DBG Software

Real Performance…It's About Time

SQL Performance

Execution Plan Analysis Basics

Background

The purpose of this document is to serve as a guide for analyzing SQL execution plan basics. This is a high level document and is not intended to contain all possible efficiencies related to SQL execution plans, but does highlight those inefficiencies that are most common.

Execution Plan Analysis

What are SQL plans

SQL plans are the sequence of operations the database performs to execute SQL statements. SQL plans consist of the following:

- The access method for each table referenced by the statement.
- The join methods used on tables referenced by the statement.
- The join sequence for each table referenced by the statement.

The intent of the database optimization process is to determine the best way to access the data. SQL plans are built during this decision process.

SQL Plan Optimization

- Cost Based optimization
  - Uses database statistics to assign "costs" to access methods and join methods
  - Evaluates options and chooses the one with the lowest "cost"
- Rule Based optimization (Oracle Only)
  - Utilizes a pre-established set of rules used to make decisions

Generally, cost based optimizers produce more efficient SQL plans than rule based optimizers because they factor in the database object statistics (tables, indexes, and columns). Therefore, cost based optimizers are most commonly used. The existence of good database object statistics is essential to the construction of good SQL plans when using a cost based optimizer.

SQL plan breakdown

Join sequence

When the SQL contains more than one table, the optimizer must determine the sequence in which to access the tables. When analyzing the SQL plan, ensure the first table accessed (usually called the "outer table" or the "driving table") is appropriate. The first table should be the most restrictive in terms of rows returned; via the predicates used to access that table. Secondly, ensure the sequence of the remaining tables is appropriate for the SQL in question. Being able to look at an SQL statement, understand it, and then visualize what the join sequence should be is critical.

When looking at an Oracle SQL plan to determine the join sequence, it must be read top-down and then right-left. As you are looking down the plan tree, any tables shown before reaching the rightmost table are processed first. It is a common misconception that plans are to be read simply right-left.

Oracle Example:

```
    Select
           A.NAME
         , C.EMPLID
         , B.DESCR
         , B.COUNTRY_2CHAR
      From
           PS_PERSONAL_DATA A
         , PS_COUNTRY_TBL B
         , PSOPRDEFN C
     Where
           A.EMPLID  = '533758'
       AND C.EMPLID  = A.EMPLID
       AND B.COUNTRY = A.COUNTRY
    /


ID  PARENT_ID QUERY_PLAN
-- ---------- ----------------------------------------------------------------------------------------------------
 0              SELECT STATEMENT  Optimizer=CHOOSE  (Cost=6 Cardinality=17 Bytes=799)
 1          0     HASH JOIN   (Cost=6 Cardinality=17 Bytes=799)
 2          1       NESTED LOOPS   (Cost=3 Cardinality=1 Bytes=31)
 3          2         TABLE ACCESS BY INDEX ROWID PS_PERSONAL_DATA (Cost=2 Cardinality=1 Bytes=25)
 4          3           INDEX UNIQUE SCAN PS_PERSONAL_DATA (Cost=1 Cardinality=241,199 Bytes=0 Columns=1)
 5          2         INDEX RANGE SCAN PSBPSOPRDEFN (Cost=1 Cardinality=1 Bytes=6 Columns=1)
 6          1       TABLE ACCESS FULL PS_COUNTRY_TBL (Cost=2 Cardinality=239 Bytes=3,824)
```

In the above Oracle example, the join sequence is as follows:

1. PS_PERSONAL_DATA
2. PSOPRDEFN
3. PS_COUNTRY_TBL

SQL Server Example:

```
SELECT dbo.Ticket.TicketID
,dbo.Ticket.GP
,dbo.Ticket.BR
,dbo.Ticket.TICKET
,dbo.Ticket.SERVCO
,dbo.Ticket.Active
,dbo.Ticket.PC_STATEMENT
,dbo.Ticket.BILDAT
,dbo.Ticket.PC_CLAIM
,dbo.Ticket.PC_REPAIRER
,dbo.Ticket.PC_REGISTRATION
,dbo.Ticket.RENTER
,dbo.Ticket.INAME
,dbo.Ticket.DUPLICATE_TICKET
,dbo.tblServiceCustomers.CustomerGroupID
  FROM dbo.tblServiceCustomers
     , oj dbo.Ticket
WHERE ((((((NOT((dbo.Ticket.Archived = 0 ) )
         AND (dbo.Ticket.GP LIKE '%' ) )
         AND (dbo.Ticket.BR LIKE '%' ) )
         AND (dbo.Ticket.TICKET LIKE '%' ) )
         AND (dbo.Ticket.SERVCO LIKE '%' ) )
         AND (   (   (dbo.Ticket.PC_CLAIM LIKE '%ma078383%' )
                  OR (dbo.Ticket.PC_REPAIRER LIKE '%ma078383%' ) )
               OR (dbo.Ticket.PC_REGISTRATION LIKE '%ma078383%' ) ) )
     AND (dbo.tblServiceCustomers.SERVCO = dbo.Ticket.SERVCO ) )
```

```
StmtText
-----------------------------------------------------------------------------------------------------------------------
|--Parallelism(Gather Streams)
       |--Hash Match(Inner Join,
              HASH:([tblServiceCustomers].[SERVCO])=([Ticket].[SERVCO]),
              RESIDUAL:([Ticket].[SERVCO]=[tblServiceCustomers].[SERVCO]))
            |--Parallelism(Repartition Streams, PARTITION
                       COLUMNS:([tblServiceCustomers].[SERVCO]))
            |    |--Table Scan(OBJECT:([ukbilling].[dbo].[tblServiceCustomers]))
            |--Parallelism(Repartition Streams, PARTITION COLUMNS:([Ticket].[SERVCO]))
                 |--Clustered Index Scan(OBJECT:([ukbilling].[dbo].[Ticket].[ucxTicket]),
                       WHERE:(Convert([Ticket].[Archived])<>0 AND ((like([Ticket].[PC_CLAIM], '%ma078383%',
                       NULL) OR like([Ticket].[PC_REPAIRER], '%ma078383%', NULL)) OR like([Ticket].[PC_R
```

In the above SQL Server example, the join sequence is as follows:

1. tblServiceCustomers
2. Ticket

Join method

When the SQL contains more than one table, the optimizer must determine the method in which to associate corresponding rows in the tables. There are several join methods (nested loop, merge join, hash join, hybrid join, etc); the types of join methods tend to change names between databases.

Oracle Example:

```
Select
       A.NAME
     , C.EMPLID
     , B.DESCR
     , B.COUNTRY_2CHAR
  From
       PS_PERSONAL_DATA A
     , PS_COUNTRY_TBL B
     , PSOPRDEFN C
 Where
       A.EMPLID  = '533758'
   AND C.EMPLID  = A.EMPLID
   AND B.COUNTRY = A.COUNTRY
/
```

```
ID  PARENT_ID QUERY_PLAN
-- ---------- ----------------------------------------------------------------------------------------
 0            SELECT STATEMENT  Optimizer=CHOOSE  (Cost=6 Cardinality=17 Bytes=799)
 1          0   HASH JOIN   (Cost=6 Cardinality=17 Bytes=799)
 2          1     NESTED LOOPS   (Cost=3 Cardinality=1 Bytes=31)
 3          2       TABLE ACCESS BY INDEX ROWID PS_PERSONAL_DATA (Cost=2 Cardinality=1 Bytes=25)
 4          3         INDEX UNIQUE SCAN PS_PERSONAL_DATA (Cost=1 Cardinality=241,199 Bytes=0 Columns=1)
 5          2       INDEX RANGE SCAN PSBPSOPRDEFN (Cost=1 Cardinality=1 Bytes=6 Columns=1)
 6          1     TABLE ACCESS FULL PS_COUNTRY_TBL (Cost=2 Cardinality=239 Bytes=3,824)
```

In the above Oracle example, the join methods are:

1. A NESTED LOOPS join is used to associate the rows in PS_PERSONAL_DATA to the corresponding rows in PSOPRDEFN.
2. A HASH JOIN is used to associate the rows in the result set of #1 to the corresponding rows in PS_COUNTRY_TBL

SQL Server Example:

```
SELECT dbo.Ticket.TicketID
,dbo.Ticket.GP
,dbo.Ticket.BR
,dbo.Ticket.TICKET
,dbo.Ticket.SERVCO
,dbo.Ticket.Active
,dbo.Ticket.PC_STATEMENT
,dbo.Ticket.BILDAT
,dbo.Ticket.PC_CLAIM
,dbo.Ticket.PC_REPAIRER
,dbo.Ticket.PC_REGISTRATION
,dbo.Ticket.RENTER
,dbo.Ticket.INAME
,dbo.Ticket.DUPLICATE_TICKET
,dbo.tblServiceCustomers.CustomerGroupID
  FROM dbo.tblServiceCustomers
     , oj dbo.Ticket
WHERE ((((((NOT((dbo.Ticket.Archived = 0 ) )
         AND (dbo.Ticket.GP LIKE '%' ) )
         AND (dbo.Ticket.BR LIKE '%' ) )
         AND (dbo.Ticket.TICKET LIKE '%' ) )
         AND (dbo.Ticket.SERVCO LIKE '%' ) )
         AND (   (   (dbo.Ticket.PC_CLAIM LIKE '%ma078383%' )
                  OR (dbo.Ticket.PC_REPAIRER LIKE '%ma078383%' ) )
               OR (dbo.Ticket.PC_REGISTRATION LIKE '%ma078383%' ) ) )
     AND (dbo.tblServiceCustomers.SERVCO = dbo.Ticket.SERVCO ) )
```

```
StmtText
-----------------------------------------------------------------------------------------------------------------------
|--Parallelism(Gather Streams)
       |--Hash Match(Inner Join,
              HASH:([tblServiceCustomers].[SERVCO])=([Ticket].[SERVCO]),
              RESIDUAL:([Ticket].[SERVCO]=[tblServiceCustomers].[SERVCO]))
            |--Parallelism(Repartition Streams, PARTITION
                       COLUMNS:([tblServiceCustomers].[SERVCO]))
            |    |--Table Scan(OBJECT:([ukbilling].[dbo].[tblServiceCustomers]))
            |--Parallelism(Repartition Streams, PARTITION COLUMNS:([Ticket].[SERVCO]))
                 |--Clustered Index Scan(OBJECT:([ukbilling].[dbo].[Ticket].[ucxTicket]),
                       WHERE:(Convert([Ticket].[Archived])<>0 AND ((like([Ticket].[PC_CLAIM], '%ma078383%',
                       NULL) OR like([Ticket].[PC_REPAIRER], '%ma078383%', NULL)) OR like([Ticket].[PC_R
```

In the above SQL Server example, the join methods are:

1. A HASH JOIN is used to associate rows selected from TBLSERVICECUSTOMERS to corresponding rows in the TICKET table

Access Method

The optimizer must determine "how" to access data within each table. There are several access methods (Full Table Scan, Index Unique Range scan, Index range scan, Table Access By Index Rowid, etc); the types of access methods tend to change names between databases.

Oracle Example:

```
Select
       A.NAME
     , C.EMPLID
     , B.DESCR
     , B.COUNTRY_2CHAR
  From
       PS_PERSONAL_DATA A
     , PS_COUNTRY_TBL B
     , PSOPRDEFN C
 Where
       A.EMPLID  = '533758'
   AND C.EMPLID  = A.EMPLID
   AND B.COUNTRY = A.COUNTRY
/
```

```
ID  PARENT_ID QUERY_PLAN
-- ---------- ------------------------------------------------------------------------------------------
 0              SELECT STATEMENT  Optimizer=CHOOSE  (Cost=6 Cardinality=17 Bytes=799)
 1          0     HASH JOIN   (Cost=6 Cardinality=17 Bytes=799)
 2          1       NESTED LOOPS   (Cost=3 Cardinality=1 Bytes=31)
 3          2         TABLE ACCESS BY INDEX ROWID PS_PERSONAL_DATA (Cost=2 Cardinality=1 Bytes=25)
 4          3           INDEX UNIQUE SCAN PS_PERSONAL_DATA (Cost=1 Cardinality=241,199 Bytes=0 Columns=1)
 5          2         INDEX RANGE SCAN PSBPSOPRDEFN (Cost=1 Cardinality=1 Bytes=6 Columns=1)
 6          1       TABLE ACCESS FULL PS_COUNTRY_TBL (Cost=2 Cardinality=239 Bytes=3,824)
```

In the above Oracle example, the access methods are:

1. The table PS_PERSONAL_DATA is accessed via the index ROWID retrieved from the PS_PERSONAL_DATA index.
    a. The ROWID is retrieved from the PS_PERSONAL_DATA index via a direct lookup using the Where clause local predicate.
2. The table PSOPRDEFN is accessed via the index PSBOPRDEFN.
    a. In this case all the data Selected by the query exists in the index so the table does not have to be accessed.
3. The table PS_COUNTRY_TBL is accessed via a full physical sequence read.

SQL Server Example:

```
SELECT dbo.Ticket.TicketID
,dbo.Ticket.GP
,dbo.Ticket.BR
,dbo.Ticket.TICKET
,dbo.Ticket.SERVCO
,dbo.Ticket.Active
,dbo.Ticket.PC_STATEMENT
,dbo.Ticket.BILDAT
,dbo.Ticket.PC_CLAIM
,dbo.Ticket.PC_REPAIRER
,dbo.Ticket.PC_REGISTRATION
,dbo.Ticket.RENTER
,dbo.Ticket.INAME
,dbo.Ticket.DUPLICATE_TICKET
,dbo.tblServiceCustomers.CustomerGroupID
  FROM dbo.tblServiceCustomers
     , oj dbo.Ticket
WHERE ((((((NOT((dbo.Ticket.Archived = 0 ) )
         AND (dbo.Ticket.GP LIKE '%' ) )
         AND (dbo.Ticket.BR LIKE '%' ) )
         AND (dbo.Ticket.TICKET LIKE '%' ) )
         AND (dbo.Ticket.SERVCO LIKE '%' ) )
         AND (   (   (dbo.Ticket.PC_CLAIM LIKE '%ma078383%' )
                  OR (dbo.Ticket.PC_REPAIRER LIKE '%ma078383%' ) )
               OR (dbo.Ticket.PC_REGISTRATION LIKE '%ma078383%' ) ) )
     AND (dbo.tblServiceCustomers.SERVCO = dbo.Ticket.SERVCO ) )
```

```
StmtText
------------------------------------------------------------------------------------------------------------------------
|--Parallelism(Gather Streams)
       |--Hash Match(Inner Join,
              HASH:([tblServiceCustomers].[SERVCO])=([Ticket].[SERVCO]),
              RESIDUAL:([Ticket].[SERVCO]=[tblServiceCustomers].[SERVCO]))
            |--Parallelism(Repartition Streams, PARTITION
                       COLUMNS:([tblServiceCustomers].[SERVCO]))
            |    |--Table Scan(OBJECT:([ukbilling].[dbo].[tblServiceCustomers]))
            |--Parallelism(Repartition Streams, PARTITION COLUMNS:([Ticket].[SERVCO]))
                 |--Clustered Index Scan(OBJECT:([ukbilling].[dbo].[Ticket].[ucxTicket]),
                      WHERE:(Convert([Ticket].[Archived])<>0 AND ((like([Ticket].[PC_CLAIM], '%ma078383%',
                      NULL) OR like([Ticket].[PC_REPAIRER], '%ma078383%', NULL)) OR like([Ticket].[PC_R
```

In the above SQL Server example, the access methods are:

4. The table TBLSERVICECUSTOMERS is access via a full table scan

5. The table TICKET is accessed via a clustered index scan using the UCXTICKET index

SQL plan steps

After the plan has been created the optimizer then executes the plan in a series of steps. The steps can be a one time pass through or part of a loop. The NESTED LOOPS join would be an example of a repeating step.

In the below example you can see the steps taken to obtain the data by the optimizer.

Oracle Example:

```
    Select
           A.NAME
         , C.EMPLID
         , B.DESCR
         , B.COUNTRY_2CHAR
      From
           PS_PERSONAL_DATA A
         , PS_COUNTRY_TBL B
         , PSOPRDEFN C
     Where
           A.EMPLID  = '533758'
       AND C.EMPLID  = A.EMPLID
       AND B.COUNTRY = A.COUNTRY
    /


ID  PARENT_ID QUERY_PLAN
-- ---------- ----------------------------------------------------------------------------------------------
 0              SELECT STATEMENT  Optimizer=CHOOSE  (Cost=6 Cardinality=17 Bytes=799)
 1           0     HASH JOIN   (Cost=6 Cardinality=17 Bytes=799)
 2           1       NESTED LOOPS   (Cost=3 Cardinality=1 Bytes=31)
 3           2         TABLE ACCESS BY INDEX ROWID PS_PERSONAL_DATA (Cost=2 Cardinality=1 Bytes=25)
 4           3           INDEX UNIQUE SCAN PS_PERSONAL_DATA (Cost=1 Cardinality=241,199 Bytes=0 Columns=1)
 5           2         INDEX RANGE SCAN PSBPSOPRDEFN (Cost=1 Cardinality=1 Bytes=6 Columns=1)
 6           1       TABLE ACCESS FULL PS_COUNTRY_TBL (Cost=2 Cardinality=239 Bytes=3,824)
```

In the above Oracle example, the data is obtained as follows:

1. A ROWID from the PS_PERSONAL_DATA index is retrieved based on the local predicate supplied in the Where clause.
2. The ROWID retrieved from the PS_PERSONAL_DATA index is used to retrieve the corresponding rows of data from the PS_PERSONAL_DATA table.
3. The data retrieved from the PS_PERSONAL_DATA table is used to retrieve the corresponding rows from the PSBPSOPRDEFN index directly (no table access).
4. The table PS_COUNTRY_TBL is read through fully and all the rows from this table are compared to the results from PS_PERSONAL_DATA & PSOPRDEFN to "HASH" out the corresponding rows.

In the above example there is a loop to get all the rows from PS_PERSONAL_DATA & PSBOPRDEFN that match the where clause predicates. After this loop is complete the result set is compared to the table PS_COUNTRY_TBL in order to obtain the final result set.

SQL Server Example:

```
SELECT dbo.Ticket.TicketID
,dbo.Ticket.GP
,dbo.Ticket.BR
,dbo.Ticket.TICKET
,dbo.Ticket.SERVCO
,dbo.Ticket.Active
,dbo.Ticket.PC_STATEMENT
,dbo.Ticket.BILDAT
,dbo.Ticket.PC_CLAIM
,dbo.Ticket.PC_REPAIRER
,dbo.Ticket.PC_REGISTRATION
,dbo.Ticket.RENTER
,dbo.Ticket.INAME
,dbo.Ticket.DUPLICATE_TICKET
,dbo.tblServiceCustomers.CustomerGroupID
  FROM dbo.tblServiceCustomers
     , oj dbo.Ticket
WHERE ((((((NOT((dbo.Ticket.Archived = 0 ) )
         AND (dbo.Ticket.GP LIKE '%' ) )
         AND (dbo.Ticket.BR LIKE '%' ) )
         AND (dbo.Ticket.TICKET LIKE '%' ) )
         AND (dbo.Ticket.SERVCO LIKE '%' ) )
         AND (   (   (dbo.Ticket.PC_CLAIM LIKE '%ma078383%' )
                  OR (dbo.Ticket.PC_REPAIRER LIKE '%ma078383%' ) )
               OR (dbo.Ticket.PC_REGISTRATION LIKE '%ma078383%' ) ) )
     AND (dbo.tblServiceCustomers.SERVCO = dbo.Ticket.SERVCO ) )


StmtText
-------------------------------------------------------------------------------------------------------------------
|--Parallelism(Gather Streams)
       |--Hash Match(Inner Join,
              HASH:([tblServiceCustomers].[SERVCO])=([Ticket].[SERVCO]),
              RESIDUAL:([Ticket].[SERVCO]=[tblServiceCustomers].[SERVCO]))
            |--Parallelism(Repartition Streams, PARTITION
                       COLUMNS:([tblServiceCustomers].[SERVCO]))
            |    |--Table Scan(OBJECT:([ukbilling].[dbo].[tblServiceCustomers]))
            |--Parallelism(Repartition Streams, PARTITION COLUMNS:([Ticket].[SERVCO]))
                 |--Clustered Index Scan(OBJECT:([ukbilling].[dbo].[Ticket].[ucxTicket]),
                      WHERE:(Convert([Ticket].[Archived])<>0 AND ((like([Ticket].[PC_CLAIM], '%ma078383%',
                      NULL) OR like([Ticket].[PC_REPAIRER], '%ma078383%', NULL)) OR like([Ticket].[PC_R
```

In the above SQL Server example, the access methods are:

1. A ROWID from the UCXTICKET index is retrieved based on the local predicate supplied in the Where clause.
2. The ROWID retrieved from the UCXTICKET index is used to retrieve the corresponding rows of data from the TICKET table.
3. The data retrieved from the TICKET table is used to retrieve the corresponding rows from the TBLSERVICECUSTOMERS table.

Overall SQL Execution Plan Quality

There are many factors to take into consideration in the determination of the quality of the SQL plan for an individual SQL statement. It is often difficult to tell that you have a bad SQL plan from the plan alone. To make a quality decision, additional factors must be considered.

- Existence and accuracy of object level statistics
- Cardinality (Number of distinct values for your selection & join columns.)
    I. Data distribution of column value cardinality. This takes analyzing the cardinality of a column to the next level, which is to observe the value frequency of that column.
- Frequency of execution.
- If the SQL in question is "sequential" in nature (either returning or working with large numbers of rows early in the plan), then physical order of table data can be important. If the SQL is "random", physical order is not a concern.
- When tuning in non-production, please ensure the following:
    1. Accuracy of table data volumes to represent production
    2. Accuracy of column data distributions to represent production
- Purpose of the SQL statement
    1. Is it batch or interactive?
    2. What is the criticality of the SQL?
    3. A report SQL will typically return more rows and take longer to execute.
    4. An interactive SQL should return more itemized sets of data.

Steps to evaluate the plan:

1. Verify that the object statistics exist and are accurate.
2. Evaluate the SQL statement:
    a. Determine expected 'driving' table.
    b. Determine expected join order & columns.
    c. Determine expected join columns.
3. Evaluate the SQL plan comparing it to your expectations as determined when evaluating the SQL statement.
4. If different from expectations determine why it is different.

List of possible red flags in SQL plans:

1. Cartesian join.
    a. If you have a CARTESIAN JOIN in your plan you are most likely missing a join in your statement.
2. Full table scans.
    a. If the table is of substantial size and a smaller subset of the data is expected to be returned use of an index would typically be a better access to the data.
    b. If the table is a large subordinate table in a plan
3. Full index scans should be investigated if the index can be used with more selection criteria.
4. HASH JOIN for interactive SQL, since this typically means many rows are being interrogated or returned as a results set
5. BITMAP MERGE, since the merging of indexes is generally not as efficient as a composite index
6. Columns field does not have the number of columns matching for that lookup or join step that are available on that index.
7. NESTED LOOPS for an SQL processing a large number of records
8. No cost due to missing object statistics.
9. Execution time not meeting SLA.

Some cost based Optimizers report a "cost" value in the plan; sometimes referred to as a Timeron value. Use discretion when using this value as a basis for SQL plan efficiency. Very often, and misleadingly, a higher cost will be generated for a more efficient SQL plan and a lower cost will be generated for a less efficient SQL plan.

For example, consider the SQL plans shown below (the overall cost is highlighted in red):

```
ID  PRNT QUERY_PLAN
--- ---- --------------------------------------------------------------------------------
  0        SELECT STATEMENT
  1   0      SORT GROUP BY  (Cost=20)
  2   1        NESTED LOOPS   (Cost=18)
  3   2          NESTED LOOPS   (Cost=17)
  4   3            NESTED LOOPS   (Cost=14)
  5   4              NESTED LOOPS   (Cost=11)
  6   5                NESTED LOOPS   (Cost=8)
  7   6                  NESTED LOOPS   (Cost=7)
  8   7                    NESTED LOOPS   (Cost=5)
  9   8                      NESTED LOOPS   (Cost=4)
 10   9                        NESTED LOOPS   (Cost=2)
 11  10                          TABLE ACCESS FULL CBK_CMCT_ASSGN_ROLE (Cost=2)
 12  10                          TABLE ACCESS BY INDEX ROWID CBK_CMCT (Cost=0)
 13  12                            INDEX UNIQUE SCAN XPKCBK_CMCT (Cost=0 Columns=1)
 14   9                        TABLE ACCESS BY INDEX ROWID RENT_CNTRCT (Cost=2)
 15  14                          INDEX UNIQUE SCAN XPKRENT_CNTRCT (Cost=2 Columns=1)
 16   8                      INDEX UNIQUE SCAN XPKDRVR (Cost=1 Columns=2)
 17   7                    TABLE ACCESS BY INDEX ROWID LRD_IORG_STRCT (Cost=2)
 18  17                      INDEX RANGE SCAN XIE2LRD_IORG_STRCT (Cost=2 Columns=1)
 19   6                  INDEX RANGE SCAN XIE2OFC_DIR_BR (Cost=1 Columns=1)
 20   5                TABLE ACCESS BY INDEX ROWID LRD_IORG_STRCT (Cost=3)
 21  20                  INDEX RANGE SCAN XIE2LRD_IORG_STRCT (Cost=2 Columns=1)
 22   4              TABLE ACCESS BY INDEX ROWID LRD_IORG_STRCT (Cost=3)
 23  22                INDEX RANGE SCAN XIE2LRD_IORG_STRCT (Cost=2 Columns=1)
 24   3            TABLE ACCESS BY INDEX ROWID LRD_IORG_STRCT (Cost=3)
 25  24              INDEX RANGE SCAN XIE2LRD_IORG_STRCT (Cost=2 Columns=1)
 26   2          INDEX RANGE SCAN XPKLRD_IORG_STRCT (Cost=1 Columns=2)

ID  PRNT QUERY_PLAN
--- ---- ---------------------------------------------------------------------------------------------------
  0        SELECT STATEMENT
  1   0      SORT GROUP BY  (Cost=193)
  2   1        NESTED LOOPS   (Cost=191)
  3   2          NESTED LOOPS   (Cost=190)
  4   3            NESTED LOOPS   (Cost=166)
  5   4              HASH JOIN   (Cost=142)
  6   5                NESTED LOOPS   (Cost=43)
  7   6                  NESTED LOOPS   (Cost=24)
  8   7                    NESTED LOOPS   (Cost=13)
  9   8                      NESTED LOOPS   (Cost=7)
 10   9                        NESTED LOOPS   (Cost=4)
 11  10                          TABLE ACCESS BY INDEX ROWID LRD_IORG_STRCT (Cost=2)
 12  11                            INDEX RANGE SCAN XIF4LRD_IORG_STRCT (Cost=1 Columns=1)
 13  10                          INDEX RANGE SCAN XPKLRD_IORG_STRCT (Cost=1 Columns=1)
 14   9                        INDEX RANGE SCAN XPKLRD_IORG_STRCT (Cost=1 Columns=1)
 15   8                      INDEX RANGE SCAN XPKLRD_IORG_STRCT (Cost=1 Columns=1)
 16   7                    INDEX RANGE SCAN XPKLRD_IORG_STRCT (Cost=1 Columns=1)
 17   6                  INDEX RANGE SCAN XIE2OFC_DIR_BR (Cost=1 Columns=1)
 18   5                TABLE ACCESS FULL CBK_CMCT_ASSGN_ROLE (Cost=98)
 19   4              TABLE ACCESS BY INDEX ROWID CBK_CMCT (Cost=1)
 20  19                INDEX UNIQUE SCAN XPKCBK_CMCT (Cost=0 Columns=1)
 21   3            TABLE ACCESS BY INDEX ROWID RENT_CNTRCT (Cost=2)
 22  21              INDEX UNIQUE SCAN XPKRENT_CNTRCT (Cost=2 Columns=1)
 23   2          INDEX UNIQUE SCAN XPKDRVR (Cost=1 Columns=2)
```


	Plan 1 Execution Statistics (Cost = 20)	*Plan 2 Execution Statistics (Cost = 193)*	*Percentage Improvement*
Consistent gets	408257	86172	**78.89%**
Physical reads	4826	3713	**23.06%**
CPU time(sec)	7.33	2.36	**67.80%**
Execution Time(sec)	67.86304	3.739137	**94.49%**

Note the above plans are for the same SQL statement. Note that all of the executions statistics are better for the plan that has the higher cost.

Bottom line, never trust the cost value being reported from a cost based optimizer as an indicator of performance. Lower cost does not necessarily indicate improved performance.