DBA

# Query Optimizer
# Performance Tuning

## Contents



### 1. Enable Query Optimizer:

We can enable the query optimizer using the sql script

*SHOW PARAMETER optimizer_features_enable;*

```
SQL> SHOW PARAMETER optimizer_features_enable;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ----------------
----------
optimizer_features_enable            string      11.2.0.1
SQL>
```

Fig 1.1: Enable Optimizer features

We can use another value of optimizer using

*ALTER SYSTEM SET optimizer_features_enable='11.2.0.2';*

### 2. Components in QUERY OPTIMIZER:

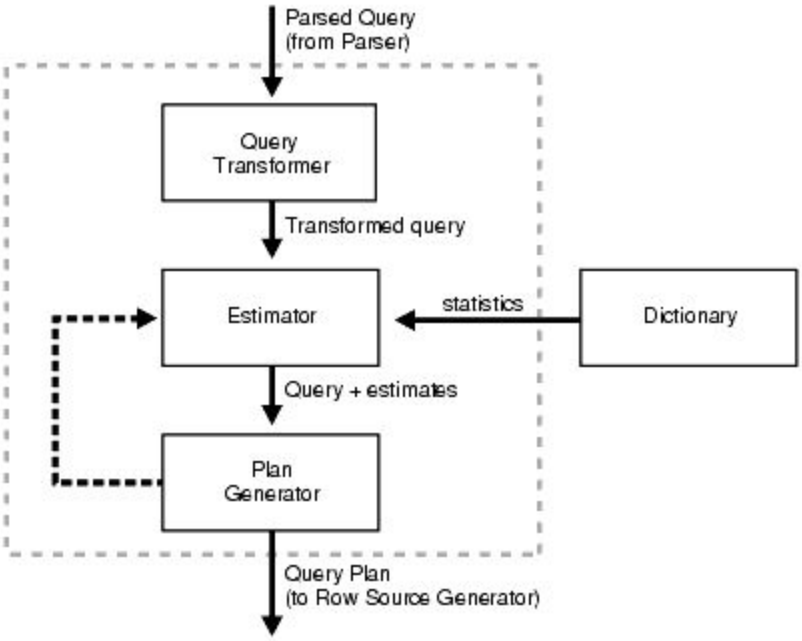


Fig 2.1: Optimizer Components

The basic components involved in query optimizer are:

*Query Transformer*: It basically rewrites the query -- using view merging, with materialized views, etc..-- to make the query easy to be estimated by the estimator.

*Estimator* : It collects the statistics from data dictionary-- can be viewed from DBMS_STATS package-- and estimated the overall cost of the execution plan. The cost basically includes CPU usage, Memory usage, disk I/O usage.

*Plan Generator*: It collects the cost of each plan and passes only the plan with minimum cost.

**3. Optimizer Access Path:**

**a. Full Table Scan:**

- Full table scan reads all rows from the table and filters those which do not meet selection criteria.
- This type of scan is good for faster access of large volume of data because making few large I/O calls is cheaper than making many smaller I/O calls.
- Better to use Full Table scan when table contains :
  - Large amount of Data

- Lack of Index

```
SQL> explain plan for
  2  select * from demo_tab;

Explained.

SQL> select plan_table_output from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-------------------------------------------------------------
Plan hash value: 76495054

--------------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          |     1 |     9 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| DEMO_TAB |     1 |     9 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------------

8 rows selected.
```

Fig 3.a.1: Table Access Full

Here, DEMO_TAB table is scanned fully by the optimizer and is viewed from explain plan.

**b. Index Scan:**

An Index scan retrieves data from an index based on the value of one or more columns in the index.

We created a table called INDEX_TABLE which had an index in id column.

```
create table index_table (
id number,
name varchar2(25)
);

alter table index_table modify id number primary key;

create index index_table_idx on index_table(id);

insert into index_table values (1, 'Saman');
insert into index_table values (2, 'Sumin');
insert into index_table values (3, 'Suresh');
```

Script Output x | Query Result x

Task completed in 0.416 seconds

```
1 row inserted.


Table INDEX_TABLE altered.


1 row inserted.


1 row inserted.
```

Fig 3.b.1: Index table

Now, when we selected the id,name form the table, the optimizer accessed table using index scan and since the where condition was used so we can see the INDEX RANGE SCAN operation and we have the PREDICATE information too about the criteria.

```
SQL> explain plan for
  2  select id,name from index_table where id<3;

Explained.

SQL> select plan_table_output from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
Plan hash value: 3850961986

--------------------------------------------------------------------------------------------
| Id  | Operation                   | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                 |     2 |    54 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| INDEX_TABLE     |     2 |    54 |     1   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | INDEX_TABLE_IDX |     2 |       |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("ID"<3)

Note
-----
   - dynamic sampling used for this statement (level=2)

18 rows selected.

SQL>
```

Fig 3.b.2: Index Scan

An INDEX RANGE SCAN is a operation to access selective data. We used WHERE clause so it was called.

**4. Nested Loop:**

Nested loops are useful when the database joins small subsets of data. Nested loop performs efficiently if the inner loop is dependent to the outer loop else hash join is a better choice.

*NESTED LOOP*
*OUTER LOOP;*
*INNER lOOP;*

**Implementation of Nested Loop:**

```
SQL> explain plan for
  2  select e.name, e.country, d.name as deparment_name from employee e, department d
  3  where e.id = d.id;

Explained.

SQL> select plan_table_output from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------
-------------------------------------------------------------------
Plan hash value: 3611790324

-----------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     3 |   204 |     6   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |               |       |       |            |          |
|   2 |   NESTED LOOPS               |               |     3 |   204 |     6   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL         | EMPLOYEE      |     3 |   123 |     3   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | DEPARTMENT_PK |     1 |       |     0   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENT    |     1 |    27 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("E"."ID"="D"."ID")

Note
-----
   - dynamic sampling used for this statement (level=2)

21 rows selected.
```

Fig 4.1: Nested Loop

Also the concept of nesting nested loop is illustrated as:

```
SELECT STATEMENT
 NESTED LOOP 3
  NESTED LOOP 2          (OUTER LOOP 3.1)
   NESTED LOOP 1         (OUTER LOOP 2.1)
    OUTER LOOP 1.1     - #1
    INNER LOOP 1.2     - #2
   INNER LOOP 2.2      - #3
  INNER LOOP 3.2       - #4
```

Fig 4.2: Nesting Nested Loops

**5. Hash Join:**

Hash Join joins larger data sets.

**Implementation of Hash Join:**

The number of records in above mentioned table(i.e Employee and Department) was more inserted so as to make the data set larger. Then we observed that optimizer chose Hash Joins instead of Nested join.

```
SQL> explain plan for
  2  select e.name, e.country, d.name as deparment_name from employee e, department d
  3  where e.id = d.id;

Explained.

SQL> select plan_table_output from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
Plan hash value: 574014670

-------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |     9 |   612 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN         |            |     9 |   612 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| EMPLOYEE   |     9 |   369 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| DEPARTMENT |     9 |   243 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."ID"="D"."ID")

Note
-----
   - dynamic sampling used for this statement (level=2)

19 rows selected.
```

Fig 5.1: Hash Join

## References:


[1].https://docs.oracle.com/cd/E11882_01/server.112/e41573/optimops.htm#PFGRF001