li>• C is a cascaded triggering statement.</li> <li>• N indicates that no triggering is involved with this statement.</li> <li>• R is a triggered statement.</li> <li>• T is a triggering statement.</li> </ul>                                                                                                                                            |
| EstCPUCost     | The CPU time for the step as estimated by the Optimizer, expressed in milliseconds.                                                                                                                                                                                                                                                                                                                                                                   |
| EstIOCost      | The I/O service time for the step as estimated by the Optimizer, expressed in milliseconds.                                                                                                                                                                                                                                                                                                                                                           |
| EstNetworkCost | The BYNET service time for the step as estimated by the Optimizer, expressed in milliseconds.                                                                                                                                                                                                                                                                                                                                                         |
| EstHRCost      | Other miscellaneous costs for the step as estimated by the Optimizer, expressed in milliseconds.                                                                                                                                                                                                                                                                                                                                                      |
| Cost           | <p>Estimated cost of performing this step, expressed in milliseconds.</p> <p>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.</p>                                                                                                                                               |
| MaxCost        | Estimated worst case cost of performing this step.                                                                                                                                                                                                                                                                                                                                                                                                    |

| Attribute         | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SourceRelation1   | <p>Number of the principal source relation in this step.</p> <p>For example, if the step is a join step and StepAttributeType is SingleRowL, then SourceRelation1 is the number of the left relation in the join.</p> <p>If StepAttributeType is SingleRowR, then SourceRelation1 is null and SourceRelation2 contains the number of the right relation in the join.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| SourceRelation2   | Number of an additional source relation in this step.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| TargetRelation1   | Number of the spool file or table for which the step operation results or acts upon.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| TargetRelation2   | Number of an additional spool file, if one is required, to hold additional results of this step.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| StepAttributeType | <p>Indicates the attribute that is described by the row.</p> <p>When the StepAttributeType column contains SnglPart, PartRg, or AllParts, the StepAttributeValue column specifies the sequence (1, 2, ...) of such a lock in the LK StepKind.</p> <ul style="list-style-type: none"> <li>AllParts specifies that Teradata locks all partitions. When RowHashFlag is T and StepAttributeType is AllParts, it specifies a RowHash-in-All-Partitions lock.</li> <li>BeginRQ occurs at the first step of the recursive block.</li> <li>EndRQ is the step number of the corresponding BeginRQ and occurs at the last step of the recursive block.</li> <li>GlobalFlag has the following codes that describe the Sum step. <ul style="list-style-type: none"> <li>F means the flag is not set.<br/>Intermediate aggregate results are computed locally.</li> <li>T means the flag is set.<br/>Intermediate aggregate results are computed globally.</li> </ul> </li> <li>GroupKey is the Grouping column and occurs at the Sum step.</li> <li>IndexNum is the Index number and occurs at one of the following steps, which indicates whether the index was used. <ul style="list-style-type: none"> <li>Abort</li> <li>BitMap</li> <li>Delete</li> <li>Single retrieve</li> </ul> </li> <li>JoinType has the following codes that describe the join type in the join step. <ul style="list-style-type: none"> <li>F is a full outer join.</li> <li>I is an inner join.</li> <li>L is a left outer join.</li> <li>R is a right outer join.</li> </ul> </li> <li>Kind has the following codes and indicates whether the samples are specified as a fraction of rows or as an absolute number of rows in the sample step. <ul style="list-style-type: none"> <li>F is a fraction sample.</li> <li>I is a fixed sample.</li> </ul> </li> <li>LeftIndex is the index number and occurs when the left relation is used in the join step.</li> <li>MergeMode has the following codes and occurs for the merge delete and merge update steps occur, indicating the merge type used in the step. <ul style="list-style-type: none"> <li>H is a match row hash.</li> <li>R is a match row ID.</li> </ul> </li> </ul> |

| Attribute          | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                    | <ul style="list-style-type: none"> <li>◦ W is a match whole row.</li> <li>• Multisrc shows whether the current JOIN, SUM, or RET step was performed as a multisource step. A value of T in the StepAttributeValue column for a Multisrc query step means that multiple JOIN, SUM, or RET steps, all of the same type but from multiple sources, were performed as a single-step optimization.</li> <li>• PartCount describes the number of hash join partitions used for a hash join and occurs only for a hash join step.</li> <li>• PartRg specifies that Teradata locks a partition range. When RowHashFlag is T and StepAttributeType is PartRg, it specifies a RowHash-in-Partition-Range lock.</li> <li>• RightIndex is the index number and occurs when the right relation is used in the join step.</li> <li>• SnglPart specifies that Teradata locks a single partition. When RowHashFlag is T and StepAttributeType is SnglPart, it specifies a RowKey lock, that is, a RowHash-in-Single-Partition lock</li> <li>• SingleRowL indicates whether single-row optimization was applied to the left relation in the join step during a DUMP EXPLAIN or an INSERT EXPLAIN operation. <ul style="list-style-type: none"> <li>◦ If the name of the left relation is available, then StepAttributeValue contains that name.</li> <li>◦ If the name of the left relation is not available, then StepAttributeValue is null.</li> </ul> </li> <li>• SingleRowR indicates whether single-row optimization was applied to the right relation in the join step during a DUMP EXPLAIN or an INSERT EXPLAIN operation. <ul style="list-style-type: none"> <li>◦ If the name of the right relation is available, then StepAttributeValue contains that name.</li> <li>◦ If the name of the right relation is not available, then StepAttributeValue is null.</li> </ul> </li> <li>• SMSKind describes one of the following types and occurs for a BitMap step indicating the set manipulation operation performed. The types are: <ul style="list-style-type: none"> <li>◦ Intersect</li> <li>◦ Minus</li> <li>◦ Union</li> </ul> </li> <li>• SourceIndex describes the index number and occurs when the index is used in the source relation in the Stat step.</li> <li>• StatOpt describes which of the following kinds of optimization was used at the Stat function step. <ul style="list-style-type: none"> <li>◦ L is a Load distribution optimization.</li> <li>◦ S is a Single AMP optimization.</li> <li>◦ Svalue describes the sample size and occurs in the Sample step.</li> <li>◦ TableIndex describes the index number and occurs at the Update step, indicating the index the step used.</li> <li>◦ True or False describes the text containing the error returned with an abort and occurs at the Abort step, indicating whether to abort when the condition is true or when the condition is false.</li> </ul> </li> </ul> |
| StepAttributeValue | Indicates the value of the attribute described by the row.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| LockType           | <p>Defines the severity of the lock specified by the query plan for this step. See <a href="#">Transaction Processing</a> for information about lock severities.</p> <ul style="list-style-type: none"> <li>• A is an ACCESS lock.</li> <li>• R is a READ lock.</li> <li>• S is a SingleWriter lock.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

| Attribute         | Definition                                                                                                                                                                                                                 |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                   | <ul style="list-style-type: none"> <li>• W is a WRITE lock.</li> <li>• X is an EXCLUSIVE lock.</li> </ul>                                                                                                                  |
| RowHashFlag       | Indicates if the table is locked on a row hash. <ul style="list-style-type: none"> <li>• F indicates that the table is not locked on a row hash.</li> <li>• T indicates that the table is locked on a row hash.</li> </ul> |
| NoWaitFlag        | Indicates if the no wait option is set for the lock step. <ul style="list-style-type: none"> <li>• F indicates that the no wait option is disabled.</li> <li>• T indicates that the no wait option is enabled.</li> </ul>  |
| Cardinality       | 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.                                               |
| IndexMaintCostEst | The cost estimated to maintain the indexes affected by this step, expressed in milliseconds.<br>This value is applicable only to steps that modify rows.                                                                   |

## RangePartExpr

### Function of RangePartExpr

Stores the individual details of a recommended range partitioning expression based on the RANGE\_N function.

Each row in this table represents one range of a given RANGE\_N function, where the range boundary values and range size are stored as SQL text strings. The set of rows belonging to a given RANGE\_N function can be identified by its common values in columns WorkloadID, RecommendationID, and TableID, and can be ordered by a sequence value that is stored in the table.

See [Partition Recommendations](#) for information about the SQL text for recommended partitioning expressions.

### RangePartExpr Table Definition

The following CREATE TABLE request defines the RangePartExpr table:

```
CREATE SET TABLE RangePartExpr (
 WorkLoadID INTEGER NOT NULL,
 RecommendationID INTEGER NOT NULL,
 TableID BYTE(6) NOT NULL,
 TestColumn VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 RangeStart VARCHAR(18) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
```

```

RangeEnd VARCHAR(18) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
RangeSize VARCHAR(50) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
RangeSequence INTEGER NOT NULL)
PRIMARY INDEX (RecommendationID, TableID);

```

### Attribute Definitions for RangePartExpr

The following table defines the RangePartExpr table attributes.

| Attribute        | Definition                                                                                                                                                  |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WorkLoadID       | An identifier of the workload that was analyzed.<br>A component of the NUPI on RangePartExpr.                                                               |
| RecommendationID | An identifier for the set of partitioning recommendations that this recommendation belongs to.<br>A component of the NUPI on RangePartExpr.                 |
| TableID          | Unique identifier for the table on which the partitioning expression recommendation is being made.                                                          |
| TestColumn       | Column name serving as the <i>test_value</i> in the RANGE_N function                                                                                        |
| RangeStart       | Starting boundary value for current range; stored in string representation                                                                                  |
| RangeEnd         | Ending boundary value for current range; stored in string representation                                                                                    |
| RangeSize        | Size of the current range as specified by an EACH clause                                                                                                    |
| RangeSequence    | Sequence number of the current range within the function expression that can be used to define the ascending order of a set of ranges.<br>Values start at 1 |

### Example: RangePartExpr

The following INITIATE PARTITION ANALYSIS request makes the entries in RangePartExpr reported by the query that follows it.

```

INITIATE PARTITION ANALYSIS ON recent_orders
FOR MyWorkload
IN MyQCD AS IPA_recent_orders;

SELECT TestColumn, RangeStart, RangeEnd, RangeSize, RangeSequence
FROM MyQCD.RangePartExpr AS rpe,
 MyQCD.PartitionRecommendations AS pr
WHERE rpe.RecommendationId = pr.RecommendationId
AND ResultNameTag = 'IPA_recent_orders';

```

| TestColumn | RangeStart        | RangeEnd          | RangeSize          | RangeSequence |
|------------|-------------------|-------------------|--------------------|---------------|
| -----      | -----             | -----             | -----              |               |
| -----      |                   |                   |                    |               |
| order_date | DATE '2004-01-01' | DATE '2005-12-31' | INTERVAL '1' MONTH | 1             |

See also [Example: PartitionRecommendations](#) for an example that displays the recommended partition expression as text.

## Relation

### Function of Relation

Describes all table and spool files in the access plan for the captured query.

### Relation Table Definition

The following CREATE TABLE request defines the Relation table:

```
CREATE TABLE Relation(
 RelationKey INTEGER NOT NULL,
 QueryID INTEGER NOT NULL,
 UDB_Key INTEGER NOT NULL,
 Name VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 RelationID INTEGER NOT NULL,
 RelationKind CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC NOT NULL,
 SortInfo CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC,
 SortKind CHARACTER(3) CHARACTER SET LATIN
 NOT CASESPECIFIC,
 SortKey VARCHAR(1024) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 GeogInfo CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC,
 Cached CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC, NOT NULL,
 SyncScan CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC NOT NULL,
 Cardinality FLOAT,
 Confidence CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC,
 MaxCardinality FLOAT,
 ViewName VARCHAR128 CHARACTER SET UNICODE
```

```

TableDDL UPPERCASE NOT CASESPECIFIC,
 VARCHAR(20000) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
TableName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC,
PartitionInfo CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC NOT NULL,
Overflow CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC,
Complete CHARACTER(1) CHARACTER SET LATIN,
 NOT CASESPECIFIC,
Version SMALLINT,
SpoolCompressedAllowed CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC,
SpoolSize FLOAT,
RelationAttributeType CHARACTER(10) CHARACTER SET LATIN
 NOT CASESPECIFIC
RelationAttributeValue VARCHAR(100) CHARACTER SET LATIN
 NOT CASESPECIFIC
TemporalProperty CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC NOT NULL,
ResultTemporalProperty CHARACTER(1) CHARACTER SET LATIN
 NOT CASESPECIFIC NOT NULL,
NumCombinedPartitions BIGINT,
NumContexts INTEGER,
NumCPReferences INTEGER)
PRIMARY INDEX (QueryID),
UNIQUE INDEX (RelationKey);

```

## Attribute Definitions for Relation

The following table defines the Relation table attributes:

| Attribute   | Definition                                                                                                                                                                       |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RelationKey | <ul style="list-style-type: none"> <li>Unique identifier for the relation within its database.</li> <li>USI for the Relation table.</li> </ul>                                   |
| QueryID     | <ul style="list-style-type: none"> <li>Unique identifier for the query generated by the system when the query plan is captured.</li> <li>NUPI for the Relation table.</li> </ul> |
| UDB_Key     | Identifier for the user or database containing the relation described by this row.                                                                                               |
| Name        | One of the following: <ul style="list-style-type: none"> <li>Alias name of the table.</li> </ul>                                                                                 |

| Attribute    | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <ul style="list-style-type: none"> <li>• The word SPOOL.</li> <li>• The word ROWIDSPPOOL.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| RelationID   | Unique identifier for the relation or spool file within the database.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| RelationKind | <p>Distinguishes among derived tables, global temporary tables, hash indexes, join indexes, permanent tables, volatile tables, and spool files.</p> <ul style="list-style-type: none"> <li>• D is a derived table.<br/>The Name attribute contains the name of the derived table.</li> <li>• G is a global temporary table.</li> <li>• H is a hash index.</li> <li>• J is a join index.</li> <li>• O is a NoPI table.</li> <li>• P is a permanent table.<br/>This does not differentiate between ordinary permanent tables and queue tables.<br/>To make this discrimination, you can join Relation with DBC.TVM or with the DBC.Tables view, where TVM.QueueFlag=T or Tables.QueueFlag=T.<br/>T is the DBC.TVM.QueueFlag value that indicates a queue table.</li> <li>• S is a spool file.</li> <li>• V is a volatile table.</li> </ul>                      |
| SortInfo     | <p>Describes whether the relation is sorted or not.</p> <ul style="list-style-type: none"> <li>• If F, the relation is not sorted.</li> <li>• If T, the relation is sorted.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| SortKind     | <p>The way the relation is sorted.<br/>Only used when the value for SortInfo is T.</p> <ul style="list-style-type: none"> <li>• F1S is a Field1 sort.</li> <li>• F1U is a Field1 unique sort.</li> <li>• FHS is a Field1 Hash sort.</li> <li>• FID is a FieldID sort.</li> <li>• FHU is a Field1 Hash unique sort.</li> <li>• HN1 is a Field1 Hash min1 sort.</li> <li>• HN2 is a Field1 Hash min2 sort.</li> <li>• HX1 is a Field1 Hash max1 sort.</li> <li>• HX2 is a Field1 Hash max2 sort.</li> <li>• JIS is a JoinIndex sort.</li> <li>• MN1 is a Field1 min1 sort.</li> <li>• MN2 is a Field1 min2 sort.</li> <li>• MX1 is a Field1 max1 sort.</li> <li>• MX, is a Field1 max2 sort.</li> <li>• RF1 is a Rowhash field1 sort.</li> <li>• RHR is a RowHashRow sort.</li> <li>• RHS is a Rowhash sort.</li> <li>• UF1 is a Unique field1 sort.</li> </ul> |



| Attribute      | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                | <ul style="list-style-type: none"> <li>• UNK is an Unknown sort kind.</li> <li>• URS is a Unique rowID sort.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| SortKey        | <p>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.</p> <p>The format for the SortKey list is the following:</p> <pre>SortKey1, SortKey2, ..., SortKeyN</pre> <p>There must be a SPACE character following each COMMA character in the list. The format for the individual SortKey strings is one of the following:</p> <ul style="list-style-type: none"> <li>• database_name.table_name.column_name</li> <li>• spool_number.column_name</li> </ul> <p>Spool numbers are used in place of database.table names whenever the table information is not available.</p> |
| GeogInfo       | <p>Describes the configuration geography of the relation.</p> <ul style="list-style-type: none"> <li>• A means the relation is duplicated on one AMP (AllRowsOneAMP).</li> <li>• D means the relation is duplicated on all AMPs.</li> <li>• H means the relation is hash-distributed across the AMPs.</li> <li>• L means the relation is built locally.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                     |
| Cached         | <p>Describes whether the relation is cached or not.</p> <ul style="list-style-type: none"> <li>• If F, the relation is not cached.</li> <li>• If T, the relation is cached.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| SyncScan       | <p>Describes whether a relation is eligible for synchronized scanning or not.</p> <ul style="list-style-type: none"> <li>• If F, the relation is not eligible for synchronized scanning.</li> <li>• If T, the relation is eligible for synchronized scanning.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Cardinality    | Estimated cardinality of the relation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Confidence     | <p>Describes the confidence level for the estimated cardinality.</p> <ul style="list-style-type: none"> <li>• H is High confidence.</li> <li>• I is Index Join confidence.</li> <li>• L is Low confidence.</li> <li>• N is No confidence.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| MaxCardinality | Estimated maximum cardinality of the relation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| ViewName       | Name of a view, if any, used by the query to access the relation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| TableDDL       | <p>The SQL DDL text for the captured relation.</p> <p>If the text exceeds the upper limit of 20,000 characters, then it overflows to the QryRelX table. See Overflow below and <a href="#">QryRelX</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| TableName      | The non-aliased name of the relation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

| Attribute              | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                        | Compare with Name above.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| PartitionInfo          | Identifies whether a table or spool has a row-partitioned primary index. <ul style="list-style-type: none"> <li>• If F, the relation does not have a row-partitioned primary index.</li> <li>• If T, the relation has a row-partitioned primary index.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Overflow               | Specifies whether there is overflow query text stored in QryRelX or not. <ul style="list-style-type: none"> <li>• If F, there is no overflow.<br/>This is the default.</li> <li>• If T, any overflow text is stored in a QryRelX table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Complete               | Identifies whether Relation stores the complete relation DDL or a truncated version. <ul style="list-style-type: none"> <li>• If F, table DDL 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 <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146). If no limit is specified by DUMP EXPLAIN or INSERT EXPLAIN, then full DDL is captured and stored.</li> <li>• If T, full table DDL is 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 <a href="#">QryRelX</a>).</li> </ul> |
| Version                | Stores the version number of the table at the time the plan was captured. Used to ensure that any changes to the schema information captured for analysis are handled correctly.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| SpoolCompressedAllowed | Identifies whether target spool files can contain compressed columns or not. <ul style="list-style-type: none"> <li>• If F, the spool cannot contain compressed columns.</li> <li>• If T, the spool can contain compressed columns.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| SpoolSize              | The size of the spool file for this relation in bytes.<br>If there is no spool for the request, the column is null.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| RelationAttributeType  | Indicates that at least one row partition of a partitioned table was accessed by a query.<br>Each time a row is written to Relation for a row-partitioned table, Vantage writes the string NUMOFPART to RelationAttributeType.<br>If Vantage does not access any row partitions for a query, RelationAttributeType is null.<br>The function of this column is analogous to the function of QCD. QuerySteps.StepAttributeType.                                                                                                                                                                                                                                                                            |
| RelationAttributeValue | The number of row partitions accessed by a query when that number is greater than zero.<br>If Vantage does not access any row partitions for a query, RelationAttributeValue is null.<br>The function of this column is analogous to the function of QCD. QuerySteps.StepAttributeValue.                                                                                                                                                                                                                                                                                                                                                                                                                 |
| TemporalProperty       | The temporal attribute for the relation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

| Attribute              | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                        | <ul style="list-style-type: none"> <li>• B is a Bitemporal relation.</li> <li>• N is not a temporal relation.</li> <li>• S is a system-versioned system-time table.</li> <li>• T is a Transaction Time relation.</li> <li>• U is a bitemporal (system-versioned system-time and valid-time) table.</li> <li>• V is a Valid Time relation.</li> <li>• W is a nontemporal table that contains a system-time derived period column but that is not system versioned.</li> <li>• X is a valid-time temporal table that contains a system-time derived period column but that is not system versioned.</li> </ul> |
| ResultTemporalProperty | <p>The temporal attribute for the result relation.</p> <ul style="list-style-type: none"> <li>• B is a Bitemporal relation.</li> <li>• N is not a temporal relation.</li> <li>• T is a Transaction Time relation.</li> <li>• V is a Valid Time relation.</li> </ul>                                                                                                                                                                                                                                                                                                                                          |
| NumCombinedPartitions  | <p>If there is static partition elimination for the relation or if the relation has column partitions, NumCombinedPartitions contains the number of combined partitions that were accessed, not the number of combined partitions that were eliminated.</p> <p>Otherwise, the column is null.</p>                                                                                                                                                                                                                                                                                                            |
| NumContexts            | <p>The number of contexts allocated, if any, to simultaneously access the partitions of a relation that has partitioning.</p> <p>Otherwise, the column is null.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| NumCPReferences        | <p>The number of column partitions referenced in a column-partitioned relation.</p> <p>Otherwise, the column is null.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

## SeqNumber

### Function of SeqNumber

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.

### SeqNumber Table Definition

The following CREATE TABLE request 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 request.

```
INSERT INTO SeqNumber VALUES(1,1,1,1,1,1,1,1,1,1,'QCF03.00.00',2,2);
```

### Attribute Definitions for SeqNumber

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.                                                                                                |
| 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 request.                   |

| Attribute | Definition          |
|-----------|---------------------|
|           | Initially set to 2. |

## SingleRowRelation

### Function of SingleRowRelation

Captures the row returned by a single row relation in the form of an INSERT statement. This statement is executed by TSET at a later time.

### SingleRowRelation Table Definition

```
CREATE SET TABLE SingleRowRelation (
 QueryID INTEGER NOT NULL,
 DatabaseName VARCHAR(128) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 TableName VARCHAR(128) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 InsertStatement VARCHAR(10000) CHARACTER SET UNICODE
 NOT CASESPECIFIC)
PRIMARY INDEX (QueryID);
```

### Attribute Definitions for SingleRowRelation

The following table defines the SingleRowRelation table attributes:

| Attribute       | Description                                                                                                                                                     |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| QueryID         | <ul style="list-style-type: none"> <li>• Unique identifier for the query to which the recommendation applies.</li> <li>• Partial NUPI for the table.</li> </ul> |
| DatabaseName    | Name of the containing database for TableName.                                                                                                                  |
| TableName       | Name of the table in which to insert the statement.                                                                                                             |
| InsertStatement | Statement to insert into the specified table.                                                                                                                   |

## StatsRecs

### Function of StatsRecs

Each row set contains a COLLECT STATISTICS request and related information for collecting the recommended statistics generated by a DUMP EXPLAIN or INSERT EXPLAIN request specified with a CHECK STATISTICS clause.

## StatsRecs Table Definition

The following CREATE TABLE request defines the StatsRecs table:

```
CREATE SET TABLE StatsRecs (
 QueryID INTEGER,
 StatsID INTEGER,
 DatabaseName VARCHAR128 CHARACTER SET UNICODE NOT CASESPECIFIC,
 TableName VARCHAR128 CHARACTER SET UNICODE NOT CASESPECIFIC,
 FieldID INTEGER,
 FieldName VARCHAR128 CHARACTER SET UNICODE NOT CASESPECIFIC,
 Level INTEGER,
 StatsDDL VARCHAR(2500) CHARACTER SET UNICODE NOT CASESPECIFIC)
PRIMARY INDEX (QueryID, StatsID);
```

## Attribute Definitions for StatsRecs

The following table defines the StatsRecs table attributes:

| Attribute    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| QueryID      | <ul style="list-style-type: none"> <li>Unique identifier for the query to which the recommendation applies.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| StatsID      | <ul style="list-style-type: none"> <li>Uniquely identifies a statistics collection recommendation for QueryID. There are multiple StatsID values corresponding to a single QueryID for multicolumn statistics recommendations. In other words, a set of rows having the same value in StatsID represents all the columns for a given recommendation.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                  |
| DatabaseName | Name of the containing database for TableName.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| TableName    | Name of the table in which FieldName is defined.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| FieldID      | Unique identifier for FieldName within TableName.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| FieldName    | Name of the column in the statistics recommendation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Level        | <p>A representation of the confidence the Optimizer has in the usefulness of the statistics recommendations it has generated for this query-column set combination. The level is determined by a number of factors, including the following:</p> <ul style="list-style-type: none"> <li>The number of columns in the recommendation.</li> <li>Whether the recommendations are for collecting single column statistics or multicolumn statistics.</li> </ul> <p>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.</p> <p>You can also aggregate levels to rank different recommendations.</p> |

| Attribute | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|           | <ul style="list-style-type: none"> <li>Level 1 is the primary recommendation, also referred to as a recommendation made with High Confidence.<br/>This recommendation is more likely to help the Optimizer to generate the least costly plan possible than any others, and you should always implement them unless you have very good reasons not to.<br/>Recommendations to collect single-column and multicolumn statistics with all the columns are considered to be primary.</li> <li>Level 2 is an optional or alternative recommendation for the multicolumn statistics, also referred to as a recommendation made with Low Confidence.<br/>Optional recommendations provide an alternative to collecting statistics on multiple unindexed columns. These recommendations can be useful when their columns are infrequently specified together as an equality condition within the given workload or when your site determines that it would be too costly to collect statistics on several different combinations of multicolumn recommendations, but unlike the case for Level 1 recommendations, you should not feel compelled to implement them.<br/>Optional recommendations are provided based on information collected from usable indexes and usable columns from the table described by the TableName column.</li> </ul> |
| StatsDDL  | <p>The DDL text of the COLLECT STATISTICS statement used to populate this row set of the table.</p> <p>For multicolumn statistics, the text of the DDL statement is stored in the row having the lowest FieldID value. In this case, the content of this column for the other rows is null.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

## TableStatistics

### Function of TableStatistics

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.

### TableStatistics Table Definition

The following CREATE TABLE request defines the TableStatistics table:

```
CREATE TABLE TableStatistics (
 MachineName VARCHAR(30) CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC NOT NULL,
 TableName VARCHAR128 CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC NOT NULL,
 DatabaseName VARCHAR128 CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC NOT NULL,
 IndexName VARCHAR128 CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC,
```

```

ColumnName VARCHAR128 CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC NOT NULL,
CollectedTime TIMESTAMP(6) NOT NULL,
SamplePercent FLOAT NOT NULL,
IndexType CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC
NOT NULL,
IndexID INTEGER,
StatisticsInfo VARBYTE(16383),
DataType SMALLINT,
DataLength INTEGER,
Attributes VARCHAR(256) CHARACTER SET LATIN NOT CASESPECIFIC,
ModifiedTime TIMESTAMP(6),
ModifiedStats VARBYTE(16383),
QueryId INTEGER,
FieldPosition BYTEINT,
StatisticsID INTEGER)
PRIMARY INDEX (MachineName, DatabaseName, TableName);

```

### Attribute Definitions for TableStatistics

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.                                                                                                                                                                                                                                                                                                                                                               |
| IndexName     | Name of the index, in case the index is named. Otherwise, IndexName is NULL.                                                                                                                                                                                                                                                                                                                                         |
| ColumnName    | Name of the column if the statistics are on a column.                                                                                                                                                                                                                                                                                                                                                                |
| CollectedTime | The timestamp value when the statistics were collected.                                                                                                                                                                                                                                                                                                                                                              |
| SamplePercent | The sample percentage of rows read to collect the statistics recorded in this row.                                                                                                                                                                                                                                                                                                                                   |
| IndexType     | Flag indicating whether the statistics represented by this row pertain to an index. <ul style="list-style-type: none"> <li>• D is a PARTITION column.</li> <li>• N is a non indexed column.</li> <li>• P is a pseudo-index.</li> <li>• Y is for the index specified by IndexID.</li> </ul>                                                                                                                           |
| IndexID       | The unique identifier for the index to which the statistics represented by this row pertain. <ul style="list-style-type: none"> <li>• If IndexType is N, IndexID is null.</li> <li>• If IndexType is P, IndexID is the ID value generated by scanning the existing index set for the table and incrementing the largest value.</li> <li>• If IndexType is Y, IndexID is the ID value assigned by Vantage.</li> </ul> |



| Attribute      | Definition                                                                                                                                                                                                                           |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| StatisticsInfo | Contains the column or index statistics.                                                                                                                                                                                             |
| DataType       | Indicates the data type of the column. The value is the same that is returned in the data type column in the PreplInfo CLlv2 parcel.                                                                                                 |
| DataLength     | The maximum length of the column in bytes.                                                                                                                                                                                           |
| Attributes     | Set null and reserved for future use.                                                                                                                                                                                                |
| ModifiedTime   | If the statistics have been modified, indicates the timestamp value when the last modification occurred.                                                                                                                             |
| 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. |
| 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 column 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 of User\_Database

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.

### User\_Database Table Definition

The following CREATE TABLE request 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 VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX PK_UDB_KEY (UDB_Key);
```

## Attribute Definitions for User\_Database

The following table defines the User\_Database table attributes:

| Attribute   | Definition                                                                                                                |
|-------------|---------------------------------------------------------------------------------------------------------------------------|
| UDB_Key     | <ul style="list-style-type: none"> <li>Unique identifier for the user or database.</li> <li>UPI for the table.</li> </ul> |
| 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

Stores user comments about captured plans written using the Teradata System Emulation Tool utility.

### UserRemarks Table Definition

The following CREATE TABLE request defines the UserRemarks table:

```
CREATE TABLE UserRemarks(
 QueryID INTEGER,
 StepID INTEGER,
 WorkloadID INTEGER,
 RowType CHARACTER(1) CHARACTER SET LATIN NOT CASESPECIFIC
 NOT NULL,
 SeqNumber INTEGER NOT NULL,
 UpdTime TIMESTAMP(6) NOT NULL,
 UserName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 Remarks VARCHAR31000 CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL)
PRIMARY INDEX (QueryID, UserName);
```

## Attribute Definitions for UserRemarks

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. |

| Attribute  | Definition                                                                                                                                                                                                                                                |
|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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 the utility that collected the remarks and whether they were saved or generated. <ul style="list-style-type: none"> <li>If T, the comments were generated automatically by the Teradata System Emulation Tool utility during import.</li> </ul> |
| SeqNumber  | A system-generated number (1 for the first row, 2 for second, and so on) to indicate the sequential order of remarks.                                                                                                                                     |
| UpdTime    | The timestamp when the remark was last updated.                                                                                                                                                                                                           |
| UserName   | The system user name of the user who last updated the row.<br>Partial NUPI for the table.                                                                                                                                                                 |
| Remarks    | The remarks of the step or query.                                                                                                                                                                                                                         |

## Related Information

For more information about the Teradata System Emulation Tool utility, see *Teradata® System Emulation Tool User Guide*, B035-2492.

# ViewTable

## Function of ViewTable

Captures the DDL for any views used in a query.

## ViewTable Table Definition

The following CREATE TABLE request defines ViewTable:

```
CREATE SET TABLE ViewTable(
 QueryID INTEGER NOT NULL,
 ViewName VARCHAR128 CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC NOT NULL,
 DBName VARCHAR128 CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC NOT NULL,
 ViewText VARCHAR(30000) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 SeqNumber BYTEINT NOT NULL,
 Complete CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY INDEX QueryID_ViewName_DBName (QueryID,ViewName,DBName);
```

## Attribute Definitions for ViewTable

The following table defines the ViewTable attributes:

| Attribute | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| QueryID   | <ul style="list-style-type: none"> <li>Unique identifier for the query.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| ViewName  | <ul style="list-style-type: none"> <li>Name of the view.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| DBName    | <ul style="list-style-type: none"> <li>Name of the database.</li> <li>Partial NUPI for the table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| ViewText  | <p>Stores the DDL view text.</p> <ul style="list-style-type: none"> <li>If ViewText &lt;= 30,000 characters, SeqNumber is set to 1.</li> <li>If ViewText &gt; 30,000 characters, SeqNumber is set to 1 + (1-<i>n</i>), where <i>n</i> is the identifier for the fragment of ViewText stored in the row following the first fragment.</li> </ul> <p>When ViewText &gt; 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.</p>                                                                                                             |
| SeqNumber | The sequence of ViewText stored in this row.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Complete  | <p>Identifies whether ViewTable stores the complete view DDL or a truncated version.</p> <ul style="list-style-type: none"> <li>If F, ViewText DDL is truncated.           <p>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.</p> <p>See <i>Teradata Vantage™ - SQL Data Manipulation Language</i>, B035-1146.</p> </li> <li>If T, Full ViewText DDL is stored.           <p>Overflow ViewText DDL is stored in additional rows in ViewTable. Individual overflow rows are identified by their respective SeqNumber.</p> </li> </ul> |

## Workload

### Function of Workload

Workload table has one entry for each workload defined in the database.

### Workload Table Definition

The following CREATE TABLE request defines the Workload table:

```
CREATE TABLE Workload(
 workloadID INTEGER NOT NULL,
 workloadName VARCHAR128 CHARACTER SET UNICODE
 UPPERCASE NOT CASESPECIFIC NOT NULL,
```

```

 CreatedTimeStamp TIMESTAMP(6) NOT NULL,
 LastModified TIMESTAMP(6) NOT NULL,
 CreatorName VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC)
 UNIQUE PRIMARY INDEX(WorkLoadName),
 UNIQUE INDEX(WorkLoadID);

```

### Attribute Definitions for Workload

The following table defines the Workload table attributes:

| Attribute        | Definition                                                                                                                                                   |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WorkloadID       | <ul style="list-style-type: none"> <li>Internally generated unique (within the database) identifier for the workload.</li> <li>USI for the table.</li> </ul> |
| WorkloadName     | <ul style="list-style-type: none"> <li>User-specified unique workload name.</li> <li>UPI for the table.</li> </ul>                                           |
| 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 of WorkloadQueries

Captures the queries that belong to the workload named by WorkloadID.

### WorkloadQueries Table Definition

The following CREATE TABLE request 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 for WorkloadQueries

The following table defines the WorkloadQueries table attributes:

| Attribute  | Definition                                                                                                                                                                                 |
|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WorkloadID | <ul style="list-style-type: none"> <li>The workload ID that identifies the data in the row.</li> <li>NUPI for the table.</li> <li>One half of the USI column set for the table.</li> </ul> |
| QueryID    | <ul style="list-style-type: none"> <li>Unique qualifier for a query. Used to map it to the workload.</li> <li>One half of the USI column set for the table.</li> </ul>                     |
| Frequency  | Indicates the number of times the query is typically performed in the workload.                                                                                                            |

## WorkloadStatus

### Function of WorkloadStatus

Maintains the status of workloads in the QCD using INITIATE INDEX ANALYSIS.

### WorkloadStatus Table Definition

The following CREATE TABLE request defines the WorkloadStatus table:

```
CREATE TABLE WorkloadStatus (
 WorkloadID INTEGER NOT NULL,
 RecommendationID INTEGER,
 IndexNameTag VARCHAR128 CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 ValidatedSystem VARCHAR(30) CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 ValidatedTimeStamp TIMESTAMP(6),
 ValidatedQCD VARCHAR128 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 for WorkloadStatus

The following table defines the WorkloadStatus table attributes:

| Attribute  | Definition                                                                                                                |
|------------|---------------------------------------------------------------------------------------------------------------------------|
| WorkloadID | <ul style="list-style-type: none"> <li>Unique identifier for the workload.</li> <li>Partial UPI for the table.</li> </ul> |

| Attribute          | Definition                                                                                                                                                                                                                                                                 |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RecommendationID   | Unique identifier for the set of indexes recommended for this workload after an index analysis.                                                                                                                                                                            |
| IndexNameTag       | <ul style="list-style-type: none"> <li>Name of the index recommendation as specified in the INITIATE INDEX ANALYSIS statement (see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144).</li> <li>Partial UPI for the table.</li> </ul> |
| ValidatedSystem    | Name of the system on which index recommendations are last validated.                                                                                                                                                                                                      |
| ValidatedTimeStamp | Timestamp of the index validation.                                                                                                                                                                                                                                         |
| ValidatedQCD       | Name of the QCD on which the validation is performed.                                                                                                                                                                                                                      |
| RecosAppliedSystem | Name of the system on which index recommendations are last created.                                                                                                                                                                                                        |
| Remarks            | Free form textual remarks about the workload.                                                                                                                                                                                                                              |

## XMLQCD

### Function of XMLQCD

Captures the XML QCD output from an INSERT EXPLAIN or BEGIN QUERY CAPTURE request submitted with the IN XML option.

### XMLQCD Table Definition

The following CREATE TABLE request defines the XMLQCD table:

```
CREATE MULTISET TABLE qcd.XMLQCD, NO FALLBACK, NO BEFORE JOURNAL,
NO AFTER JOURNAL, CHECKSUM = DEFAULT (
 ID INTEGER NOT NULL,
 Kind CHARACTER(1) NOT NULL CHARACTER SET LATIN
 NOT CASESPECIFIC,
 Seq INTEGER NOT NULL,
 Length INTEGER NOT NULL,
 Text VARCHAR(31000) NOT NULL CHARACTER SET UNICODE
 NOT CASESPECIFIC,
 UDB_Name VARCHAR(128) CHARACTER SET UNICODE
 NOT CASESPECIFIC NOT NULL,
 CaptureTimeStamp TIMESTAMP(0) NOT NULL,
 SessionID INTEGER FORMAT '---,---,---,---9' NOT NULL,
 WorkloadName VARCHAR(128) CHARACTER SET UNICODE UPPERCASE
 NOT CASESPECIFIC NOT NULL,
```

```
XMLDocType INTEGER FORMAT '---,---,---,---9')
UNIQUE PRIMARY INDEX (ID, Kind, Seq);
```

### Attribute Definitions for XMLQCD

The following table defines the XMLQCD table attributes.

| Attribute        | Definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ID               | <ul style="list-style-type: none"> <li>Unique identifier for the set of rows associated with this XML document.</li> <li>Partial UPI for the table.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Kind             | <ul style="list-style-type: none"> <li>A flag that describes the kind of operation characterized by this row.</li> <li>Partial UPI for the table.</li> </ul> <p>If Q, Vantage captures the value for ID from qcd.SeqNumber.QueryID (see <a href="#">SeqNumber</a>) whenever this query capture operation is performed.</p>                                                                                                                                                                                                                                                            |
| Seq              | <ul style="list-style-type: none"> <li>Unique identifier for the query that specifies the order of the generated row.</li> </ul> <p>The first row generated from a given XML document is assigned the sequence number 1, the second row is assigned a sequence number 2, and so on.</p> <ul style="list-style-type: none"> <li>Partial UPI for the table.</li> </ul>                                                                                                                                                                                                                  |
| Length           | <p>The number of characters that remain in the XML document beginning with this row. The equation for determining the approximate value for Length is as follows:</p> $\text{Length} = (\text{Total number of characters} - ((\text{Seq} - 1) \times 31000))$ <p>For example,</p> <ul style="list-style-type: none"> <li>If Seq is 1, then Length is the total number of characters.</li> <li>If Seq is 2, then Length is the total number of characters minus 31,000.</li> <li>If Seq is 3, then Length is the total number of characters minus 62,000.</li> </ul> <p>and so on.</p> |
| Text             | The Seq <sup>th</sup> slice of the XML document.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| UDB_Name         | Name of the user who submitted the captured query.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| CaptureTimeStamp | Time stamp of when the query was captured.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| SessionID        | ID for the session in which the query was captured.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| WorkloadName     | Name of the workload specified in the BEGIN QUERY CAPTURE request used to capture the query.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| XMLDocType       | The type of XML content stored in the Text column.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |



# XML Documents Produced by the Query Logging XMLPLAN Option

This section contains an example XML Optimizer Query Plan document produced by the XMLPLAN option for BEGIN QUERY LOGGING and REPLACE QUERY LOGGING statements.

It also provides the URL for the XML schema used for the XMLPLAN.

## XML Schema Used for the XMLPLAN Option

For details, see the schema. The XML schema that Vantage uses for the XMLPLAN option for the BEGIN QUERY LOGGING and REPLACE QUERY LOGGING statements is maintained at the following URL:

<http://schemas.teradata.com/queryplan/queryplan.xsd>.

## Example XML Optimizer Query Plan Document Produced by the XMLPLAN Option

This section provides a sample XML document instance that is generated by the XMLPLAN logging option for a typical BEGIN QUERY LOGGING or REPLACE QUERY LOGGING request.

The document instance represents the query logging information captured for the following SELECT request.

```

SELECT name, deptname, salary
FROM department d, employee e
WHERE d.deptno = e.deptno AND e.yrsexp >= 5 order by 3 desc;
<?xml version="1.0" encoding="UTF-16" standalone="no" ?>
<QryPlanXML xmlns="http://schemas.teradata.com/queryplan"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
schemas.teradata.com/queryplan http://schemas.teradata.com/
queryplan/queryplan.xsd">
 <Query CollectTimeStamp="2015-02-24T03:38:23.65"
QFCaptureTimeStamp="2015-05-06T19:36:15.75"
QueryID="307191055761854061" StartTime="2015-05-06T19:36:15.75">
 <Request CheckpointNum="0" DefaultDatabase="PERSONNEL" MaxStepMemory="3.437"
NumResultRows="3" QueryText="SELECT name (CHAR(10)), deptname (CHAR(10)),
salaryFROM department d, employee eWHERE d.deptno = e.deptno AND e.yrsexp >= 5
order by 3 desc;">
 <Statement QCFStatementType="RET" StatementGroup="Select"
StatementType="Select"/>
 </Request>
 <ObjectDefs>
 <Database DatabaseName="PERSONNEL" Id="DBPERSONNEL">

```

```

 <Relation Cardinality="4" Confidence="High" DatabaseId="DBPERSONNEL" Id="REL1"
Partitioned="false" RelationKind="Permanent" TableName="department" Version="1">
 <Field DataLength="4" FieldID="1025" FieldName="deptno" FieldType="I"
Id="REL1_FLD1025" JoinAccessFrequency="1" RangeAccessFrequency="0"
RelationId="REL1" ValueAccessFrequency="0"/>
 <Index Id="REL1_IDX1" IndexNum="1" IndexType="Nonpartitioned Primary"
OrderBy="false" QCFIndexFlag="false" RelationId="REL1" UniqueFlag="false">
 <FieldRef Ref="REL1_FLD1025"/>
 </Index>
 </Relation>
 <Relation Cardinality="12" Confidence="High" DatabaseId="DBPERSONNEL" Id="REL2"
Partitioned="false" RelationKind="Permanent" TableName="employee" Version="1">
 <Field DataLength="4" FieldID="1025" FieldName="employeeID"
FieldType="I" Id="REL2_FLD1025" JoinAccessFrequency="0" RangeAccessFrequency="0"
RelationId="REL2" ValueAccessFrequency="0"/>
 <Field DataLength="4" FieldID="1027" FieldName="deptno" FieldType="I"
Id="REL2_FLD1027" JoinAccessFrequency="1" RangeAccessFrequency="0"
RelationId="REL2" ValueAccessFrequency="0"/>
 <Field DataLength="4" FieldID="1029" FieldName="yrsexp" FieldType="I"
Id="REL2_FLD1029" JoinAccessFrequency="0" RangeAccessFrequency="1"
RelationId="REL2" ValueAccessFrequency="0"/>
 <Index Id="REL2_IDX1" IndexNum="1" IndexType="Nonpartitioned Primary"
OrderBy="false" QCFIndexFlag="false" RelationId="REL2" UniqueFlag="false">
 <FieldRef Ref="REL2_FLD1025"/>
 </Index>
 </Relation>
 </Database>
 <Database DatabaseName="DBC" Id="DBDBC">
 <Spool Cardinality="4" Compressible="true" Confidence="No" DatabaseId="DBDBC"
GeogInfo="Hash Distributed" Id="SPOOL2" Kind="Regular" Partitioned="false"
SpoolNumber="2" SpoolSize="432">
 <Field DataLength="4" FieldID="1027" FieldName="deptno" FieldType="I"
Id="SPOOL2_FLD1027" JoinAccessFrequency="0" RangeAccessFrequency="0"
RelationId="SPOOL2" ValueAccessFrequency="0"/>
 </Spool>
 <Spool Cardinality="6" Compressible="false" Confidence="No" DatabaseId="DBDBC"
GeogInfo="Local" Id="SPOOL1" Kind="Regular" Partitioned="false" SpoolNumber="1"
SpoolSize="312"/>
 </Database>
</ObjectDefs>
<Plan CacheFlag="False" IPEEligibility="NotEligible"
NumSteps="6" PlanType="Static">
 <PlanStep QCFParallelKind="Sequential" QCFStepKind="LK" QCFStepNum="1"
StepLev1Num="1" StepName="MLK" StepText="1) First, we lock PERSONNEL.d

```

```

for read on a reserved RowHash to prevent global deadlock."
StepWD="13" TriggerType="None">
 <StepDetails>
 <MLK>
 <LockOperation LockKind="Proxy" LockLevel="Row" LockSeverity="Read"
NowaitFlag="false" PLLKind="NotPLL">
 <RelationRef AliasName="d" Ref="REL1"/>
 </LockOperation>
 </MLK>
 </StepDetails>
 <AmpStepUsage NumofActiveAMPs="1" QCFampUsage="One" StepRowsReturned="1"
StepStartTime="2015-05-06T19:36:16.47" StepStopTime="2015-05-06T19:36:16.47"/>
 <AmpStepVHFSGUsage/>
 <SchedulerAmpStepUsage AMPOtherWaitTime="0.086" MaxAMPOtherWaitTime="0.086"
MaxOtherWaitTimeAMPNum="3"/>
 <StepDBQL StatementNum="1" StepStartTime="2015-05-06T19:36:16.47"
StepStopTime="2015-05-06T19:36:16.47" TriggerKind="None"/>
</PlanStep>
<PlanStep QCFParallelKind="Sequential" QCFStepKind="LK" QCFStepNum="2"
StepLev1Num="2" StepName="MLK" StepText="2) Next, we lock PERSONNEL.e
for read on a reserved RowHash to prevent global deadlock."
StepWD="13" TriggerType="None">
 <StepDetails>
 <MLK>
 <LockOperation LockKind="Proxy" LockLevel="Row" LockSeverity="Read"
NowaitFlag="false" PLLKind="NotPLL">
 <RelationRef AliasName="e" Ref="REL2"/>
 </LockOperation>
 </MLK>
 </StepDetails>
 <AmpStepUsage NumofActiveAMPs="1" QCFampUsage="One" StepRowsReturned="1"
StepStartTime="2015-05-06T19:36:16.47" StepStopTime="2015-05-06T19:36:16.47"/>
 <AmpStepVHFSGUsage/>
 <SchedulerAmpStepUsage AMPOtherWaitTime="0.528" MaxAMPOtherWaitTime="0.528"
MaxOtherWaitTimeAMPNum="1"/>
 <StepDBQL StatementNum="1" StepStartTime="2015-05-06T19:36:16.47"
StepStopTime="2015-05-06T19:36:16.47" TriggerKind="None"/>
</PlanStep>
<PlanStep QCFParallelKind="Sequential" QCFStepKind="LK" QCFStepNum="3"
StepLev1Num="3" StepName="MLK" StepText="3) We lock PERSONNEL.d for read, and we
lock PERSONNEL.e for read." StepWD="13" TriggerType="None">
 <StepDetails>
 <MLK>
 <LockOperation LockKind="Real" LockLevel="Table" LockSeverity="Read"

```

```

NoWaitFlag="false" PLLKind="NotPLL">
 <RelationRef AliasName="d" Ref="REL1"/>
 </LockOperation>
 <LockOperation LockKind="Real" LockLevel="Table" LockSeverity="Read"
NoWaitFlag="false" PLLKind="NotPLL">
 <RelationRef AliasName="e" Ref="REL2"/>
 </LockOperation>
</MLK>
</StepDetails>
<AmpStepUsage NumofActiveAMPs="4" QCFampUsage="All" StepRowsReturned="4"
StepStartTime="2015-05-06T19:36:16.47" StepStopTime="2015-05-06T19:36:16.47"/>
 <AmpStepVHFSGUsage/>
 <SchedulerAmpStepUsage AMPOtherWaitTime="0.175" MaxAMPOtherWaitTime="0.088"
MaxOtherWaitTimeAMPNum="2"/>
 <StepDBQL StatementNum="1" StepStartTime="2015-05-06T19:36:16.47"
StepStopTime="2015-05-06T19:36:16.47" TriggerKind="None"/>
</PlanStep>
<PlanStep QCFParallelKind="Sequential" QCFStepKind="SR" QCFStepNum="4"
StepLev1Num="4" StepName="RET" StepText="4) We do an all-AMPs RETRIEVE step
from PERSONNEL.e by way of an all-rows scan with a condition of ("(NOT
(PERSONNEL.e.deptno IS NULL)) AND (PERSONNEL.e.yrsexp >= 5)") into Spool 2
(all_amps), which is redistributed by the hash code of (PERSONNEL.e.deptno) to
all AMPs. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2
is estimated with no confidence to be 4 rows (432 bytes). The estimated time for
this step is 0.03 seconds." StepWD="13" TriggerType="None">
 <SourceAccess AccessPosition="1" SyncScanEligible="false">
 <RelationRef AliasName="e" Ref="REL2"/>
 <PPIAccess TotalParts="0"/>
 <CPInfo NumCombinedPartitions="0" NumContexts="0" NumOfColPartsReferences="0"/>
 </SourceAccess>
 <TargetStore Cached="true" GeogInfo="Hash Distributed">
 <SpoolRef Ref="SPOOL2"/>
 <SortKey SortKind="Rowhash"/>
 </TargetStore>
 <Predicate PredicateKind="Source" PredicateText="(NOT (PERSONNEL.e.deptno IS
NULL)) AND (PERSONNEL.e.yrsexp >= 5)">
 <FieldRef Ref="REL2_FLD1027"/>
 <FieldRef Ref="REL2_FLD1029"/>
 </Predicate>
 <OptStepEst EstCPUCost="0.015" EstCPUTime="0.000" EstHRCost="0.000"
EstIOCost="26.637" EstIOTime="0.027" EstNetCost="0.022" EstProcTime="0.027"
EstRowCount="4"/>
 <AmpStepUsage IOCount="60" IOKB="2012.000" MaxAMPSPool="1536"
MaxAmpIO="27" MaxIOAmpNumber="1" MinAMPSPool="0" MinAmpIO="4"

```

```

NumofActiveAMPs="4" QCFampUsage="All" SpoolUsage="3072" StepRowsReturned="3"
StepStartTime="2015-05-06T19:36:16.47" StepStopTime="2015-05-06T19:36:16.47"/>
 <AmpStepVHFSGUsage/>
 <SchedulerAmpStepUsage AMPCPURunDelay="0.002" AMPOtherWaitTime="2.223"
MaxAMPCPURunDelay="0.001" MaxAMPOtherWaitTime="0.626" MaxCPURunDelayAMPNum="2"
MaxOtherWaitTimeAMPNum="2"/>
 <StepDBQL StatementNum="1" StepStartTime="2015-05-06T19:36:16.47"
StepStopTime="2015-05-06T19:36:16.47" TriggerKind="None"/>
</PlanStep>
<PlanStep QCFParallelKind="Sequential" QCFStepKind="MJ" QCFStepNum="5"
StepLev1Num="5" StepName="JIN" StepText="5) We do an all-AMPs JOIN step from
Spool 2 (Last Use) by way of a RowHash match scan, which is joined to
PERSONNEL.d by way of a RowHash match scan. Spool 2 and PERSONNEL.d are
joined using a merge join, with a join condition of ("PERSONNEL.d.deptno =
deptno"). The result goes into Spool 1 (group_amps), which is built locally
on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool
field1 (PERSONNEL.e.salary). The size of Spool 1 is estimated with no confidence
to be 6 rows (312 bytes). The estimated time for this step is 0.11 seconds."
StepWD="13" TriggerType="None">
 <SourceAccess AccessPosition="1" SyncScanEligible="false">
 <SpoolRef Ref="SPOOL2"/>
 <PPIAccess TotalParts="0"/>
 <CPIInfo NumCombinedPartitions="0" NumContexts="0" NumOfColPartsReferences="0"/>
 </SourceAccess>
 <SourceAccess AccessPosition="2" SyncScanEligible="false">
 <RelationRef AliasName="d" Ref="REL1"/>
 <PPIAccess TotalParts="0"/>
 <CPIInfo NumCombinedPartitions="0" NumContexts="0" NumOfColPartsReferences="0"/>
 </SourceAccess>
 <TargetStore Cached="true" GeogInfo="Local">
 <SpoolRef Ref="SPOOL1"/>
 <SortKey SortKind="Field1"/>
 </TargetStore>
 <Predicate PredicateKind="Join" PredicateText="PERSONNEL.d.deptno = deptno">
 <FieldRef Ref="REL1_FLD1025"/>
 <FieldRef Ref="SPOOL2_FLD1027"/>
 </Predicate>
 <StepDetails>
 <JIN JoinKind="Merge Join" JoinType="Inner" LeftOneRowOpt="false"
RightOneRowOpt="false"/>
 </StepDetails>
 <OptStepEst EstCPUCost="0.016" EstCPUTime="0.000" EstHRCost="0.000"
EstIOWCost="106.559" EstIOWTime="0.107" EstNetCost="0.000" EstProcTime="0.107"
EstRowCount="6"/>

```

```

 <AmpStepUsage IOCount="63" IOKB="2147.000" MaxAMPSpool="1024"
MaxAmpIO="51" MaxIOAmpNumber="1" MinAMPSpool="0" MinAmpIO="7"
NumofActiveAMPs="4" QCFampUsage="All" SpoolUsage="2048" StepRowsReturned="3"
StepStartTime="2015-05-06T19:36:16.47" StepStopTime="2015-05-06T19:36:16.48"/>
 <AmpStepVHFSGUsage/>
 <SchedulerAmpStepUsage AMPCPURunDelay="0.006" AMPOtherWaitTime="0.002"
MaxAMPCPURunDelay="0.003" MaxAMPOtherWaitTime="0.001" MaxCPURunDelayAMPNum="3"
MaxOtherWaitTimeAMPNum="2"/>
 <StepDBQL StatementNum="1" StepStartTime="2015-05-06T19:36:16.47"
StepStopTime="2015-05-06T19:36:16.48" TriggerKind="None"/>
 </PlanStep>
 <PlanStep QCFParallelKind="Sequential" QCFStepKind="MS" QCFStepNum="6"
StepLev1Num="6" StepName="Edt" StepText="6) Finally, we send out an END
TRANSACTION step to all AMPs involved in processing the request. -> The contents
of Spool 1 are sent back to the user as the result of statement 1. The total
estimated time is 0.13 seconds." StepWD="13">
 <AmpStepUsage IOCount="36" MaxAMPSpool="1024" MaxAmpIO="26"
MaxIOAmpNumber="1" MinAMPSpool="0" NumofActiveAMPs="4" QCFampUsage="Group"
SpoolUsage="2048" StepRowsReturned="4" StepStartTime="2015-05-06T19:36:16.48"
StepStopTime="2015-05-06T19:36:16.48"/>
 <AmpStepVHFSGUsage/>
 <SchedulerAmpStepUsage AMPOtherWaitTime="0.005" MaxAMPOtherWaitTime="0.002"
MaxOtherWaitTimeAMPNum="3"/>
 <StepDBQL StatementNum="1" StepStartTime="2015-05-06T19:36:16.48"
StepStopTime="2015-05-06T19:36:16.48" TriggerKind="None"/>
 </PlanStep>
 <OptPlanEst EstMaxRowCount="6" EstProcTime="0.133" EstResultRows="6"/>
 <PlanDBQLStats CallNestingLevel="0" DataCollectAlg="1" RequestMode="Exec"
StatementCount="1" TxnMode="BTET"/>
 <AmpPlanUsage AMPCPUTime="0.000" AMPCPUTimeNorm="0.000" DisCPUTimeNorm="0.269"
FirstRespTime="2015-05-06T19:36:16.48" FirstStepTime="2015-05-06T19:36:16.47"
MaxAMPCPUTime="0.000" MaxAMPCPUTimeNorm="0.000" MaxAmpIO="53" MaxIOAmpNumber="1"
MinAMPCPUTime="0.000" MinAMPCPUTimeNorm="0.000" MinAmpIO="9" NumofActiveAMPs="4"
ParserCPUTime="0.024" ParserCPUTimeNorm="1.614" ReqIOKB="4419" SpoolUsage="3072"
TotalIOCount="98"/>
 <AmpPlanVHFSGUsage/>
 <PlanDecayLevel CPUDecayLevel="0" IODecayLevel="0"/>
 <PlanTacticalException TacticalCPUException="0" TacticalIOException="0"/>
 <PEPlanUsage PECPURunDelay="0.008" PEIOWaitTime="0.503" PEIOWaitTimePDE="0.000"
PEOtherWaitTime="0.851" PEOtherWaitTimePDE="0.000" ParserCPUKernelTime="0.004"
ParserCPUKernelTimeNorm="0.000"/>
 <SchedulerAmpPlanUsage AMPCPUKernelTime="0.000" AMPCPUKernelTimeNorm="0.000"
SeqRespTime="0.010"/>
</Plan>

```

```

<Configuration NumAMPs="4" NumNodes="1" NumPEs="2" PEnum="30719"
ReleaseInfo="15.10n.00.42" SystemName="localhost" VersionInfo="15.10n.00.42">
 <TLE CostProfileName="TD15"/>
</Configuration>
<User AcctString="DBC" ExpandAcctString="DBC" UserID="00000504"
UserName="PERSONNEL" ZoneID="00000000"/>
<Session AppID="BTEQ" ClientAddr="153.65.210.211"
ClientID="BS255018" InternalRequestNum="4" KeepFlag="false" LogicalHostID="1"
LogonDateTime="2015-05-06T19:36:13.67" LogonSource="(TCP/IP) d027 153.65.210.211
IE1510 5516 BS255018 BTEQ 01 LSS" NumRequestCtx="1" RequestNum="4"
SessionID="1281" SessionWDID="13"/>
 <WLMgmt FinalWDID="13" OpEnvID="1" ResponseTimeMet="true" SysConID="1"
TDWMEstLastRows="6" TDWMEstMaxRows="6" TDWMEstMemory="4" TDWMEstTotalTime="133"
ThrottleBypass="false" TxnUniq="-378667007" WDID="13"/>
 <ReDrive QueryReDriven="N"/>
</Query>
</QryPlanXML>

```

## Examples of Processing XML Documents Produced by the XMLPLAN Option

These examples show how to use XML data type functions and methods to extract information from XML documents like those produced by the XMLPLAN option for BEGIN QUERY LOGGING and REPLACE QUERY LOGGING.

For more information, see Teradata XML.

---

### Note:

You can also create an XML plan document by using the INSERT EXPLAIN statement with the IN XML option or by using BEGIN QUERY CAPTURE, and then extract information from it.

---

### Using XMLEXTRACT to extract information from an XML plan

This example uses the XMLEXTRACT method to extract query execution plan information from an XML plan. The IPEEligibility attribute indicates whether or not the plan is eligible for Incremental Planning and Execution optimization. In this case, the IPEEligibility value that the query returns indicates that the request is not eligible.

```

select
 createxml(xmltextinfo).xmlextract(
 '//Plan/@IPEEligibility',
 'default=http://schemas.teradata.com/queryplan') as IPEEligibility
from

```



```

 dbc.qrylogv l,
 dbc.qrylogxmlv x
where
 l.queryid = x.queryid and
 l.statementtype = 'Select';
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
IPEEligibility

IPEEligibility="NotEligible"

```

## Using XMLTABLE to Extract Step Details

This example demonstrates how to use the XMLTABLE table function to extract step details, including join information, from an XML document.

```

select
 steplev1num,
 stepname,
 jointype,
 joinkind
from (
select
 l.queryid,
 createxml(xmltextinfo) as plan
from
 dbc.qrylogv l,
 dbc.qrylogxmlv x1
where
 l.queryid = x1.queryid and
 l.statementtype = 'Select') as x2,
xmltable(
xmlnamespaces(default 'http://schemas.teradata.com/queryplan'),
'//PlanStep' passing by value x2.plan
columns
 "QueryID" decimal(18,0) path '../../@QueryID',
 "StepLev1Num" integer path './@StepLev1Num',
 "StepName" char(4) path './@StepName',
 "JoinType" char(10) path './StepDetails/JIN/@JoinType',
 "JoinKind" char(10) path './StepDetails/JIN/@JoinKind')
as t (
 "QueryID",
 "StepLev1Num",
 "StepName",

```



```

 "JoinType",
 "JoinKind")
where x2.queryid = t.queryid
order by t.queryid, steplev1num;
*** Query completed. 6 rows found. 4 columns returned.
*** Total elapsed time was 1 second.
StepLev1Num StepName JoinType JoinKind

 1 MLK ? ?
 2 MLK ? ?
 3 MLK ? ?
 4 RET ? ?
 5 JIN Inner Merge Join
 6 Edt ? ?

```

## Related Information

The following topics and documents are related to the material covered by this section:

- [XMLQCD](#)
- BEGIN QUERY LOGGING and REPLACE QUERY LOGGING in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- INSERT EXPLAIN and EXPLAIN in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146

## Additional Information

### Teradata Links

| Link                                                                                                    | Description                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="https://docs.teradata.com/">https://docs.teradata.com/</a>                                     | Search Teradata Documentation, customize content to your needs, and download PDFs.<br>Customers: Log in to access Orange Books.                                                                                                                            |
| <a href="https://support.teradata.com">https://support.teradata.com</a>                                 | One-stop source for Teradata community support, software downloads, and product information.<br>Log in for customer access to: <ul style="list-style-type: none"><li>• Community support</li><li>• Software updates</li><li>• Knowledge articles</li></ul> |
| <a href="https://www.teradata.com/University/Overview">https://www.teradata.com/University/Overview</a> | Teradata education network                                                                                                                                                                                                                                 |
| <a href="https://support.teradata.com/community">https://support.teradata.com/community</a>             | Link to Teradata community                                                                                                                                                                                                                                 |