Teradata – Statistics

Teradata optimizer comes up with an execution strategy for every SQL query. This execution strategy is based on the statistics collected on the tables used within the SQL query. Statistics on the table is collected using COLLECT STATISTICS command. Optimizer requires environment information and data demographics to come up with optimal execution strategy.

### Environment Information

* Number of Nodes, AMPs and CPUs
* Amount of memory

### Data Demographics

* Number of rows
* Row size
* Range of values in the table
* Number of rows per value
* Number of Nulls

There are three approaches to collect statistics on the table.

* Random AMP Sampling
* Full statistics collection
* Using SAMPLE option

## Collecting Statistics

COLLECT STATISTICS command is used to collect statistics on a table.

### Syntax

Following is the basic syntax to collect statistics on a table.

COLLECT STATISTICS

INDEX (indexname) COLUMN (columnname)

ON <tablename>;

### Example

The following example collects statistics on EmployeeNo column of Employee table.

Teradata BTEQ 13.00.00.03 for WIN32. Enter your logon or BTEQ command:

.logon localtd/dbc

.logon localtd/dbc

Password:

\*\*\* Logon successfully completed.

\*\*\* Teradata Database Release is 13.00.00.12

\*\*\* Teradata Database Version is 13.00.00.12

\*\*\* Transaction Semantics are BTET.

\*\*\* Character Set Name is 'ASCII'.

\*\*\* Total elapsed time was 1 second.

**To change the password of a user.**

BTEQ -- Enter your DBC/SQL request or BTEQ command:

MODIFY USER tdhari AS PASSWORD ="admin$123";

MODIFY USER tdhari AS PASSWORD ="admin$123";

\*\*\* Database/User has been modified.

\*\*\* Total elapsed time was 1 second.

BTEQ -- Enter your DBC/SQL request or BTEQ command:

.logon localtd/tdhari

.logon localtd/tdhari

Password:

\*\*\* Logon successfully completed.

\*\*\* Teradata Database Release is 13.00.00.12

\*\*\* Teradata Database Version is 13.00.00.12

\*\*\* Transaction Semantics are BTET.

\*\*\* Character Set Name is 'ASCII'.

\*\*\* Total elapsed time was 1 second.

.set width 32767

BTEQ -- Enter your DBC/SQL request or BTEQ command:

select \* from emp1;

select \* from emp1;

\*\*\* Query completed. 5 rows found. 6 columns returned.

\*\*\* Total elapsed time was 1 second.

EMP\_NO DEPT\_NO FIRST\_NAME LAST\_NAME SALARY ADDRESS

----------- ----------- -------------------- -------------------- ------------ ---------

3000 20 Mark Taylor 50000.00 Pune

5000 30 Gita Singh 50000.00 Pune

2000 10 Dev Singh 50000.00 Pune

1000 10 Ram Singh 50000.00 Pune

4000 40 HariYadav 50000.00 Pune

BTEQ -- Enter your DBC/SQL request or BTEQ command:

COLLECT STATISTICS COLUMN(EMP\_NO) ON Emp1;

COLLECT STATISTICS COLUMN(EMP\_NO) ON Emp1;

\*\*\* Update completed. One row changed.

\*\*\* Total elapsed time was 1 second.

BTEQ -- Enter your DBC/SQL request or BTEQ command:

COLLECT STATISTICS INDEX(emp\_no) ON emp1;

COLLECT STATISTICS INDEX(emp\_no) ON emp1;

\*\*\* Update completed. One row changed.

\*\*\* Total elapsed time was 1 second.

BTEQ -- Enter your DBC/SQL request or BTEQ command:

COLLECT STATISTICS ON emp1;

COLLECT STATISTICS ON emp1;

\*\*\* Update completed. One row changed.

\*\*\* Total elapsed time was 1 second.

## Viewing Statistics

You can view the collected statistics using HELP STATISTICS command.

### Syntax

Following is the syntax to view the statistics collected.

HELP STATISTICS <tablename>;

### Example 1.

Following is an example to view the statistics collected on Employee table emp1.

BTEQ -- Enter your DBC/SQL request or BTEQ command:

HELP STATISTICS emp1;

HELP STATISTICS emp1;

\*\*\* Help information returned. One row.

\*\*\* Total elapsed time was 1 second.

Date Time Unique Values Column Names

-------- -------- -------------------- --------------

17/07/25 23:02:33 5 EMP\_NO

**Example 2.**

First gather statistics on column or index and then table level, otherwise error is reported.

BTEQ -- Enter your DBC/SQL request or BTEQ command:

COLLECT STATISTICS ON TDHARI.department;

COLLECT STATISTICS ON TDHARI.department;

\*\*\* Failure 3624 There are no statistics defined for the table.

Statement# 1, Info =0

\*\*\* Total elapsed time was 1 second.

BTEQ -- Enter your DBC/SQL request or BTEQ command:

COLLECT STATISTICS ON TDHARI.department COLUMN(DEPARTMENTNO);

COLLECT STATISTICS ON TDHARI.department COLUMN(DEPARTMENTNO);

\*\*\* Update completed. One row changed.

\*\*\* Total elapsed time was 1 second.

BTEQ -- Enter your DBC/SQL request or BTEQ command:

COLLECT STATISTICS ON TDHARI.department;

COLLECT STATISTICS ON TDHARI.department;

\*\*\* Update completed. One row changed.

\*\*\* Total elapsed time was 1 second.

BTEQ -- Enter your DBC/SQL request or BTEQ command:

HELP STATISTICS DEPARTMENT;

HELP STATISTICS department;

\*\*\* Help information returned. One row.

\*\*\* Total elapsed time was 1 second.

Date Time Unique Values Column Names

-------- -------- -------------------- ---------------------------

17/07/25 23:28:12 3 DEPARTMENTNO

BTEQ -- Enter your DBC/SQL request or BTEQ command:

# Teradata - Compression

Compression is used to reduce the storage used by the tables. In Teradata, compression can compress up to 255 distinct values including NULL. Since the storage is reduced, Teradata can store more records in a block. This results in improved query response time since any I/O operation can process more rows per block. Compression can be added at table creation using CREATE TABLE or after table creation using ALTER TABLE command.

## Limitations

* Only 255 values can be compressed per column.
* Primary Index column cannot be compressed.
* Volatile tables cannot be compressed.

## Multi-Value Compression (MVC)

The following table compresses the field DepatmentNo for values 1, 2 and 3. When compression is applied on a column, the values for this column is not stored with the row. Instead the values are stored in the Table header in each AMP and only presence bits are added to the row to indicate the value.

CREATE SET TABLE employees(

EmployeeNo INTEGER,

Ename CHAR(10),

BirthDate DATE FORMAT 'YYYY-MM-DD',

employee\_gender CHAR(1),

DepartmentNo INTEGERCOMPRESS(1,2,3)

)

UNIQUE PRIMARY INDEX(EmployeeNo);

INSERT INTO employees VALUES(1,'Dev','1990-10-30','M',1);

INSERT INTO employees VALUES(2,'Ram','1990-10-30','M',1);

INSERT INTO employees VALUES(3,'Mark','1990-10-30','M',1);

INSERT INTO employees VALUES(4,'Taylor','1990-10-30','M',2);

INSERT INTO employees VALUES(5,'Tyson','1990-10-30','M',2);

INSERT INTO employees VALUES(6,'John','1990-10-30','M',3);

INSERT INTO employees VALUES(7,'July','1990-10-30','F',3);

INSERT INTO employees VALUES(8,'Hari','1990-10-30','M',4);

BTEQ -- Enter your DBC/SQL request or BTEQ command:

select \* from employees order by 1;

select \* from employees order by 1;

\*\*\* Query completed. 8 rows found. 5 columns returned.

\*\*\* Total elapsed time was 1 second.

EmployeeNoEnameBirthDateemployee\_genderDepartmentNo

----------- ---------- ---------- --------------- ------------

1 Dev 1990-10-30 M 1

2 Ram 1990-10-30 M 1

3 Mark 1990-10-30 M 1

4 Taylor 1990-10-30 M 2

5 Tyson 1990-10-30 M 2

6 John 1990-10-30 M 3

7 July 1990-10-30 F 3

8 Hari 1990-10-30 M 4

BTEQ -- Enter your DBC/SQL request or BTEQ command:

SHOW TABLE DEPARTMENT;

\*\*\* Text of DDL statement returned.

\*\*\* Total elapsed time was 1 second.

-----------------------------------------------------------------------

CREATE SET TABLE TDHARI.DEPARTMENT ,NO FALLBACK ,

NO BEFORE JOURNAL,

NO AFTER JOURNAL,

CHECKSUM = DEFAULT

(

DEPARTMENTNO INTEGER,

departmentname VARCHAR(20) CHARACTER SET LATIN NOT CASESPECIFIC,

departmentloc VARCHAR(20) CHARACTER SET LATIN NOT CASESPECIFIC)

PRIMARY INDEX ( DEPARTMENTNO );

BTEQ -- Enter your DBC/SQL request or BTEQ command:

ALTER TABLE department ADD projnochar(10) COMPRESS;

ALTER TABLE department ADD projnochar(10) COMPRESS;

\*\*\* Table has been modified.

\*\*\* Total elapsed time was 1 second.

BTEQ -- Enter your DBC/SQL request or BTEQ command:

ALTER TABLE department ADD projnnamechar(10) COMPRESS('test001');

ALTER TABLE department ADD projnnamechar(10) COMPRESS('test001');

\*\*\* Table has been modified.

\*\*\* Total elapsed time was 1 second.

INSERT INTO department values(40,'FINANCE','PUNE','T001','TEST001');

INSERT INTO department values(50,'TESTING','PUNE','T002','TEST001');

BTEQ -- Enter your DBC/SQL request or BTEQ command:

select \* from department;

select \* from department;

\*\*\* Query completed. 5 rows found. 5 columns returned.

\*\*\* Total elapsed time was 1 second.

DEPARTMENTNO departmentnamedepartmentlocprojnoprojnname

------------ -------------------- -------------------- ---------- ----------

40 FINANCE PUNE ? ?

30 INSURANCE BANGALORE ? ?

40 FINANCE PUNE T001 TEST001

10 ACCOUNTING BANGALORE ? ?

20 SALES BANGALORE ? ?

50 TESTING PUNE ? ?

50 TESTING PUNE T002 TEST001

BTEQ -- Enter your DBC/SQL request or BTEQ command:

Multi-Value compression can be used when you have a column in a large table with finite values.

# Teradata - Explain

EXPLAIN command returns the execution plan of parsing engine in English. It can be used with any SQL statement except on another EXPLAIN command. When a query is preceded with EXPLAIN command, the execution plan of the Parsing Engine is returned to the user instead of AMPs.

## Examples of EXPLAIN

Consider the table Employee with the following definition.

CREATE SET TABLE EMPLOYEE,FALLBACK(

EmployeeNo INTEGER,

FirstName VARCHAR(30),

LastName VARCHAR(30),

DOB DATE FORMAT 'YYYY-MM-DD',

JoinedDate DATE FORMAT 'YYYY-MM-DD',

DepartmentNo BYTEINT

)

UNIQUE PRIMARY INDEX (EmployeeNo);

Some examples of EXPLAIN plan are given below.

## Full Table Scan (FTS)

When no conditions are specified in the SELECT statement, then the optimizer may choose to use Full Table Scan where each and every row of the table is accessed.

### Example

Following is a sample query where the optimizer may choose FTS.

EXPLAIN SELECT \* FROM employee;

When the above query is executed, it produces the following output. As can be seen the optimizer chooses to access all AMPs and all rows within the AMP.

1) First, we lock a distinct TDUSER."pseudo table" for read on a

RowHash to prevent global deadlock for TDUSER.employee.

2) Next, we lock TDUSER.employee for read.

3) We do an all-AMPs RETRIEVE step from TDUSER.employee 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 2 rows (116 bytes).

The estimated time for this step is 0.03 seconds.

4) Finally, we send out an END TRANSACTION step to all AMPs involved

in processing the request.

→ The contents of Spool 1 are sent back to the user as the result of

statement 1. The total estimated time is 0.03 seconds.

## Unique Primary Index

When the rows are accessed using Unique Primary Index, then it is one AMP operation.

EXPLAIN SELECT \* FROM employee WHERE EmployeeNo = 101;

When the above query is executed, it produces the following output. As can be seen it is a single-AMP retrieval and the optimizer is using the unique primary index to access the row.

1) First, we do a single-AMP RETRIEVE step from TDUSER.employee by

way of the unique primary index "TDUSER.employee.EmployeeNo = 101"

with no residual conditions. The estimated time for this step is

0.01 seconds.

→ The row is sent directly back to the user as the result of

statement 1. The total estimated time is 0.01 seconds.

## Unique Secondary Index

When the rows are accessed using Unique Secondary Index, it’s a two amp operation.

### Example

Consider the table Salary with the following definition.

CREATE SET TABLE SALARY,FALLBACK(

EmployeeNo INTEGER,

Gross INTEGER,

Deduction INTEGER,

NetPay INTEGER

)

PRIMARY INDEX (EmployeeNo)

UNIQUE INDEX (EmployeeNo);

Consider the following SELECT statement.

EXPLAIN SELECT \* FROM Salary WHERE EmployeeNo = 101;

When the above query is executed, it produces the following output. As can be seen the optimizer retrieves the row in two amp operation using unique secondary index.

1) First, we do a two-AMP RETRIEVE step from TDUSER.Salary

by way of unique index # 4 "TDUSER.Salary.EmployeeNo =

101" with no residual conditions. The estimated time for this

step is 0.01 seconds.

→ The row is sent directly back to the user as the result of

statement 1. The total estimated time is 0.01 seconds.

## Additional Terms

Following is the list of terms commonly seen in EXPLAIN plan.

**... (Last Use) …**

A spool file is no longer needed and will be released when this step completes.

**... with no residual conditions …**

All applicable conditions have been applied to the rows.

**... END TRANSACTION …**

Transaction locks are released, and changes are committed.

**... eliminating duplicate rows ...**

Duplicate rows only exist in spool files, not set tables. Doing a DISTINCT operation.

**... by way of a traversal of index #n extracting row ids only …**

A spool file is built containing the Row IDs found in a secondary index (index #n)

**... we do a SMS (set manipulation step) …**

Combining rows using a UNION, MINUS, or INTERSECT operator.

**... which is redistributed by hash code to all AMPs.**

Redistributing data in preparation for a join.

**... which is duplicated on all AMPs.**

Duplicating data from the smaller table (in terms of SPOOL) in preparation for a join.

**... (one\_AMP) or (group\_AMPs)**

Indicates one AMP or subset of AMPs will be used instead of all AMPs.

# Teradata - Primary Index

Primary index is used to specify where the data resides in Teradata. It is used to specify which AMP gets the data row. Each table in Teradata is required to have a primary index defined.If the primary index is not defined, Teradata automatically assigns the primary index. Primary index provides the fastest way to access the data. A primary may have a maximum of 64 columns.

Primary index is defined while creating a table. There are 2 types of Primary Indexes.

* Unique Primary Index(UPI)
* Non Unique Primary Index(NUPI)

## Unique Primary Index (UPI)

If the table is defined to be having UPI, then the column deemed as UPI should not have any duplicate values. If any duplicate values are inserted, they will be rejected.

### Create Unique Primary Index

The following example creates the Salary table with column EmployeeNo as Unique Primary Index.

CREATE SET TABLE Salary (

EmployeeNo INTEGER,

Gross INTEGER,

Deduction INTEGER,

NetPay INTEGER

)

UNIQUE PRIMARY INDEX(EmployeeNo);

## Non Unique Primary Index (NUPI)

If the table is defined to be having NUPI, then the column deemed as UPI can accept duplicate values.

### Create Non Unique Primary Index

The following example creates the employee accounts table with column EmployeeNo as Non Unique Primary Index. EmployeeNo is defined as Non Unique Primary Index since an employee can have multiple accounts in the table; one for salary account and another one for reimbursement account.

CREATE SET TABLE Employee \_Accounts (

EmployeeNo INTEGER,

employee\_bank\_account\_type BYTEINT.

employee\_bank\_account\_number INTEGER,

employee\_bank\_name VARCHAR(30),

employee\_bank\_city VARCHAR(30)

)

PRIMARY INDEX(EmployeeNo);

# Teradata - Partitioned Primary Index

Partitioned Primary Index (PPI) is an indexing mechanism that is useful in improving the performance of certain queries. When rows are inserted into a table, they are stored in an AMP and arranged by their row hash order. When a table is defined with PPI, the rows are sorted by their partition number. Within each partition, they are arranged by their row hash. Rows are assigned to a partition based on the partition expression defined.

## Advantages

* Avoid full table scan for certain queries.
* Avoid using secondary index that requires additional physical structure and additional I/O maintenance.
* Access a subset of a large table quickly.
* Drop the old data quickly and add new data.

## Example

Consider the following Orders table with Primary Index on OrderNo.

|  |  |  |  |
| --- | --- | --- | --- |
| **StoreNo** | **OrderNo** | **OrderDate** | **OrderTotal** |
| 101 | 7501 | 2015-10-01 | 900 |
| 101 | 7502 | 2015-10-02 | 1,200 |
| 102 | 7503 | 2015-10-02 | 3,000 |
| 102 | 7504 | 2015-10-03 | 2,454 |
| 101 | 7505 | 2015-10-03 | 1201 |
| 103 | 7506 | 2015-10-04 | 2,454 |
| 101 | 7507 | 2015-10-05 | 1201 |
| 101 | 7508 | 2015-10-05 | 1201 |

Assume that the records are distributed between AMPs as shown in the following tables. Recorded are stored in AMPs, sorted based on their row hash.

**AMP 1**

|  |  |  |
| --- | --- | --- |
| **RowHash** | **OrderNo** | **OrderDate** |
| 1 | 7505 | 2015-10-03 |
| 2 | 7504 | 2015-10-03 |
| 3 | 7501 | 2015-10-01 |
| 4 | 7508 | 2015-10-05 |

**AMP 2**

|  |  |  |
| --- | --- | --- |
| **RowHash** | **OrderNo** | **OrderDate** |
| 1 | 7507 | 2015-10-05 |
| 2 | 7502 | 2015-10-02 |
| 3 | 7506 | 2015-10-04 |
| 4 | 7503 | 2015-10-02 |

If you run a query to extract the orders for a particular date, then the optimizer may choose to use Full Table Scan, then all the records within the AMP may be accessed. To avoid this, you can define the order date as Partitioned Primary Index. When rows are inserted into orders table, they are partitioned by the order date. Within each partition they will be ordered by their row hash.

The following data shows how the records will be stored in AMPs, if they are partitioned by Order Date. If a query is run to access the records by Order Date, then only the partition that contains the records for that particular order will be accessed.

**AMP 1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Partition** | **RowHash** | **OrderNo** | **OrderDate** |
| 0 | 3 | 7501 | 2015-10-01 |
| 1 | 1 | 7505 | 2015-10-03 |
| 1 | 2 | 7504 | 2015-10-03 |
| 2 | 4 | 7508 | 2015-10-05 |

**AMP 2**

|  |  |  |  |
| --- | --- | --- | --- |
| **Partition** | **RowHash** | **OrderNo** | **OrderDate** |
| 0 | 2 | 7502 | 2015-10-02 |
| 0 | 4 | 7503 | 2015-10-02 |
| 1 | 3 | 7506 | 2015-10-04 |
| 2 | 1 | 7507 | 2015-10-05 |

Following is an example to create a table with partition primary Index. PARTITION BY clause is used to define the partition.

CREATE SET TABLE Orders(

StoreNo SMALLINT,

OrderNo INTEGER,

OrderDate DATE FORMAT 'YYYY-MM-DD',

OrderTotal INTEGER

)

PRIMARY INDEX(OrderNo)

PARTITION BY RANGE\_N (

OrderDate BETWEEN DATE '2010-01-01' AND '2016-12-31' EACH INTERVAL '1' DAY

);

In the above example, the table is partitioned by OrderDate column. There will be one separate partition for each day.

# Teradata - Secondary Index

A table can contain only one primary index. More often, you will come across scenarios where the table contains other columns, using which the data is frequently accessed. Teradata will perform full table scan for those queries. Secondary indexes resolve this issue.

Secondary indexes are an alternate path to access the data. There are some differences between the primary index and the secondary index.

* Secondary index is not involved in data distribution.
* Secondary index values are stored in sub tables. These tables are built in all AMPs.
* Secondary indexes are optional.
* They can be created during table creation or after a table is created.
* They occupy additional space since they build sub-table and they also require maintenance since the sub-tables need to be updated for each new row.

There are two types of secondary indexes −

* Unique Secondary Index (USI)
* Non-Unique Secondary Index (NUSI)

## Unique Secondary Index (USI)

A Unique Secondary Index allows only unique values for the columns defined as USI. Accessing the row by USI is at least two ampor possibly all amp operation.

### Create Unique Secondary Index

The following example creates USI on EmployeeNo column of employee table.

CREATE UNIQUE INDEX(EmployeeNo) on employee;

## Non Unique Secondary Index (NUSI)

A Non-Unique Secondary Index allows duplicate values for the columns defined as NUSI. Accessing the row by NUSI is all-amp operation.

### Create Non Unique Secondary Index

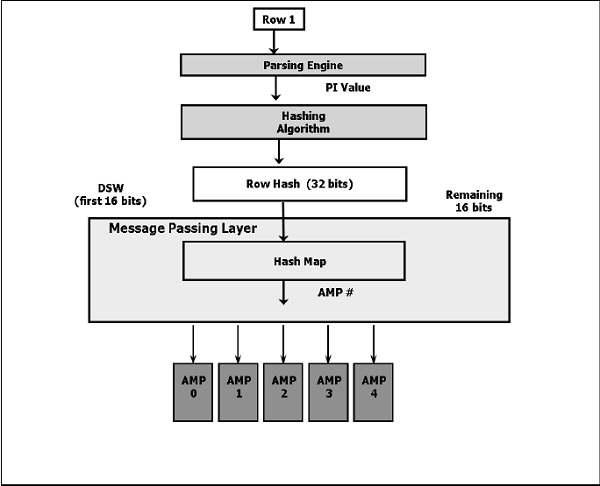
The following example creates NUSI on FirstName column of employee table.

CREATE INDEX(FirstName) on Employee;

# Teradata - Hashing Algorithm

A row is assigned to a particular AMP based on the primary index value. Teradata uses hashing algorithm to determine which AMP gets the row.

Following is a high level diagram on hashing algorithm.



Where

DSW - Destination selection word or hash bucket.

Following are the steps to insert the data.

* The client submits a query.
* The parser receives the query and passes the PI (primary index) value of the record to the hashing algorithm.
* The hashing algorithm hashes the primary index value and returns a 32 bit number, called Row Hash.
* The higher order bits of the row hash (first 16 bits) is used to identify the hash map entry (hash bucket). The hash map contains one AMP #. Hash map is an array of buckets which contains specific AMP #.
* BYNET sends the data to the identified AMP.
* AMP uses the 32 bit Row hash to locate the row within its disk.
* If there is any record with same row hash, then it increments the uniqueness ID which is a 32 bit number. For new row hash, uniqueness ID is assigned as 1 and incremented whenever a record with same row hash is inserted.
* The combination of Row hash and Uniqueness ID is called as Row ID.
* Row ID prefixes each record in the disk.
* Each table row in the AMP is logically sorted by their Row IDs.

## How Tables are Stored

Tables are sorted by their Row ID (Row hash + uniqueness id) and then stored within the AMPs. Row ID is stored with each data row.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Row Hash** | **Uniqueness ID** | **EmployeeNo** | **FirstName** | **LastName** |
| 2A01 2609 | 0000 0001 | 101 | Mike | James |
| 2A01 2610 | 0000 0001 | 102 | Tayson | James |
| 2A01 2610 | 0000 0002 | 102 | Mark | James |
| 2A01 2612 | 0000 0001 | 103 | Alex | Stuart |
| 2A01 2613 | 0000 0001 | 104 | Robert | Williams |
| 2A01 2614 | 0000 0001 | 105 | Robert | James |
| 2A01 2615 | 0000 0001 | 106 | Peter | Paul |

# Teradata - JOIN Index

JOIN INDEX is a materialized view. Its definition is permanently stored and the data is updated whenever the base tables referred in the join index is updated. JOIN INDEX may contain one or more tables and also contain pre-aggregated data. Join indexes are mainly used for improving the performance.

There are different types of join indexes available.

* Single Table Join Index (STJI)
* Multi Table Join Index (MTJI)
* Aggregated Join Index (AJI)

## Single Table Join Index

Single Table Join index allows to partition a large table based on the different primary index columns than the one from the base table.

### Syntax

Following is the syntax of a JOIN INDEX.

CREATE JOIN INDEX <index name>

AS

<SELECT Query>

<Index Definition>;

### Example

Consider the following Employee and Salary tables.

DROP TABLE EMPLOYEES;

CREATE SET TABLE EMPLOYEES

,FALLBACK

(

EmployeeNo INTEGER,

FirstName VARCHAR(10),

LastName VARCHAR(10),

DOB DATE FORMAT 'YYYY-MM-DD',

JoinedDate DATE FORMAT 'YYYY-MM-DD',

DepartmentNo BYTEINT

)

UNIQUE PRIMARY INDEX (EmployeeNo);

INSERT INTO employees VALUES(1,'Dev','Singh','1990-10-30','2015-10-30',1);

INSERT INTO employees VALUES(2,'Ram','Singh','1990-10-30','2016-10-30',1);

INSERT INTO employees VALUES(3,'Mark', 'Taylor', '1990-10-30','2014-10-30',1);

INSERT INTO employees VALUES(4,'Taylor','Cook','1990-10-30','2013-10-30',2);

INSERT INTO employees VALUES(5,'Tyson','Cook','1990-10-30','2016-10-30',2);

INSERT INTO employees VALUES(6,'John','Taylor','1990-10-30','2012-10-30',3);

INSERT INTO employees VALUES(7,'July','Taylor','1990-10-30','2014-10-30',3);

INSERT INTO employees VALUES(8,'Hari','Singh','1990-10-30','2016-10-30',4);

SELECT \* FROM EMPLOYEES;

EmployeeNo FirstName LastName DOB JoinedDate DepartmentNo

----------- ---------- ---------- ---------- ---------- ------------

7 July Taylor 1990-10-30 2014-10-30 3

5 Tyson Cook 1990-10-30 2016-10-30 2

3 Mark Taylor 1990-10-30 2014-10-30 1

1 Dev Singh 1990-10-30 2015-10-30 1

8 Hari Singh 1990-10-30 2016-10-30 4

6 John Taylor 1990-10-30 2012-10-30 3

4 Taylor Cook 1990-10-30 2013-10-30 2

2 Ram Singh 1990-10-30 2016-10-30 1

CREATE SET TABLE SALARY

,FALLBACK

(

EmployeeNo INTEGER,

Gross INTEGER,

Deduction INTEGER,

NetPay INTEGER

)

PRIMARY INDEX (EmployeeNo);

INSERT INTO salary VALUES(1,500000,100000,300000);

INSERT INTO salary VALUES(2,1000000,300000,600000);

INSERT INTO salary VALUES(3,600000,200000,300000);

INSERT INTO salary VALUES(4,700000,100000,600000);

INSERT INTO salary VALUES(5,800000,200000,500000);

SELECT \* FROM salary;

EmployeeNo Gross Deduction NetPay

----------- ----------- ----------- -----------

5 800000 200000 500000

3 600000 200000 300000

1 500000 100000 300000

4 700000 100000 600000

2 1000000 300000 600000

Following is an example that creates a Join index named Employees\_JI on Employee table.

CREATE JOIN INDEX Employees\_JI

AS

SELECT EmployeeNo,FirstName,LastName,DOB,JoinedDate,DepartmentNo

FROM Employees

PRIMARY INDEX(FirstName);

If the user submits a query with a WHERE clause on EmployeeNo, then the system will query the Employees table using the unique primary index.

If the user queries the employees table using FirstName, then the system may access the join index Employees\_JI using FirstName. The rows of the join index are hashed on FirstName column.

If the join index is not defined and the LastName is not defined as secondary index, then the system will perform full table scan to access the rows which is time consuming.

You can run the following EXPLAIN plan and verify the optimizer plan. In the following example you can see that the optimizer is using the Join Index instead of base Employees table when the table queries using the FirstName column.

EXPLAIN SELECT \* FROM EMPLOYEES WHERE EmployeeNo = 5;

\*\*\* Help information returned. 6 rows.

\*\*\* Total elapsed time was 1 second.

Explanation

--------------------------------------------------------------------------------

1) First, we do a single-AMP RETRIEVE step from TDHARI.EMPLOYEES by

way of the unique primary index "TDHARI.EMPLOYEES.EmployeeNo = 5"

with no residual conditions. The estimated time for this step is

0.01 seconds.

-> The row is sent directly back to the user as the result of

statement 1. The total estimated time is 0.01 seconds.

EXPLAIN SELECT \* FROM EMPLOYEES WHERE FirstName = 'Dev';

\*\*\* Help information returned. 8 rows.

\*\*\* Total elapsed time was 1 second.

Explanation

--------------------------------------------------------------------------------

1) First, we do a single-AMP RETRIEVE step from TDHARI.EMPLOYEES\_JI

by way of the primary index "TDHARI.EMPLOYEES\_JI.FirstName = 'Dev'"

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 1 row (76 bytes). The estimated time for this

step is 0.02 seconds.

-> The contents of Spool 1 are sent back to the user as the result of

statement 1. The total estimated time is 0.02 seconds.

**NOTE: Here joine index table is used because LastName column is present in this table.**

EXPLAIN SELECT \* FROM EMPLOYEES WHERE LastName = 'Cook';

\*\*\* Help information returned. 13 rows.

\*\*\* Total elapsed time was 1 second.

Explanation

--------------------------------------------------------------------------------

1) First, we lock a distinct TDHARI."pseudo table" for read on a

RowHash to prevent global deadlock for TDHARI.EMPLOYEES\_JI.

2) Next, we lock TDHARI.EMPLOYEES\_JI for read.

3) We do an all-AMPs RETRIEVE step from TDHARI.EMPLOYEES\_JI by way of

an all-rows scan with a condition of (

"TDHARI.EMPLOYEES\_JI.LastName = 'Cook'") 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 (76 bytes). The

estimated time for this step is 0.03 seconds.

4) Finally, we send out an END TRANSACTION step to all AMPs involved

in processing the request.

-> The contents of Spool 1 are sent back to the user as the result of

statement 1. The total estimated time is 0.03 seconds.

drop join index EMPLOYEES\_JI;

EXPLAIN SELECT \* FROM EMPLOYEES WHERE LastName = 'Cook';

\*\*\* Help information returned. 14 rows.

\*\*\* Total elapsed time was 1 second.

Explanation

---------------------------------------------------------------------------

1) First, we lock a distinct TD\_BIM\_FR\_TRNG\_DB."pseudo table" for

read on a RowHash to prevent global deadlock for

TD\_BIM\_FR\_TRNG\_DB.EMPLOYEES.

2) Next, we lock TD\_BIM\_FR\_TRNG\_DB.EMPLOYEES for read.

3) We do an all-AMPs RETRIEVE step from TD\_BIM\_FR\_TRNG\_DB.EMPLOYEES

by way of an all-rows scan with a condition of (

"TD\_BIM\_FR\_TRNG\_DB.EMPLOYEES.LastName = 'Cook'") 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 (608 bytes).

The estimated time for this step is 0.02 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 0.02 seconds.

## Multi Table Join Index

A multi-table join index is created by joining more than one table. Multi-table join index can be used to store the result set of frequently joined tables to improve the performance.

### Example

The following example creates a JOIN INDEX named Employee\_Salary\_JI by joining Employee and Salary tables.

CREATE JOIN INDEX Employee\_Salary\_JI

AS

SELECT a.EmployeeNo,a.FirstName,a.LastName,

a.BirthDate,a.JoinedDate,a.DepartmentNo,b.Gross,b.Deduction,b.NetPay

FROM Employee a

INNER JOIN Salary b

ON(a.EmployeeNo=b.EmployeeNo)

PRIMARY INDEX(FirstName);

Whenever the base tables Employee or Salary are updated, then the Join index Employee\_Salary\_JI is also automatically updated. If you are running a query joining Employee and Salary tables, then the optimizer may choose to access the data from Employee\_Salary\_JI directly instead of joining the tables. EXPLAIN plan on the query can be used to verify if the optimizer will choose the base table or Join index.

## Aggregate Join Index

If a table is consistently aggregated on certain columns, then aggregate join index can be defined on the table to improve the performance. One limitation of aggregate join index is that it supports only SUM and COUNT functions.

### Example

In the following example Employee and Salary is joined to identify the total salary per Department.

CREATE JOIN INDEX Employee\_Salary\_JI

AS

SELECT a.DepartmentNo,SUM(b.NetPay) AS TotalPay

FROM Employee a

INNER JOIN Salary b

ON(a.EmployeeNo=b.EmployeeNo)

GROUP BY a.DepartmentNo

PrimaryIndex(DepartmentNo);

# Teradata - JOIN strategies

This chapter discusses the various JOIN strategies available in Teradata.

## Join Methods

Teradata uses different join methods to perform join operations. Some of the commonly used Join methods are −

* Merge Join
* Nested Join
* Product Join

## Merge Join

Merge Join method takes place when the join is based on the equality condition. Merge Join requires the joining rows to be on the same AMP. Rows are joined based on their row hash. Merge Join uses different join strategies to bring the rows to the same AMP.

### Strategy #1

If the join columns are the primary indexes of the corresponding tables, then the joining rows are already on the same AMP. In this case, no distribution is required.

Consider the following Employee and Salary Tables.

CREATE SET TABLE EMPLOYEE,FALLBACK(

EmployeeNo INTEGER,

FirstNameVARCHAR(30),

LastNameVARCHAR(30),

DOB DATE FORMAT 'YYYY-MM-DD',

JoinedDate DATE FORMAT 'YYYY-MM-DD',

DepartmentNo BYTEINT

)

UNIQUE PRIMARY INDEX (EmployeeNo);

CREATE SET TABLE Salary(

EmployeeNo INTEGER,

Gross INTEGER,

Deduction INTEGER,

NetPay INTEGER

)

UNIQUE PRIMARY INDEX(EmployeeNo);

When these two tables are joined on EmployeeNo column, then no redistribution takes place since EmployeeNo is the primary index of both the tables which are being joined.

### Strategy #2

Consider the following Employee and Department tables.

CREATE SET TABLE EMPLOYEE,FALLBACK(

EmployeeNo INTEGER,

FirstNameVARCHAR(30),

LastNameVARCHAR(30),

DOB DATE FORMAT 'YYYY-MM-DD',

JoinedDate DATE FORMAT 'YYYY-MM-DD',

DepartmentNo BYTEINT

)

UNIQUE PRIMARY INDEX (EmployeeNo);

CREATE SET TABLE DEPARTMENT,FALLBACK(

DepartmentNo BYTEINT,

DepartmentNameCHAR(15)

)

UNIQUE PRIMARY INDEX (DepartmentNo);

If these two tables are joined on DeparmentNo column, then the rows need to be redistributed since DepartmentNo is a primary index in one table and non-primary index in another table. In this scenario, joining rows may not be on the same AMP. In such case, Teradata may redistribute employee table on DepartmentNo column.

### Strategy #3

For the above Employee and Department tables, Teradata may duplicate the Department table on all AMPs, if the size of Department table is small.

## Nested Join

Nested Join doesn’t use all AMPs. For the Nested Join to take place, one of the condition should be equality on the unique primary index of one table and then joining this column to any index on the other table.

In this scenario, the system will fetch the one row using Unique Primary index of one table and use that row hash to fetch the matching records from other table. Nested join is the most efficient of all Join methods.

## Product Join

Product Join compares each qualifying row from one table with each qualifying row from other table. Product join may take place due to some of the following factors −

* Where condition is missing.
* Join condition is not based on equality condition.
* Table aliases is not correct.
* Multiple join conditions.

# Teradata - OLAP Functions

OLAP functions are similar to aggregate functions except that the aggregate functions will return only one value whereas the OLAP function will provide the individual rows in addition to the aggregates.

### Syntax

Following is the general syntax of the OLAP function.

<Aggregate function> OVER

([PARTITION BY] [ORDER BY columnname][ROWS BETWEEN

UNBOUDED PRECEDING AND UNBOUNDED FOLLOWING)

Aggregation functions can be SUM, COUNT, MAX,MIN, AVG.

### Example

Consider the following Salary table.

|  |  |  |  |
| --- | --- | --- | --- |
| **EmployeeNo** | **Gross** | **Deduction** | **NetPay** |
| 101 | 40,000 | 4,000 | 36,000 |
| 102 | 80,000 | 6,000 | 74,000 |
| 103 | 90,000 | 7,000 | 83,000 |
| 104 | 75,000 | 5,000 | 70,000 |

UNBOUNDED PRECEDING, all rows before the current row -> fixed

UNBOUNDED FOLLOWING, all rows after the current row -> fixed

x PRECEDING, x rows before the current row -> relative

y FOLLOWING, y rows after the current row -> relative

Following is an example to find the cumulative sum or running total of NetPay on Salary table. Records are sorted by EmployeeNo and cumulative sum is calculated on NetPay column.

SELECT

EmployeeNo,NetPay,

SUM(Netpay) OVER(ORDER BY EmployeeNo ROWS

UNBOUNDED PRECEDING)asTotalSalary

FROM Salary;

When the above query is executed, it produces the following output.

EmployeeNoNetPayTotalSalary

----------- ----------- -----------

101 36000 36000

102 74000 110000

103 83000 193000

104 70000 263000

105 18000 281000

## RANK

RANK function orders the records based on the column provided. RANK function can also filter the number of records returned based on the rank.

### Syntax

Following is the generic syntax to use the RANK function.

RANK() OVER

([PARTITION BY columnnlist] [ORDER BY columnlist][DESC|ASC])

### Example

Consider the following Employee table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EmployeeNo** | **FirstName** | **LastName** | **JoinedDate** | **DepartmentID** | **BirthDate** |
| 101 | Mike | James | 3/27/2005 | 1 | 1/5/1980 |
| 102 | Robert | Williams | 4/25/2007 | 2 | 3/5/1983 |
| 103 | Peter | Paul | 3/21/2007 | 2 | 4/1/1983 |
| 104 | Alex | Stuart | 2/1/2008 | 2 | 11/6/1984 |
| 105 | Robert | James | 1/4/2008 | 3 | 12/1/1984 |

Following query orders the records of the employee table by Joined Date and assigns the ranking on Joined Date.

SELECT EmployeeNo,JoinedDate,RANK()

OVER(ORDER BY JoinedDate)asSeniority

FROM Employee;

When the above query is executed, it produces the following output.

EmployeeNoJoinedDate Seniority

----------- ---------- -----------

101 2005-03-27 1

103 2007-03-21 2

102 2007-04-25 3

105 2008-01-04 4

104 2008-02-01 5

PARTITION BY clause groups the data by the columns defined in the PARTITION BY clause and performs the OLAP function within each group. Following is an example of the query that uses PARTITION BY clause.

SELECT EmployeeNo,JoinedDate,RANK()

OVER(PARTITION BY DeparmentNo ORDER BY JoinedDate)asSeniority

FROM Employee;

When the above query is executed, it produces the following output. You can see that the Rank is reset for each Department.

EmployeeNoDepartmentNoJoinedDate Seniority

----------- ------------ ---------- -----------

101 1 2005-03-27 1

103 2 2007-03-21 1

102 2 2007-04-25 2

104 2 2008-02-01 3

105 3 2008-01-04 1

# Teradata - Performance Tuning

This chapter discusses the procedure of performance tuning in Teradata.

## Explain

The first step in performance tuning is the use of EXPLAIN on your query. EXPLAIN plan gives the details of how optimizer will execute your query. In the Explain plan, check for the keywords like confidence level, join strategy used, spool file size, redistribution, etc.

## Collect Statistics

Optimizer uses Data demographics to come up with effective execution strategy. COLLECT STATISTICS command is used to collect data demographics of the table. Make sure that the statistics collected on the columns are up to date.

* Collect statistics on the columns that are used in WHERE clause and on the columns used in the joining condition.
* Collect statistics on the Unique Primary Index columns.
* Collect statistics on Non Unique Secondary Index columns. Optimizer will decide if it can use NUSI or Full Table Scan.
* Collect statistics on the Join Index though the statistics on base table is collected.
* Collect statistics on the partitioning columns.

## Data Types

Make sure that proper data types are used. This will avoid the use of excessive storage than required.

## Conversion

Make sure that the data types of the columns used in join condition are compatible to avoid explicit data conversions.

## Sort

Remove unnecessary ORDER BY clauses unless required.

## Spool Space Issue

Spool space error is generated if the query exceeds per AMP spool space limit for that user. Verify the explain plan and identify the step that consumes more spool space. These intermediate queries can be split and put as separately to build temporary tables.

## Primary Index

Make sure that the Primary Index is correctly defined for the table. The primary index column should evenly distribute the data and should be frequently used to access the data.

## SET Table

If you define a SET table, then the optimizer will check if the record is duplicate for each and every record inserted. To remove the duplicate check condition, you can define Unique Secondary Index for the table.

## UPDATE on Large Table

Updating the large table will be time consuming. Instead of updating the table, you can delete the records and insert the records with modified rows.

## Dropping Temporary Tables

Drop the temporary tables (staging tables) and volatiles if they are no longer needed. This will free up permanent space and spool space.

## MULTISET Table

If you are sure that the input records will not have duplicate records, then you can define the target table as MULTISET table to avoid the duplicate row check used by SET table.