Performance Tuning for the Teradata Database

Matthew W Froemsdorf
Teradata Partner Engineering and Technical Consulting

Document Changes

Rev.	Date	Section	Comment
1.0	2010-10-26	All	Initial document

Conventions Used in This Document

The following notational conventions are used in this document:

- Code is shown in Courier New 10 pt: `While True Do`
- Commands are shown in Courier New 11 pt: `Delete`

Trademarks

All trademarks and service marks mentioned in this document are marks of their respective owners and are as such acknowledged by TERADATA Corporation.

Contents

1 Purpose of this Document

The purpose of this document is to instruct the reader how to optimize the performance of a Teradata system through the use of good physical database design principles.

2 Performance Tuning Considerations

In order to understand the concepts behind performance tuning for Teradata, an understanding of some basic Teradata concepts is required. An understanding of the Teradata primary index and table partitioning is required in order to understand Teradata join processing, which is the most common point of performance contention.

2.1 Primary Index

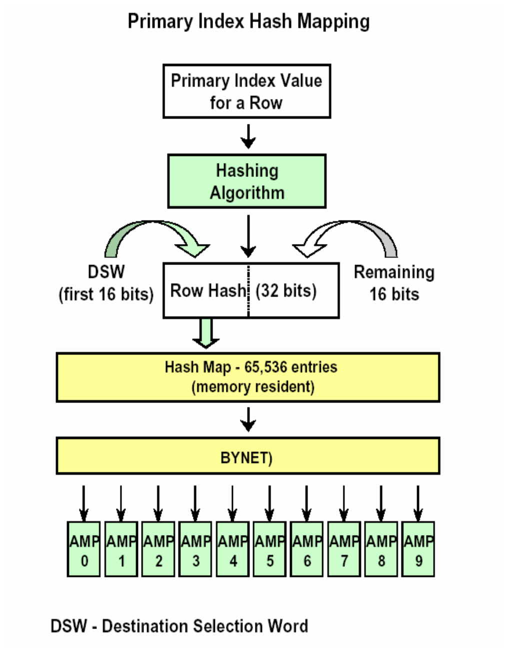
The Teradata primary index is not an index in the traditional sense, as it is not a lookup table. Instead, the primary index is a mechanism that determines where each data row is physically located on the Teradata system. The primary index of a table may be defined as either a single column or as multiple columns. The values of the primary index columns within the table may be unique or non-unique.

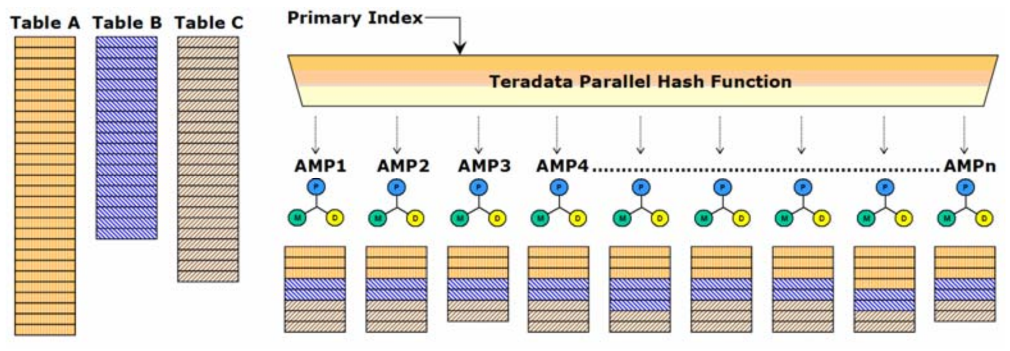
It should be noted that the primary index of a table should not be confused with the primary key of a table. The primary index is a part of the physical database model, and affects the storage and retrieval of data rows. The primary key is a part of the logical database model, and uniquely identifies each record in the table. Often, the primary key of a table is a good candidate for the primary index of a table, particularly for smaller “dimension” or “lookup” tables, but this is not always the case for other tables.

2.1.1 Primary Index Hashing

When rows are inserted into a table, the value of the primary index columns for each row is run through a hashing algorithm to determine the row’s location on the system. The Teradata primary index hashing algorithm is “commutative”, meaning that the order of columns in a multicolumn primary index does not affect the results of the hashing algorithm.

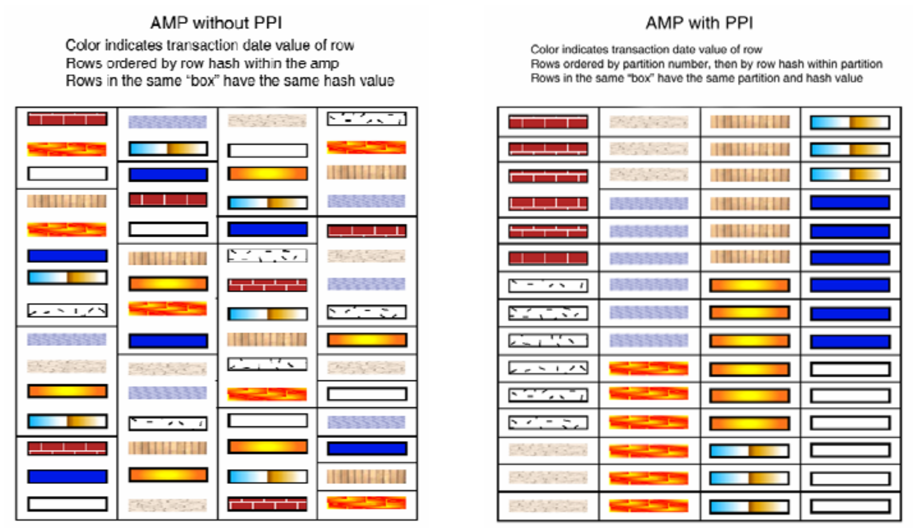
The data rows are distributed based on the first 16 bits of the hash value, which form the row hash. A hash map allocates each of these 65,536 possible values to one of the **Access Module Processors** (AMPs) on the system, such that the hash values are as evenly distributed throughout the system as possible. This ensures that rows with the same primary index value are always stored in the same physical location on the same AMP, and that tables with evenly-distributed primary index values will be evenly physically distributed throughout the Teradata system.


Primary Index Hash Mapping
Primary Index Value for a Row
Hashing Algorithm
DSW (first 16 bits)
Row Hash (32 bits)
Remaining 16 bits
Hash Map - 65,536 entries (memory resident)
BYNET)
AMP 0
AMP 1
AMP 2
AMP 3
AMP 4
AMP 5
AMP 6
AMP 7
AMP 8
AMP 9
DSW - Destination Selection Word



2.1.2 Partitioning

Teradata provides the option to define a partitioning expression in the table definition. The data in a partitioned table is arranged differently from that in a non-partitioned table. Rows are stored on the appropriate AMP based on the hash value of the primary index, and are then placed in a partition based on the partitioning expression, and then ordered by row hash. This allows for rows with the same primary index value in the same partition to be physically grouped together.


AMP without PPI
Color indicates transaction date value of row
Rows ordered by row hash within the amp
Rows in the same "box" have the same hash value
AMP with PPI
Color indicates transaction date value of row
Rows ordered by partition number, then by row hash within partition
Rows in the same "box" have the same partition and hash value

The most common example of partitioning in Teradata is to partition a table on a date column. The partitioning expression may then consist either of individual dates in each partition, or a date range in each partition. Integer values or ranges may also be used in a partitioning expression.

Partitioning can greatly improve the performance of inserts and deletes of entire partitions. Inserting data into an empty partition is extremely fast, because it does not require **transient journaling**. Transient journaling is the process that tracks changed data rows in running queries, so that if an error is encountered during the execution of the query, Teradata can roll back the changes based on the contents of the transient journal. When inserting data into an empty partition, however, the inserts can be rolled back simply by invalidating the partition header of the originally empty partition in the Teradata file system, making transient journaling unnecessary in this case. Deleting an entire partition is even faster, as it does not require transient journaling and does not even require deleting physical rows; it simply requires invalidating the partition header in the Teradata file system.

Partitioning also improves the performance of SELECT statements where a partition value, range of values, or group of values is specified in the WHERE clause. This allows Teradata to avoid a full-table scan of the underlying table, as it instead scans only the relevant partitions.

2.1.3 Primary Index Selection Criteria

There are three main criteria that should be considered when selecting a primary index for a table on Teradata. The primary index should provide a good access path to the data, it should provide good distribution for the data, and the volatility of the data values should be minimal.

2.1.3.1 Access

One criterion for selecting a primary index for a table is to provide the best access path to the data. If individual records are commonly selected from the table, the primary key of the table may be a good choice for the primary index. If the table is typically joined to other tables on a particular column or group of columns, the join columns may be a good candidate for the primary index of the table.

2.1.3.2 Distribution

Another criterion for selecting a primary index for a table is to provide good, even data distribution across the system in order to optimize parallel processing. Evenly-distributed data allows the optimal use of Teradata parallelism for both ETL performance and full-table scan performance. Unique primary indexes will always provide excellent data distribution. Non-unique primary indexes can also provide good data distribution depending on the data.

Note that due to the nature of the Teradata hashing function, a series of consecutive numeric values (e.g. account numbers), if implemented as a `CHAR` or `VARCHAR` data type, will tend to generate a large number of hash collisions that may negatively impact query performance. For this reason, it is suggested that numeric values be implemented as numeric types such as `INTEGER` or `DECIMAL`, which do not encounter this problem.

2.1.3.3 Volatility

Another criterion for selecting a primary index for a table is to minimize volatility of the primary index data values. When the primary index value of a row is changed, this requires moving the entire row to the new destination AMP. Choosing a column with stable data values reduces the overhead of data maintenance.

2.1.3.4 Primary Index Selection Example

As an example of these principles, consider an Order table keyed by the column Order Number, and a Line Item table keyed by the two columns (Order Number, Line Item Sequence Number).

Orders are commonly looked up by Order Number in the Order table, and line items are typically accessed by joining the Order table to the Line Item table on Order Number. Order Numbers are unique in the Order table, and non-unique in the Line Item table. Since Order Number is the key to the Order table Order Number values do not typically change.

In this example, the Order table should use Order Number as its primary index, as this satisfies all of the conditions of Access, Distribution, and Volatility.

As for the Line Item table, the primary key of (Order Number, Line Item Sequence Number) satisfies the conditions of Distribution and Volatility, but not Access, because the compound key of (Order Number, Line Item Sequence Number) will generate a different row hash value than Order Number alone, which implies that using the primary key as the primary index would require the table to be redistributed every time it is joined to the Order table. The Order Number column by itself satisfies the conditions of Access and Volatility, and also satisfies the Distribution condition as long as the typical order contains fewer than one hundred line items. Therefore, Order Number should be selected as the primary index of both the Order and the Line Item tables.

2.2 Join Processing

The shared-nothing architecture of Teradata ensures that each AMP has its own unit of work that it does not share with the other AMPs in the system. Therefore, in order to perform a join, the rows that are going to be joined must both be present on the same AMP. Teradata has several ways of accomplishing this.

2.2.1 Data Movement in Join Processing

When a query is executed on Teradata, the optimizer must first generate a query execution plan, which is a series of independent steps that will be executed on the AMPs. Each AMP works only on its own set of data, and the AMPs do not communicate with each other until each step is completed.

In order to ensure that the rows to be joined are present on the same AMP, the optimizer may specify that table redistribution, table duplication, or table local-AMP sorting is performed on each of the tables in preparation for a join. In each of these cases, a temporary data set, or **spool**, is created that contains a copy of the original table.

2.2.1.1 Table Redistribution

When Teradata performs table redistribution in preparation for a join, the column or columns of the table to be joined are run through the Teradata hashing algorithm. A copy of the table is then placed in a spool which is then redistributed on the Teradata system, so that each row of the spool is placed on its target AMP, as though the join columns were the primary index of the spool.

2.2.1.2 Table Duplication

When Teradata performs table duplication in preparation for a join, a full copy of the table is sent to every AMP, where it resides in spool. This guarantees that every AMP on the system will have a complete copy of the table.

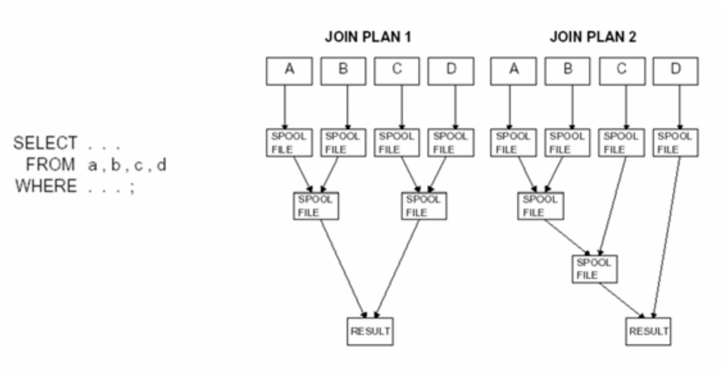
2.2.1.3 Table Local Sorting

When Teradata performs local table sorting in preparation for a join, a spool copy of the table is created which remains on the same AMP, being neither redistributed nor duplicated. However, the spool is then sorted on each AMP by the row hash of the join column(s).

2.2.2 Join Processing Methods

Teradata may use any of several methods to join tables. The most common of these is the merge join, followed by the product join. In certain circumstances, Teradata may perform other, less common joins, such as the **nested join** or the **hash join**.

Many analytical and decision-support queries must be able to join more than two tables. All of the Teradata join approaches are designed to join two tables in a single join. In order to join more than two tables, the query plan generated by the optimizer will determine two tables to join first, placing the results of this join operation into a spool. This spool may then be joined either to another table or to another spool, and that result to another table or spool, and so forth until the entire query is resolved.

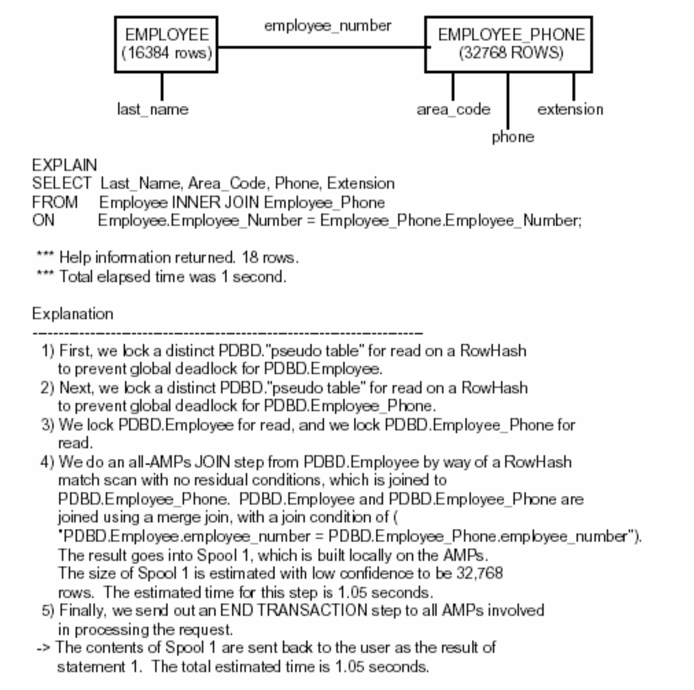


2.2.2.1 Merge Join

The **merge join** is the most common method of joining data sets on Teradata. In a merge join, each AMP compares rows from one data set that has been sorted by hash value with rows from another data set that has been sorted by hash value. Since both data sets are sorted by hash value, the total number of comparisons operations that need to be performed by the join operation is minimized.

2.2.2.1.1 In-Place Merge Join

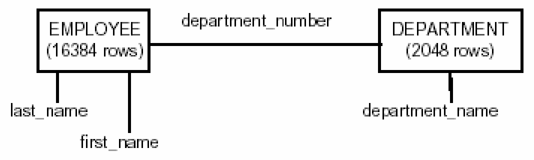
When two tables are joined, and the join criteria specify the primary index of the first table being equal to the primary index of the second table, a merge join can be performed in-place. This requires no redistribution, no duplication, and no sorting of either table. This is almost always the most efficient join type possible on Teradata.



```
EXPLAIN
SELECT  Last_Name, Area_Code, Phone, Extension
FROM    Employee INNER JOIN Employee_Phone
ON      Employee.Employee_Number = Employee_Phone.Employee_Number;

 *** Help information returned. 18 rows.
 *** Total elapsed time was 1 second.

Explanation
---------------------------------------------------------------------------
  1) First, we lock a distinct PDBD."pseudo table" for read on a RowHash
     to prevent global deadlock for PDBD.Employee.
  2) Next, we lock a distinct PDBD."pseudo table" for read on a RowHash
     to prevent global deadlock for PDBD.Employee_Phone.
  3) We lock PDBD.Employee for read, and we lock PDBD.Employee_Phone for
     read.
  4) We do an all-AMPs JOIN step from PDBD.Employee by way of a RowHash
     match scan with no residual conditions, which is joined to
     PDBD.Employee_Phone.  PDBD.Employee and PDBD.Employee_Phone are
     joined using a merge join, with a join condition of (
     "PDBD.Employee.employee_number = PDBD.Employee_Phone.employee_number").
     The result goes into Spool 1, which is built locally on the AMPs.
     The size of Spool 1 is estimated with low confidence to be 32,768
     rows.  The estimated time for this step is 1.05 seconds.
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
 -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 1.05 seconds.
```

2.2.2.1.2 Merge Join with One Table Redistribution

When two tables are joined, and the join criteria specify the primary index of one of the tables being equal to a column or columns of the second table which do not comprise the second table's primary index, a merge join may be performed after redistributing the second table. In this case, the second table must be placed in spool and then re-hashed on the join columns.



```
EXPLAIN
SELECT  Department_Name, Last_Name, First_Name
FROM    Department INNER JOIN Employee
ON      Employee.Department_Number = Department.Department_Number;


 *** Help information returned. 23 rows.
 *** Total elapsed time was 1 second.

Explanation
---------------------------------------------------------------------------
  1) First, we lock a distinct PDBD."pseudo table" for read on a RowHash
     to prevent global deadlock for PDBD.Employee.
  2) Next, we lock a distinct PDBD."pseudo table" for read on a RowHash
     to prevent global deadlock for PDBD.Department.
  3) We lock PDBD.Employee for read, and we lock PDBD.Department for read.
  4) We do an all-AMPs RETRIEVE step from PDBD.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 with
     high confidence to be 16,384 rows.  The estimated time for this
     step is 0.68 seconds.
  5) We do an all-AMPs JOIN step from PDBD.Department by way of a
     RowHash match scan with no residual conditions, which is joined to
     Spool 2 (Last Use).  PDBD.Department and Spool 2 are joined using a
     merge join, with a join condition of ("department_number =
     PDBD.Department.department_number").  The result goes into Spool 1,
     which is built locally on the AMPs.  The size of Spool 1 is
     estimated with low confidence to be 16,384 rows.  The estimated
     time for this step is 0.35 seconds.
  6) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 1.03 seconds.
```

2.2.2.1.3 Merge Join with Two Table Redistribution

When two tables are joined, and the join criteria specify non-PI columns of the first table being equal to non-PI columns of the second table, a merge join may be performed after redistributing both tables. In this case, both tables will be placed in spool and then re-hashed on the join columns.

2.2.2.1.4 Merge Join with Duplication

When two tables are joined with an equality condition, and it is not a PI-to-PI join, a merge join with duplication may be performed. In this case, the smaller table is duplicated across all AMPs and then sorted by the row hash of the join column, and the larger table remains on its current AMP but is locally sorted by the row hash of the join column.

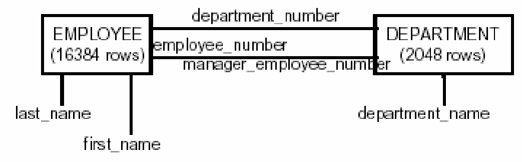
The Teradata optimizer will typically choose this approach when it is less resource-intensive to duplicate and sort the smaller table than it is to hash and redistribute the larger table.

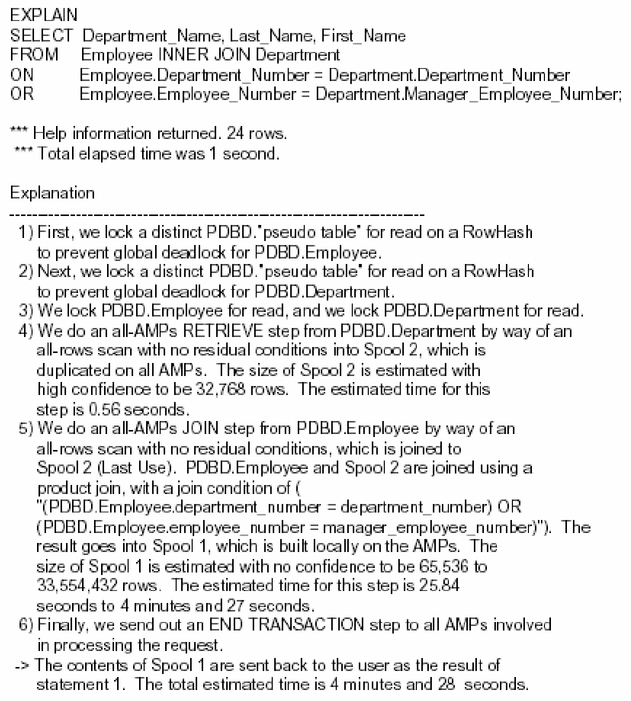
2.2.2.2 Product Join

The **product join** is another method of joining tables on Teradata. In a product join, each AMP compares rows from one unsorted data set with rows from another unsorted data set. Since neither of the data sets is sorted, every row in the first data set must be compared with every row from the second data set, unlike in a merge join.

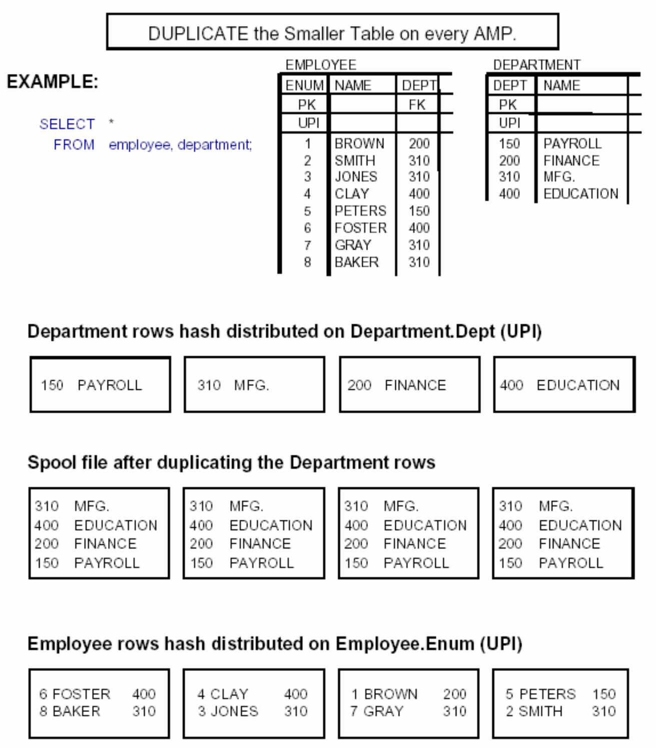
When two tables are joined on a non-equivalence condition, or when an extremely large table is joined to an extremely small table, a product join may be performed. To perform a product join, the smaller of the two data sets must always be duplicated across all AMPs so that the larger data set may remain in place.

The product join is often the best way of joining an extremely small table with an extremely large table, since the large table will neither have to be redistributed nor sorted. However, it is usually the worst possible way to join two large tables, as duplicating a large table in spool is an extremely expensive operation in terms of system resources.





```
EXPLAIN
SELECT  Department_Name, Last_Name, First_Name
FROM    Employee INNER JOIN Department
ON      Employee.Department_Number = Department.Department_Number
OR      Employee.Employee_Number = Department.Manager_Employee_Number;

*** Help information returned. 24 rows.
 *** Total elapsed time was 1 second.

Explanation
---------------------------------------------------------------------------
  1) First, we lock a distinct PDBD."pseudo table" for read on a RowHash
     to prevent global deadlock for PDBD.Employee.
  2) Next, we lock a distinct PDBD."pseudo table" for read on a RowHash
     to prevent global deadlock for PDBD.Department.
  3) We lock PDBD.Employee for read, and we lock PDBD.Department for read.
  4) We do an all-AMPs RETRIEVE step from PDBD.Department 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 with
     high confidence to be 32,768 rows.  The estimated time for this
     step is 0.56 seconds.
  5) We do an all-AMPs JOIN step from PDBD.Employee by way of an
     all-rows scan with no residual conditions, which is joined to
     Spool 2 (Last Use).  PDBD.Employee and Spool 2 are joined using a
     product join, with a join condition of (
     "(PDBD.Employee.department_number = department_number) OR
     (PDBD.Employee.employee_number = manager_employee_number)").  The
     result goes into Spool 1, which is built locally on the AMPs.  The
     size of Spool 1 is estimated with no confidence to be 65,536 to
     33,554,432 rows.  The estimated time for this step is 25.84
     seconds to 4 minutes and 27 seconds.
  6) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
 -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 4 minutes and 28  seconds.
```



DUPLICATE the Smaller Table on every AMP.

EXAMPLE:

```
SELECT  *
  FROM  employee, department;
```

EMPLOYEE

ENUM	NAME	DEPT
PK		FK
UPI		
1	BROWN	200
2	SMITH	310
3	JONES	310
4	CLAY	400
5	PETERS	150
6	FOSTER	400
7	GRAY	310
8	BAKER	310

DEPARTMENT

DEPT	NAME
PK	
UPI	
150	PAYROLL
200	FINANCE
310	MFG.
400	EDUCATION

Department rows hash distributed on Department.Dept (UPI)

150 PAYROLL	310 MFG.	200 FINANCE	400 EDUCATION

Spool file after duplicating the Department rows

310 MFG.	310 MFG.	310 MFG.	310 MFG.
400 EDUCATION	400 EDUCATION	400 EDUCATION	400 EDUCATION
200 FINANCE	200 FINANCE	200 FINANCE	200 FINANCE
150 PAYROLL	150 PAYROLL	150 PAYROLL	150 PAYROLL

Employee rows hash distributed on Employee.Enum (UPI)

6 FOSTER 400	4 CLAY 400	1 BROWN 200	5 PETERS 150
8 BAKER 310	3 JONES 310	7 GRAY 310	2 SMITH 310

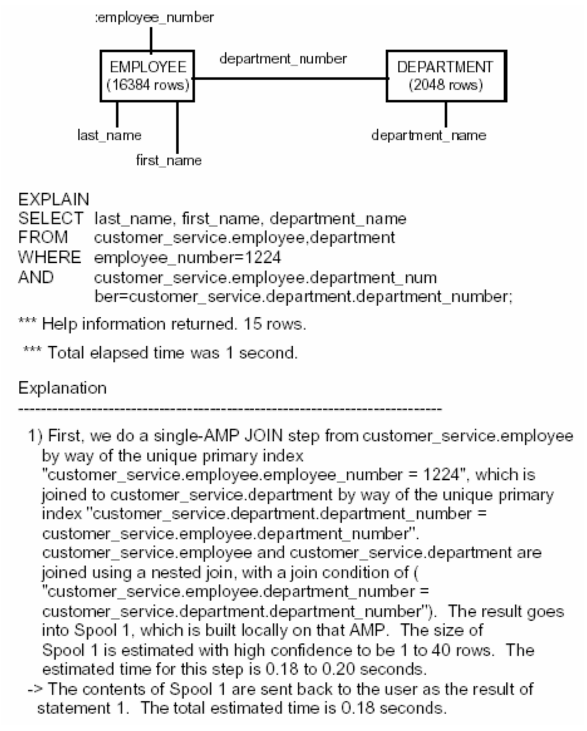
2.2.2.3 Other Join Types

Two other join types supported by Teradata are the nested join and the hash join.

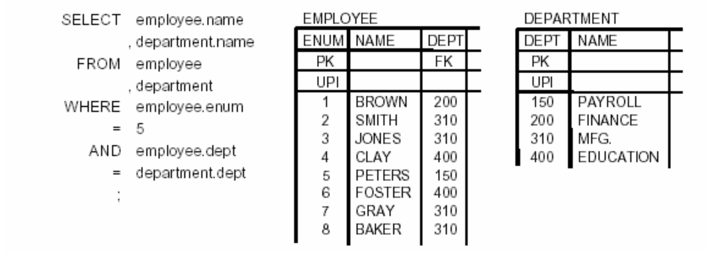
2.2.2.3.1 Nested Join

When a single value is specified for a unique index column of a table, and the join criteria includes a column or columns of this first table being equal to any index of the second table, a **nested join** may sometimes be performed. In a nested join, Teradata uses the single row from the first table, and uses the hash of the join column in that row in order to access matching rows from the second table. The nested join is an all-AMPs operation when the index used in the second table is a NUSI. The nested join is a few-AMPs join when the join is on a UPI, NUPI, or USI of the second table.

The nested join is generally a very efficient operation on Teradata.



```
EXPLAIN
SELECT  last_name, first_name, department_name
FROM    customer_service.employee,department
WHERE   employee_number=1224
AND     customer_service.employee.department_num
        ber=customer_service.department.department_number;

*** Help information returned. 15 rows.
 *** Total elapsed time was 1 second.

Explanation
---------------------------------------------------------------------------
  1) First, we do a single-AMP JOIN step from customer_service.employee
     by way of the unique primary index
     "customer_service.employee.employee_number = 1224", which is
     joined to customer_service.department by way of the unique primary
     index "customer_service.department.department_number =
     customer_service.employee.department_number".
     customer_service.employee and customer_service.department are
     joined using a nested join, with a join condition of (
     "customer_service.employee.department_number =
     customer_service.department.department_number"). The result goes
     into Spool 1, which is built locally on that AMP. The size of
     Spool 1 is estimated with high confidence to be 1 to 40 rows. The
     estimated time for this step is 0.18 to 0.20 seconds.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1. The total estimated time is 0.18 seconds.
```



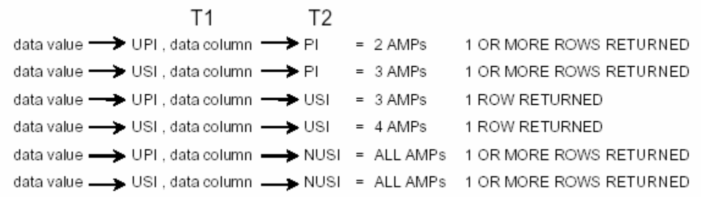
```
SELECT   employee.name
       , department.name
  FROM   employee
       , department
 WHERE   employee.enum
     =   5
   AND   employee.dept
     =   department.dept
     ;
```

EMPLOYEE

ENUM	NAME	DEPT
PK		FK
UPI		
1	BROWN	200
2	SMITH	310
3	JONES	310
4	CLAY	400
5	PETERS	150
6	FOSTER	400
7	GRAY	310
8	BAKER	310

DEPARTMENT

DEPT	NAME
PK	
UPI	
150	PAYROLL
200	FINANCE
310	MFG.
400	EDUCATION

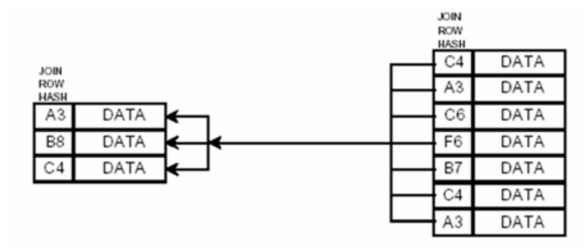


	T1	T2		
data value →	UPI , data column →	PI	= 2 AMPs	1 OR MORE ROWS RETURNED
data value →	USI , data column →	PI	= 3 AMPs	1 OR MORE ROWS RETURNED
data value →	UPI , data column →	USI	= 3 AMPs	1 ROW RETURNED
data value →	USI , data column →	USI	= 4 AMPs	1 ROW RETURNED
data value →	UPI , data column →	NUSI	= ALL AMPs	1 OR MORE ROWS RETURNED
data value →	USI , data column →	NUSI	= ALL AMPs	1 OR MORE ROWS RETURNED

2.2.2.3.2 Hash Join

The **hash join** is an alternative join scheme to the merge join with duplication. In a hash join, the smaller table is duplicated on all AMPs, and then sorted in row hash sequence. The larger table remains in place; however, unlike in the merge join with duplication, the larger table does not need to be sorted in order to perform a hash join. The join column row hash of the larger table is then used to search the “hash map” of the smaller table.

A hash join can be more efficient than a merge join when the hash map of the smaller table can remain in memory on all AMPs, but it becomes less efficient when the hash map of the smaller table is too big, or when the underlying data is badly skewed.



2.3 Join Indexes

On Teradata, a **Join Index** is a data structure that contains data from one or more tables, with or without aggregation, and with or without filtering. It may contain columns from either a single table or from multiple tables.

Join indexes are implemented like physical tables on Teradata. Each join index has its own primary index, and may have secondary indexes defined as well. Join indexes cannot be accessed directly by users; instead, the data in a join index is automatically maintained by Teradata as the data in the underlying tables changes, and the Teradata optimizer will decide when to use a join index to resolve a query if it provides a more efficient path than selecting from the underlying tables.

A single-table join index is more likely to be useful when it **covers** a query, meaning that all columns of the underlying table needed by the query are present in the join index. One reason to define a covering single-table join index would be to specify a different primary index for the join index, in order to provide an alternate access path to the underlying data. In this example, a table may have its primary key defined as its primary index, and a join index may then be defined on the same table with a foreign key defined as the join index's primary index. Another reason to define a covering single-table join index is the **sparse join index** example described later in this document.

Multi-table join indexes can be useful even when they do not cover a query, although a covering join index always gives the greatest performance benefit.

Although join indexes provide several performance benefits, there are some drawbacks to their implementation. They require the use of additional permanent space on Teradata. They require additional processing, as each time a row in one of the underlying tables is inserted, updated, or deleted, the join index must be updated to reflect the change. Join indexes may not be defined on a table with triggers. Join indexes are not compatible with the use of the Fastload and Multiload utilities (or these utilities' implementation in Teradata Parallel Transporter) on the underlying tables.

2.3.1 Aggregate Join Index

An **aggregate join index** allows the user to specify aggregates in the join index. Aggregate Join Indexes provide a tremendous performance benefit for queries that frequently specify the same aggregate operation on the same column or columns.

2.3.2 Sparse Join Index

A **Sparse Join Index** is a special type of single table join index, in which a WHERE filter is placed on the table to reduce the number of rows that qualify for the index. Sparse Join Indexes can be very beneficial when defined on a large table, in cases where a small subset of the table needs to be accessed frequently, as it allows Teradata to work with a much smaller data set.

2.4 Statistics

Teradata statistics are data demographics that may be used by the optimizer to create a query execution plan. By issuing a `COLLECT STATISTICS` command, Teradata will analyze the data distribution of the specified column or group of columns of the specified table, and store these statistics in the data dictionary. The statistics will then be used by the optimizer in order to determine the best possible query execution plan.

2.4.1 When to Collect Statistics

As a general rule, it is recommended to collect statistics on the following.

- All non-unique primary indexes of all tables
- All unique primary indexes of small tables (where the number of rows is less than 100 times the number of AMPs on the system)
- Unique primary or secondary indexes that are commonly used in range constraints
- All sets of columns used in join conditions for all tables
- All non-unique secondary indexes with an uneven distribution of values

Note that collection and maintenance of statistics requires significant use of system resources for larger tables. For this reason, statistics should not be collected on all columns of all tables without regard to how the statistics would benefit query performance.

2.4.2 How to Maintain Collected Statistics

Teradata statistics need to be maintained as the underlying tables change. In some cases, having stale statistics may be worse than having no statistics at all. As an example, a sales table which is loaded daily may have a Sale Date column. If database statistics on this column are over a week old, the optimizer may assume that there are no rows with a Sale Date for the current week, which may generate problematic query plans if a report for the current week is run.

Generally, statistics should be re-collected when 10% of the rows of the underlying table have changed, and should be re-collected daily on date columns of fact tables.

2.4.3 Dynamic AMP Sampling

When statistics are unavailable, the optimizer will rely on Dynamic AMP Sampling, a process in which Teradata analyzes a small set of data from a random AMP in the system, from which it will extrapolate a set of statistics. This process is very accurate when sampling unique primary indexes of extremely large tables. However, Dynamic AMP Sampling is often unreliable for very small tables, and can give inconsistent results for data that is not evenly distributed. For this reason, it is particularly important to maintain statistics on columns or indexes with skewed or lumpy data distributions, as Teradata statistics can easily account for these kinds of distributions.

3 Performance Monitoring and Tuning

Some of the most effective ways to monitor and tune Teradata for performance involve the use of the Teradata Explain Facility, the PMON Utility, and the Database Query Log

3.1 The Explain Facility

When a query is prefixed with the `EXPLAIN` keyword, Teradata does not return the query results, but instead returns the explain plan generated for the query by the optimizer. The explain plan is a readable English-language explanation of the steps to be performed by the AMPs to resolve the query, and includes the following information.

- The estimated size of data sets
- Confidence levels of data size estimates (based on the presence or absence of Teradata statistics)
- Estimated processing times
- Locks placed on database objects
- Join techniques used to resolve the query
- The order in which tables are joined and aggregations are performed
- Whether rows are accessed from source tables by primary index value, by secondary index lookup, by partition, or by full-table scan

Several examples of explain plans can be found in section 2.2 of this document (Join Processing).

3.2 The PMON Utility

Teradata **PMON**, or Teradata Performance Monitor, is a client utility that allows database administrators and privileged users to monitor active sessions on Teradata. With PMON, a user may view the SQL and optimizer query plan of any active session in near real-time, as well as additional details such as the current step of the query plan, the execution time of each completed step, and the number of rows processed during each completed step.

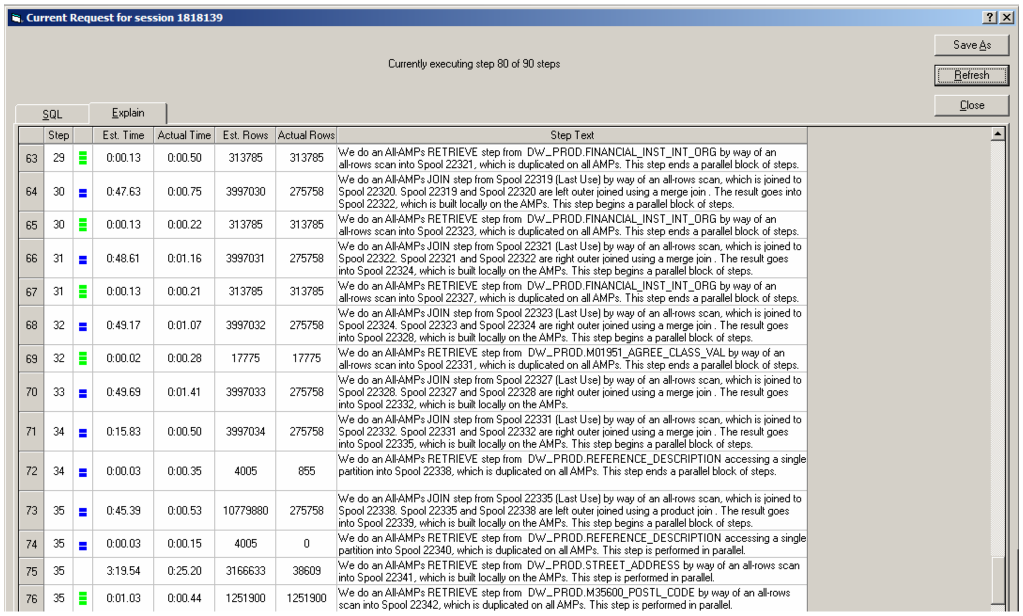
These features of PMON allow users to identify during runtime which steps in a query are troublesome. Long-running steps, unnecessary product joins, and bad spool size estimates made by the optimizer are readily identifiable, and skewed processing is much easier to detect.

3.2.1 Performance Tuning Examples Using PMON

Suppose that a database administrator notices that selecting from the view V01_INFO, a view which joins a large number of tables, seems to encounter performance problems.

To research this, the DBA will start from the PMON **Session** screen. From there, the DBA should identify the session running the query and click on it. From there, click the SQL button to see the session's SQL. When the SQL appears, click the Explain tab to see the explain plan.

In this case, PMON shows that the query runs well until the last few steps, at which point each step begins to take longer than anticipated, and the Estimated Rows appear to be very different from the Actual Rows in each spool. The problems appear to begin when the optimizer makes an inaccurate, no-confidence estimate of the size of a spool retrieved from the STREET_ADDRESS table in Step 35.



Currently executing step 80 of 90 steps

	Step		Est. Time	Actual Time	Est. Rows	Actual Rows	Step Text
63	29	🟩	0:00.13	0:00.50	313785	313785	We do an All-AMPs RETRIEVE step from DW_PROD.FINANCIAL_INST_INT_ORG by way of an all-rows scan into Spool 22321, which is duplicated on all AMPs. This step ends a parallel block of steps.
64	30	🟦	0:47.63	0:00.75	3997030	275758	We do an All-AMPs JOIN step from Spool 22319 (Last Use) by way of an all-rows scan, which is joined to Spool 22320. Spool 22319 and Spool 22320 are left outer joined using a merge join . The result goes into Spool 22322, which is built locally on the AMPs. This step begins a parallel block of steps.
65	30	🟩	0:00.13	0:00.22	313785	313785	We do an All-AMPs RETRIEVE step from DW_PROD.FINANCIAL_INST_INT_ORG by way of an all-rows scan into Spool 22323, which is duplicated on all AMPs. This step ends a parallel block of steps.
66	31	🟦	0:48.61	0:01.16	3997031	275758	We do an All-AMPs JOIN step from Spool 22321 (Last Use) by way of an all-rows scan, which is joined to Spool 22322. Spool 22321 and Spool 22322 are right outer joined using a merge join . The result goes into Spool 22324, which is built locally on the AMPs. This step begins a parallel block of steps.
67	31	🟩	0:00.13	0:00.21	313785	313785	We do an All-AMPs RETRIEVE step from DW_PROD.FINANCIAL_INST_INT_ORG by way of an all-rows scan into Spool 22327, which is duplicated on all AMPs. This step ends a parallel block of steps.
68	32	🟦	0:49.17	0:01.07	3997032	275758	We do an All-AMPs JOIN step from Spool 22323 (Last Use) by way of an all-rows scan, which is joined to Spool 22324. Spool 22323 and Spool 22324 are right outer joined using a merge join . The result goes into Spool 22328, which is built locally on the AMPs. This step begins a parallel block of steps.
69	32	🟩	0:00.02	0:00.28	17775	17775	We do an All-AMPs RETRIEVE step from DW_PROD.M01951_AGREE_CLASS_VAL by way of an all-rows scan into Spool 22331, which is duplicated on all AMPs. This step ends a parallel block of steps.
70	33	🟦	0:49.69	0:01.41	3997033	275758	We do an All-AMPs JOIN step from Spool 22327 (Last Use) by way of an all-rows scan, which is joined to Spool 22328. Spool 22327 and Spool 22328 are right outer joined using a merge join . The result goes into Spool 22332, which is built locally on the AMPs.
71	34	🟦	0:15.83	0:00.50	3997034	275758	We do an All-AMPs JOIN step from Spool 22331 (Last Use) by way of an all-rows scan, which is joined to Spool 22332. Spool 22331 and Spool 22332 are right outer joined using a merge join . The result goes into Spool 22335, which is built locally on the AMPs. This step begins a parallel block of steps.
72	34	🟦	0:00.03	0:00.35	4005	855	We do an All-AMPs RETRIEVE step from DW_PROD.REFERENCE_DESCRIPTION accessing a single partition into Spool 22338, which is duplicated on all AMPs. This step ends a parallel block of steps.
73	35	🟦	0:45.39	0:00.53	10779880	275758	We do an All-AMPs JOIN step from Spool 22335 (Last Use) by way of an all-rows scan, which is joined to Spool 22338. Spool 22335 and Spool 22338 are left outer joined using a product join . The result goes into Spool 22339, which is built locally on the AMPs. This step begins a parallel block of steps.
74	35	🟦	0:00.03	0:00.15	4005	0	We do an All-AMPs RETRIEVE step from DW_PROD.REFERENCE_DESCRIPTION accessing a single partition into Spool 22340, which is duplicated on all AMPs. This step is performed in parallel.
75	35		3:19.54	0:25.20	3166633	38609	We do an All-AMPs RETRIEVE step from DW_PROD.STREET_ADDRESS by way of an all-rows scan into Spool 22341, which is built locally on the AMPs. This step is performed in parallel.
76	35	🟩	0:01.03	0:00.44	1251900	1251900	We do an All-AMPs RETRIEVE step from DW_PROD.M35600_POSTL_CODE by way of an all-rows scan into Spool 22342, which is duplicated on all AMPs. This step is performed in parallel.

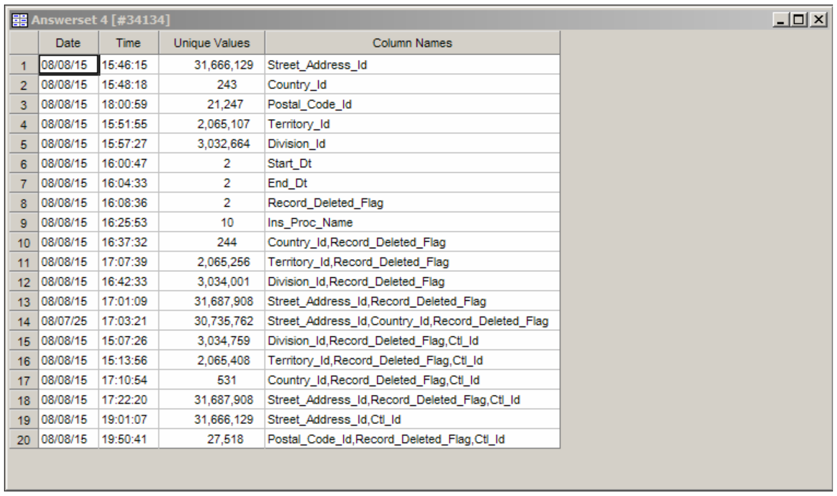
To determine how the STREET_ADDRESS table is accessed by this view, a SHOW VIEW statement is run, which returns the following.

```
REPLACE VIEW DW_PROD.V01_INFO
AS LOCKING ROW FOR ACCESS
SELECT
```

```
...

FROM DW_PROD.AGREEMENT AS AGREEMENT

INNER JOIN DW_PROD.VRSK_BUSINESSDATE_M AS BD
ON BD.BUSINESSDATE BETWEEN AGREEMENT.START_DT AND AGREEMENT.END_DT
AND AGREEMENT.RECORD_DELETED_FLAG = 0
AND AGREEMENT.ACCOUNT_MODIFIER_NUM = 'AM14'
AND AGREEMENT.CTL_ID = '004'

...

LEFT OUTER JOIN DW_PROD.AGREEMENT_TO_LOCATION AS AGREEMENT_TO_LOCATION
ON AGREEMENT.Account_Num = AGREEMENT_TO_LOCATION.Account_Num
AND AGREEMENT.Account_Modifier_Num =
AGREEMENT_TO_LOCATION.Account_Modifier_Num
AND BD.BUSINESSDATE BETWEEN AGREEMENT_TO_LOCATION.START_DT AND
AGREEMENT_TO_LOCATION.END_DT
AND AGREEMENT_TO_LOCATION.RECORD_DELETED_FLAG = 0
AND AGREEMENT_TO_LOCATION.CTL_ID = '004'

LEFT OUTER JOIN DW_PROD.STREET_ADDRESS AS STREET_ADDRESS
ON AGREEMENT_TO_LOCATION.Location_ID = STREET_ADDRESS.STREET_ADDRESS_ID
AND BD.BUSINESSDATE BETWEEN STREET_ADDRESS.START_DT AND
STREET_ADDRESS.END_DT
AND STREET_ADDRESS.RECORD_DELETED_FLAG = 0
AND STREET_ADDRESS.CTL_ID = '004';
```

In this case, the view filters the STREET_ADDRESS table on the Ctl_Id and Record_Deleted_Flag fields. To see whether the necessary statistics are present on STREET_ADDRESS, the following SQL command is run.

```
HELP STATISTICS DW_PROD.STREET_ADDRESS;
```

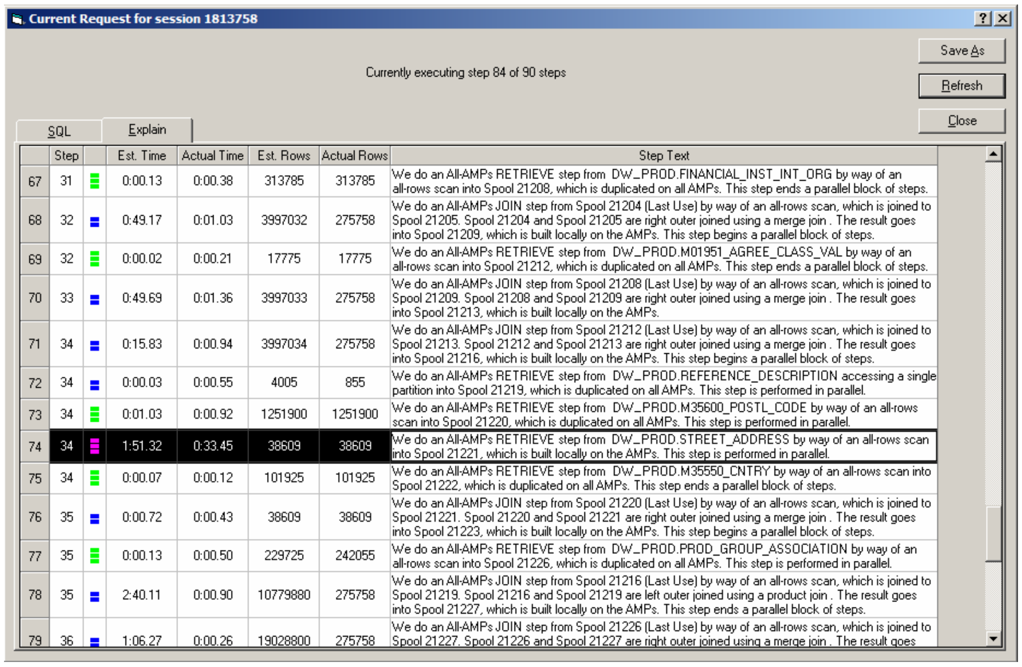


	Date	Time	Unique Values	Column Names
1	08/08/15	15:46:15	31,666,129	Street_Address_Id
2	08/08/15	15:48:18	243	Country_Id
3	08/08/15	18:00:59	21,247	Postal_Code_Id
4	08/08/15	15:51:55	2,065,107	Territory_Id
5	08/08/15	15:57:27	3,032,664	Division_Id
6	08/08/15	16:00:47	2	Start_Dt
7	08/08/15	16:04:33	2	End_Dt
8	08/08/15	16:08:36	2	Record_Deleted_Flag
9	08/08/15	16:25:53	10	Ins_Proc_Name
10	08/08/15	16:37:32	244	Country_Id,Record_Deleted_Flag
11	08/08/15	17:07:39	2,065,256	Territory_Id,Record_Deleted_Flag
12	08/08/15	16:42:33	3,034,001	Division_Id,Record_Deleted_Flag
13	08/08/15	17:01:09	31,687,908	Street_Address_Id,Record_Deleted_Flag
14	08/07/25	17:03:21	30,735,762	Street_Address_Id,Country_Id,Record_Deleted_Flag
15	08/08/15	15:07:26	3,034,759	Division_Id,Record_Deleted_Flag,Ctl_Id
16	08/08/15	15:13:56	2,065,408	Territory_Id,Record_Deleted_Flag,Ctl_Id
17	08/08/15	17:10:54	531	Country_Id,Record_Deleted_Flag,Ctl_Id
18	08/08/15	17:22:20	31,687,908	Street_Address_Id,Record_Deleted_Flag,Ctl_Id
19	08/08/15	19:01:07	31,666,129	Street_Address_Id,Ctl_Id
20	08/08/15	19:50:41	27,518	Postal_Code_Id,Record_Deleted_Flag,Ctl_Id

The HELP STATISTICS command shows that statistics are present on Record_Deleted_Flag, but not on Ctl_Id. It is not sufficient for statistics to be present on only one of the columns; it would be preferable to have statistics present on each column, and best of all to have statistics present on the two columns together. To accomplish this, the following SQL commands are run.

```
COLLECT STATISTICS ON DW_PROD.STREET_ADDRESS
COLUMN (Ctl_Id);

COLLECT STATISTICS ON DW_PROD.STREET_ADDRESS
COLUMN (Ctl_Id, Record_Deleted_Flag);
```

After this is complete, the original query is run once again, and monitored in PMON to determine whether this helps the performance of the query. PMON shows that Teradata now has high confidence in the estimated size of the spool retrieved from the STREET_ADDRESS table, and the subsequent steps of the query run much faster and do not encounter performance issues.



Currently executing step 84 of 90 steps

	Step		Est. Time	Actual Time	Est. Rows	Actual Rows	Step Text
67	31	🟩	0:00.13	0:00.38	313785	313785	We do an All-AMPs RETRIEVE step from DW_PROD.FINANCIAL_INST_INT_ORG by way of an all-rows scan into Spool 21208, which is duplicated on all AMPs. This step ends a parallel block of steps.
68	32	🟦	0:49.17	0:01.03	3997032	275758	We do an All-AMPs JOIN step from Spool 21204 (Last Use) by way of an all-rows scan, which is joined to Spool 21205. Spool 21204 and Spool 21205 are right outer joined using a merge join . The result goes into Spool 21209, which is built locally on the AMPs. This step begins a parallel block of steps.
69	32	🟩	0:00.02	0:00.21	17775	17775	We do an All-AMPs RETRIEVE step from DW_PROD.M01951_AGREE_CLASS_VAL by way of an all-rows scan into Spool 21212, which is duplicated on all AMPs. This step ends a parallel block of steps.
70	33	🟦	0:49.69	0:01.36	3997033	275758	We do an All-AMPs JOIN step from Spool 21208 (Last Use) by way of an all-rows scan, which is joined to Spool 21209. Spool 21208 and Spool 21209 are right outer joined using a merge join . The result goes into Spool 21213, which is built locally on the AMPs.
71	34	🟦	0:15.83	0:00.94	3997034	275758	We do an All-AMPs JOIN step from Spool 21212 (Last Use) by way of an all-rows scan, which is joined to Spool 21213. Spool 21212 and Spool 21213 are right outer joined using a merge join . The result goes into Spool 21216, which is built locally on the AMPs. This step begins a parallel block of steps.
72	34	🟦	0:00.03	0:00.55	4005	855	We do an All-AMPs RETRIEVE step from DW_PROD.REFERENCE_DESCRIPTION accessing a single partition into Spool 21219, which is duplicated on all AMPs. This step is performed in parallel.
73	34	🟩	0:01.03	0:00.92	1251900	1251900	We do an All-AMPs RETRIEVE step from DW_PROD.M35600_POSTL_CODE by way of an all-rows scan into Spool 21220, which is duplicated on all AMPs. This step is performed in parallel.
74	34	🟪	1:51.32	0:33.45	38609	38609	We do an All-AMPs RETRIEVE step from DW_PROD.STREET_ADDRESS by way of an all-rows scan into Spool 21221, which is built locally on the AMPs. This step is performed in parallel.
75	34	🟩	0:00.07	0:00.12	101925	101925	We do an All-AMPs RETRIEVE step from DW_PROD.M35550_CNTRY by way of an all-rows scan into Spool 21222, which is duplicated on all AMPs. This step ends a parallel block of steps.
76	35	🟦	0:00.72	0:00.43	38609	38609	We do an All-AMPs JOIN step from Spool 21220 (Last Use) by way of an all-rows scan, which is joined to Spool 21221. Spool 21220 and Spool 21221 are right outer joined using a merge join . The result goes into Spool 21223, which is built locally on the AMPs. This step begins a parallel block of steps.
77	35	🟩	0:00.13	0:00.50	229725	242055	We do an All-AMPs RETRIEVE step from DW_PROD.PROD_GROUP_ASSOCIATION by way of an all-rows scan into Spool 21226, which is duplicated on all AMPs. This step is performed in parallel.
78	35	🟦	2:40.11	0:00.90	10779880	275758	We do an All-AMPs JOIN step from Spool 21216 (Last Use) by way of an all-rows scan, which is joined to Spool 21219. Spool 21216 and Spool 21219 are left outer joined using a product join . The result goes into Spool 21227, which is built locally on the AMPs. This step ends a parallel block of steps.
79	36	🟦	1:06.27	0:00.26	19028800	275758	We do an All-AMPs JOIN step from Spool 21226 (Last Use) by way of an all-rows scan, which is joined to Spool 21227. Spool 21226 and Spool 21227 are right outer joined using a merge join . The result goes

Note that the SQL in the view remains unchanged, and the physical design of the database remains unchanged. The only change that was made was collecting statistics on one of the tables referenced by the view.

3.3 Teradata Database Query Log

DBQL, or Database Query Log, is a Teradata feature that is not enabled by default, which can be very useful for researching performance issues. DBQL will log attributes about each query executed on the system, such as the user who submitted the query, the leading few characters of the SQL text, the time the query was received by the database, the number of steps completed, the time the first step began, and the time the answer set was returned to the client. Depending on which features of DBQL are enabled, the logging may also include the complete SQL text of each query, its entire explain plan, and detailed information about each execution step similar to that which is shown by PMON.

It is worth noting that DBQL data is not immediately written to the database upon completion of a query. Rather, DBQL data is written to a database cache, which is periodically flushed to the data dictionary tables.

3.3.1 Enabling DBQL

DBQL is enabled when a database administrator submits a `BEGIN QUERY LOGGING` statement to Teradata. For purposes of performance management and tuning, it is recommended that the `EXPLAIN`, `SQL`, and `STEPINFO` logging options are specified when enabling DBQL.

For example, to begin query logging on user PROD1, the database administrator would execute the following statement.

```
BEGIN QUERY LOGGING WITH EXPLAIN, SQL, STEPINFO ON PROD1;
```

DBQL is disabled when the database administrator submits an `END QUERY LOGGING` statement to Teradata.

3.3.2 DBQL Skew Detection Report

If DBQL logging with SQL and STEPINFO have been enabled, then the following SQL statement may be executed to detect queries that encountered skewed processing during execution the previous day.

Skewed processing is detected by comparing the CPU time used on the busiest AMP on the system for each query to the average AMP CPU time for the query, which is implemented by a filter in blue in the SQL statement below. Smaller, less significant queries are excluded from the report by a minimum CPU time filter, which is in green. The report includes only queries from the previous day due to a timestamp filter which is in red.

```
SELECT   SQ.SqlTextInfo
        ,CASE WHEN ST.QueryID = 0 THEN 0
             ELSE ((ST.ProcID / 100000)*10000000000)+ ST.QueryID
         END (NAMED QueryID, DECIMAL(18,0), FORMAT '-Z(17)9')
        ,ST.ProcID
        ,ST.CollectTimeStamp
        ,ST.StepLev1Num
        ,ST.StepLev2Num
        ,ST.StepName
        ,ST.StepStartTime
        ,ST.StepStopTime
        ,((ST.StepStopTime-ST.StepStartTime) HOUR(2) to SECOND)
```

```
          (NAMED ElapsedTime)
        ,ST.EstProcTime
        ,ST.CPUtime (Format 'ZZ,ZZZ,ZZ9.999')
        ,ST.IOcount
        ,ST.EstRowCount
        ,ST.RowCount
        ,ST.HotAmp1CPU (Format 'ZZ,ZZZ,ZZ9.999')
        ,ST.HotCPUAmpNumber
        ,ST.HotAmp1IO
        ,ST.HotIOAmpNumber
FROM     DBC.DBQLStepTbl ST
INNER JOIN
         DBC.DBQLSQLTbl  SQ
ON       ST.QueryID = SQ.QueryID
AND      ST.ProcID  = SQ.ProcID
AND      CAST(ST.CollectTimeStamp AS DATE) = CAST(SQ.CollectTimeStamp
AS DATE)
WHERE    ST.HotAmp1CPU > 60
  AND    ST.HotAmp1CPU/NULLIFZERO(CPUtime) > .25
  AND    ST.CPUtime >= ST.HotAmp1CPU
  AND    SQ.SqlRowNo = 1
  AND    CAST(ST.CollectTimeStamp AS DATE) = CURRENT_DATE - 1
;
```

3.3.3 DBQL Product Join Detection Report

If DBQL logging with SQL and STEPINFO have been enabled, then the following SQL statement may be executed to detect queries that encountered problematic product joins during execution the previous day.

Large product joins are detected by comparing the CPU time used by the query to the I/O performed by the query, which is implemented by a filter in blue in the SQL statement below. Smaller, less significant queries are excluded from the report by a minimum CPU time filter, which is in green. The report includes only queries from the previous day due to a timestamp filter which is in red.

```
SELECT   SQ.SqlTextInfo
        ,CASE WHEN ST.QueryID = 0 THEN 0
              ELSE ((ST.ProcID / 100000)*10000000000)+ ST.QueryID
         END (NAMED QueryID, DECIMAL(18,0), FORMAT '-Z(17)9')
        ,ST.ProcID
        ,ST.CollectTimeStamp
        ,ST.StepLev1Num
        ,ST.StepLev2Num
        ,ST.StepName
        ,ST.StepStartTime
        ,ST.StepStopTime
        ,((ST.StepStopTime-ST.StepStartTime) HOUR(2) to SECOND)
          (NAMED ElapsedTime)
        ,ST.EstProcTime
        ,ST.CPUtime (Format 'ZZ,ZZZ,ZZ9.999')
        ,ST.IOcount
        ,ST.EstRowCount
        ,ST.RowCount
        ,ST.HotAmp1CPU (Format 'ZZ,ZZZ,ZZ9.999')
        ,ST.HotCPUAmpNumber
        ,ST.HotAmp1IO
        ,ST.HotIOAmpNumber
FROM     DBC.DBQLStepTbl ST
INNER JOIN
         DBC.DBQLSQLTbl  SQ
ON       ST.QueryID = SQ.QueryID
AND      ST.ProcID  = SQ.ProcID
AND      CAST(ST.CollectTimeStamp AS DATE) = CAST(SQ.CollectTimeStamp
AS DATE)
WHERE    ST.CPUtime*10 >= ST.IOCount
  AND    ST.CPUtime > 60
  AND    SQ.SqlRowNo = 1
  AND    ST.StepName = 'JIN'
  AND    CAST(ST.CollectTimeStamp AS DATE) = CURRENT_DATE - 1
;
```