# Revamping SQL Queries for Cost Based Optimization

Vamsi Krishna Myalapalli,
Oracle Corporation,
Cyber Park, Salarpuria, Hitec City,
Hyderabad, India
vamsikrishna.vasu@gmail.com

***Abstract*—There exists manifold ways of scripting a query to reach the same business logic. Though, the query may satisfy the intended business logic, it might be insidious in the perspective of execution plan. Optimal execution plan is ensured when the query runs on minimal time and resources. On the occasion if tie occurs in optimal execution plans the one with lowest predicted execution time must be favored. As such, this paper proposes various re-writing techniques for ensuring cost based optimization of queries. The exploratory analysis of this paper could serve as a tuning / bench-marking / management tool for overhauling querying practices and processing. Experimental fall-outs of our investigation and exploration strongly advocate that query performance and operational costs are optimized.**

**Index Terms—*SQL Tuning; SQL Optimization; Query Tuning; Query Optimization; Query Rewrite; PL/SQL Tuning.***

## I. INTRODUCTION

About 80 percent of the problems pertaining to database performance can be diminished by pre tuning the SQL query even before its actual deployment.

The key inputs to the CBO reckoning process are the object statistic(s) which are held in data dictionary. These statistics would designate, (Ex: How many blocks are in a table) and, therefore, how many multi-block reads will be necessary to read it in its entirety. Statistics would be held for indexes, so the CBO will have certain basis for approximating how many single block reads would be necessary to read information from a table consuming an index.

Object statistics are the most essential contributions to the CBO costing algorithm however by no means the only ones. System statistics, initialization parameter settings, SQL profiles, dynamic sampling and SQL baselines are all other examples belongings the CBO contemplates when making cost estimation.

The assessed execution time can be considered as the plan cost. Further paper is organized in the following manner. Section 2 explains the related work and background. Section 3 deals with Proposed Model i.e. the methodologies. Section 4 demonstrates experimental setup i.e. pragmatic implementation of methodologies. Section 5 explains comparative analysis and finally section 6 concludes the paper.

## II. BACKGROUND AND RELATED WORK

It is essential to realize that when it comes to optimization, the human optimization procedure which strives to better the CBO, the aim is to minimize the real-time elapsed time that a query consumes to execute. It is not significant how great the CBO cost evaluation is as long as the query goes fast, and in a similar way, we will not be impressed by a low cost estimate if the query consumes ages to execute.

Query Optimization can take form of either Rule Based or Cost Based referred as Rule Based Optimization (RBO) or CBO respectively. When the database is in mounted state but not open, query cannot use CBO and only RBO is used. High Performance SQL [1] explained RBO and CBO level tuning models. It made part of the queries faster via CBO approach and rest others via RBO approach.

An Appraisal to Optimize SQL Queries [2] explained basic model that rewrites sundry queries to minimize CPU Cost and raise the index utilization. It also reduced rate of parsing (hard parse) and increased query reuse (soft parse).

Augmenting Database Performance via SQL Tuning [3] explained query tuning through index tuning, access types and hints, and database tuning through session handling.

High Performance PL/SQL [4] explained the tuning model to reduce the rate of Context Switching (an overhead) among SQL and PL/SQL engines.

Optimizing SQL Queries in OLAP Database Systems [5] explained models to tune OLAP queries on data ware house systems, where queries are prevalently used for retrieving statistical information instead of transactional information.

This paper explains sundry novel and sophisticated query rewriting methodologies that we have implemented to tune the real time queries.

## III. PROPOSED BENCHMARK

We have tried rewriting the raw queries, which run on production database and witnessed that optimization is seen as per CBO statistics. Cost Based Optimization is independent over architecture and platform. The following are several scenarios that we have come across

*1) Deter Full Table Scan (FTS):* FTS [6] retrieves each and every bit of data from the table i.e. triggers massive amount of disk I/O. This type of access will always turn into a night mare for the production database as it pulls billions of records. FTS would arise in the following cases

a) WHERE clause absence.
b) Index and Statistics of table are not updated.
c) Absence of filtering of rows at WHERE clause.

*2) Using Multi-column Non-unique Indexes precisely:* Adding a column or two to an index typically levies overhead to queries or DML, but creating further indexes would be expensive towards DML.

*3) Index Range Scan (IRS) and (Index Full Scan) IFS:* We should exploit index to obviate explicit sort operations, as index is a pre-sorted structure and attempt to use a local index to sort rows.

**Thwart expressions on columns used in Predicates:** Using function in a predicate might lead to cardinality error(s) in CBO.

*4) Partition Elimination and Expressions in Predicates:* We should strive to recognize the partitions and then access them instead of using functions. This lets the optimizer to take advantage of partitioned scanning.

*5) Enforcing a Join if a Function is used on a Column:* Enforcing a join when using a function is applied on a column will turn an FTS into an IFS.

*6) Equality vs. Inequality Predicates:* Using an inequality operator in a predicate is occasionally inevitable. However its exploitation is so recurrent that SQL Tuning Advisor flags up warning(s). Inequality operators would lead to FTS.

*7) Implicit Data-Type Conversions:* Though the DBMS engine auto transforms data from one type to another, this silent conversion might mask severe performance concerns especially at peak times.

*8) OR vs. UNION ALL vs. UNION:* Prefer the set operator UNION ALL over UNION operator, and OR over UNION ALL. In addition OR has the capablity to type cast, which the other two set operators does not possess.

*9) Evading Dynamic SQL with UNION ALL:* Dynamic SQL [9] must be deterred over Static SQL. Leveraging data access paths and bind variables efficiently will obviate dynamic SQL. Hints [3] might be required to accomplish this.

*10) Evading Multiple Similar Sub-queries:* Like multiple similar aggregations many queries contain multiple similar subqueries. Subqueries increase wait time for outer query evaluation.

*11) Optimizing Sorts with Analytic Functions:* Sorting must be done as less as possible and filtering must be done as much as possible. Apart from this we must always ensure that sorting occurs after filtering.

*12) Thwarting Duplicate Sorts:* Wet should not try to sort more often than required. There is no matter in sorting the same data multiple times, however occasionally duplicated sorting might occur.

*13) Sorting Few Rows:* We can minimize the quantity of information being sorted by minimizing the columns which are sorted. The further way to lessen the quantity of records being sorted is to sort lesser records, which minimizes the CPU effort by avoiding comparisons.

*14) Evading Densification of Data:* Transforming sparse-data into denser form is called Data-Densification. Though data-densification is chiefly targeted on analytics, maximum analytic functions (comprising median and deviations) could be executed with certain development effort far more proficiently without data-densification.

*15) Standalone Procedures vs. Packaged Procedures:* Packaged procedure offers following benefits:

a) Minimized disk I/O: Whole package is fed into RAM when a packaged procedure is invoked for first time. This loading is done in single operation, contrasted with separate loads necessary for standalone procedures.

b) Maintenance: Package body can be substituted and re-compiled without disturbing the specification. Thus, objects which refer package constructs by specification need no recompilation except when package specification is also replaced. Hence, unnecessary recompilations can be reduced.

## IV. EXPERIMENTAL SETUP

In this section, the methodologies specified in earlier section are explained pragmatically via queries that serve in the form of exemplars.

The enhanced queries are tuned as per CBO statistics, so that optimization persists irrespective of the platform and architecture.

1. SQL> select name from system where id = 532;

*Instead of*

SQL> select * from system; // Results in FTS

c) Updating statistics of table and concerned Indexes.

Oracle

SQL> analyze table <table_name> compute statistics;

Updating statistics of specific Index

SQL> analyze index <index_name> compute statistics;

SQL Sever

SQL> update statistics <table_name>;

2. SQL> select * from oe.customers where upper (cust_first_name) = 'vamsi' and upper (cust_last_name) = 'krishna';

*Instead of*

SQL> select * from oe.customers where upper (cust_last_name) = 'krishna';

3. SQL> SELECT /*+ cardinality(s 1e9) */ s.* FROM sh.sales s ORDER BY s.time_id;

*Instead of*

SQL> SELECT /*+ cardinality(s 1e9) index(s (cust_id)) */ s.* FROM sh.sales s ORDER BY s.cust_id;

The first case uses cardinality hint to see what the CBO would do if SALES table held 1,000,000,000 rows and we wanted to sort by TIME_ID. We can use our index on TIME_ID to avoid any kind of sort because the data is already sorted in the index.

The second case represents a failed attempt to use a local index to sort data.

4. SQL> select count (distinct quantity_sold) from sh.sales where time_stamp >= date '1991-01-01' and time_stamp < date '1992-01-01';

*Instead of*

SQL> select count (distinct quantity_sold) from sh.sales where extract (year from time_stamp) = 1991;

5. SQL> select count (distinct quantity_sold) from sale sl, times t where extract (month from t.time_stamp) = 10 and t.time_stamp = sl.time_stamp;

*Instead of*

SQL> select count (distinct quantity_sold) from sale where extract (month from time_stamp) = 10;

6. SQL> select count (gender) from voters where gender =’M’;

*Instead of*

SQL> select count (gender) from voters where gender! = ’F’;

In this case we are converting an FTS into IRS.

7. SQL> select my_date from my_date_table where my_date = sysdate;

The above statement leads to IRS.

*Instead of*

SQL> select my_date from my_date_table where my_date = systimestamp;

The above statement leads to FTS.

9. SQL> select /*+ USE_CONCAT */ * from voters where (gender = 'M' and :b1 = 'M') or (gender = 'F' and :b1 = 'F');

*Instead of*

SQL> select * from voters where gender = 'M' and :b1 = 'M' union all select * from voters where gender = 'F' and :b1 = 'F';

In the second case, first part of the query selects only rows whose bind variable is ‘M’. As such, an indexed path is chosen. The second part of the query would return rows only if bind variable value is ‘F’, and a FTS is therefore appropriate.

In the first case, query does the same with an OR operator, using ‘USE_CONCAT’ hint. CBO does not comprehend that only one of the two halves of the query would be executed and adds the cost of the FTS and the cost of the indexed access path to arrive at the total statement cost.

10. SQL> select p.prod_num, p.prod_label, p.prod_type, sum (amount_sold) sum_amount_sold, sum (qty_sold) sum_qty_sold from sh.sales s, sh.products p where s.prod_num = p.prod_num group by p.prod_num, p.prod_label, p.prod_type;

*Instead of*

SQL> select p.prod_num ,p.prod_label, p.prod_type, (select sum (amount_sold)from sh.sales s where s.prod_num = p.prod_num)sum_amount_sold, (select sum (qty_sold) from sh.sales s where s.prod_num = p.prod_num) sum_qty_sold from sh.products p;

11. Finding the highest rate of ‘AMOUNT_SOLD’ for every ‘PROD_NUM’ from the ‘SALES’ table and we wanted to know ‘CUST_NUM’ by which such sales are related.

SQL> select prod_num, max (amount_sold) biggest_sale, min (cust_num) keep (dense_rank first order by amount_sold desc) biggest_sale_customer from sales group by prod_num;

*Instead of*

SQL> with dq as (select prod_num, amount_sold, max (amount_sold) over (partition by prod_num) biggest_sale, first_value (cust_num) over (partition by prod_num order by amount_sold desc, cust_num asc) biggest_sale_customer from sales) select distinct prod_num, biggest_sale, biggest_sale_customer from dq where amount_sold = biggest_sale;

In second case ‘FIRST_VALUE’ analytic function is used to recognize the value of CUST_NUM that is related with biggest value of AMOUNT_SOLD per product (first value when rows are sorted by AMOUNT_SOLD in descending fashion). The problem with FIRST_VALUE function is that when there are several values of CUST_NUM that all made transactions with the biggest value of AMOUNT_SOLD the result is unspecified. To evade ambiguity CUST_NUM is also sorted so that getting lowest value from qualifying set of values of CUST_NUM is guaranteed. Here we are doing sorting all rows before filtering (happening in outer query).

In the first case ‘FIRST’ aggregate function just keeps one row for each of the product and hence consumes tiny amount of memory. The enhancement in performance is not quite as substantial as estimated cost recommends but it is still computable, even with small volumes.

12. SQL> select cust_num, time_stamp, sum (amount_sold) daily_quantity, sum (sum (amount_sold)) over (partition by cust_num order by time_stamp desc

range between current row and interval '7' day following) week_quantity from sales group by cust_num, time_stamp order by cust_num, time_stamp desc;

*Instead of*

SQL> select cust_num, time_stamp, sum (amount_sold) daily_quantity, sum (sum (amount_sold)) over (partition by cust_num order by time_stamp range between interval '7' day preceding and current row) week_quantity from sales group by cust_num, time_stamp order by cust_num, time_stamp desc;

Second query aggregates data so that multiple sales by the same customer on that day are added together. This is first sort. Using a window clause in analytic function to find the total sales by the present customer in the week up to and counting the present day. This requires second sort. Descending the date order is third sort. By a minor change to code, we can evade one of these sorts.

In first query, analytic function orders the data in a similar way the ORDER BY did, and thus one sort was eliminated.

13. SQL> with dq as (select sls.*, avg (quantity_sold) over (partition by cust_num order by time_stamp range between interval '7' day preceding and current row) avg_week_quantity from sales sls where time_stamp >=date '1991-12-20' and time_stamp <= date '1991-12-27') select * from dq where time_stamp = date '1991-12-27';

*Instead of*

SQL> with dq as (select sls.*, avg (quantity_sold) over (partition by cust_num order by time_stamp range between interval '7' day preceding and current row) avg_week_quantity from sales sls) select * from dq where time_stamp = date '1991-12-27';

'Current Row' indicates current value when RANGE is stated. Hence, all sales from the current day are included even if they appear in subsequent rows.

Second query lists all the records, which happened on December 27, 1991. 'Avg_Weekly_Quantity' lists average value performed by present customer in 7 days ending the present day. This query is very inefficient as sorting entire SALES table before choosing the records we want. We cannot push the predicate into view as we require a week's worth data to compute analytic function. We enhanced the query by introducing a second different, predicate applied before the analytic function is called.

In the first case a second predicate is added which limits the data being sorted to just the 6 days that are needed, ensuing in a massive win of performance.

14. SQL> with dq as (select cust_id, time_id, sum (amount_sold) today_sale, sum (nvl (sum (amount_sold), 0)) over (partition by cust_id order by time_stmp range between interval '7' day preceding and current row) / 7 avg_daily_sale, avg (nvl (sum (amount_sold), 0)) over (partition by cust_id order by time_stmp range between interval '7' day preceding and current row) avg_daily_sale_orig, var_pop (nvl (sum (amount_sold), 0)) over ( partition by cust_id order by time_stmp range between interval '7' day preceding and current row) variance_daily_sale_orig, count (time_stmp) over (partition by cust_id order by time_stmp range between interval '7' day preceding and current row) total_distinct from sh.sales group by cust_id, time_stmp) select cust_id, time_stmp, today_sale, avg_dailysale, sqrt (( ( count_distinct * (variance_daily_sales_orig + power (avg_dailysale_orig - avg_dailysale, 2))) + ((7 - count_distinct) * power (avg_dailysale, 2))) / 6) std_dev_dailysale from dq where time_stmp >=date '1998-01-07';

*Instead of*

SQL> with dq as ( select cust_id, time_stmp, sum (amount_sold) today_sale, avg (nvl (sum (amount_sold), 0)) over (partition by cust_id order by time_stmp range between interval '7' day preceding and current row) avg_daily_sale, stddev (nvl (sum (amount_sold), 0)) over (partition by cust_id order by time_stmp range between interval '7' day preceding and current row) std_dev_dailysale from sh.times t left join sh.sales s partition by (cust_id) using (time_stmp) group by cust_id, time_stmp) select * from dq where today_sale > 0 and time_stmp >=date '1998-01-07';

The second query produces the moving average and moving standard deviation but increased data densification lead to process higher number of rows.

In the first case we picked TIME_STMP and CUST_ID columns from SALES and computed TODAY_SALE, the total sale for the day. We further calculated average daily sale for the previous week, AVG_DAILYSALE, by dividing sum of those sales by 7. Then we generated some additional columns which help the main query compute the precise value for standard deviation without data-densification.

15. SQL> create package my_package as
procedure pack_proc;
end;
/
SQL> create package body my_package as
procedure pack_proc is
begin
dbms_output.put_line('Packaged procedure');
end pack_proc;
end my_package;
/

*Instead of*

```
SQL> create procedure std_proc as
        begin
          dbms_output.put_line('Standalone procedure');
        end;
         /
```

## V. OUTPUT SCREENS AND COMPARISION

Although optimization is seen in all cases we represent some of the results beneath in the form of output screens to describe comparative results. The corresponding serial number to previous section is mentioned in parenthesis with # symbol.

Query results are suppressed to emphasize the execution plan and cost based results i.e. CPU Cost, CPU Time, Access Type, Rows Processed, Bytes Accessed and Turn-Around Time.

```
SQL> WITH dq AS (SELECT prod_id ,amount_sold, MAX (amount_sold) OVER (PARTITION BY prod_id) biggest_sale,
  2   FIRST_VALUE (cust_id)OVER (PARTITION BY prod_id ORDER BY amount_sold DESC, cust_id ASC)
  3    biggest_sale_customer FROM sh.sales) SELECT DISTINCT prod_id, biggest_sale, biggest_sale_customer
  4      FROM dq WHERE amount_sold = biggest_sale;
Elapsed: 00:00:00.25

Execution Plan
----------------------------------------------------------
Plan hash value: 511097491

-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |       |    72 |  2520 |       |  5133   (3)| 00:01:02 |       |       |
|   1 |  HASH UNIQUE         |       |    72 |  2520 |       |  5133   (3)| 00:01:02 |       |       |
|*  2 |   VIEW               |       |  918K|    30M|       |  5074   (2)| 00:01:01 |       |       |
|   3 |    WINDOW SORT       |       |  918K|    12M|    21M|  5074   (2)| 00:01:01 |       |       |
|   4 |     PARTITION RANGE ALL|       |  918K|    12M|       |   499   (4)| 00:00:06 |     1 |    28 |
|   5 |      TABLE ACCESS FULL | SALES |  918K|    12M|       |   499   (4)| 00:00:06 |     1 |    28 |
-------------------------------------------------------------------------------------------------
```

**Fig 5.1: Sorting and Filtering with Analytic Function FIRST_VALUE (#11)**

The figure 5.1 demonstrates the course of processing rows through analytical functions and filtering rows after sorting huge set of rows.

Statistics: CPU Cost - 21,412; CPU Time – 4Sec18ms.

```
SQL> SELECT prod_id, MAX (amount_sold) largest_sale,
  2   MIN (cust_id) KEEP (DENSE_RANK FIRST ORDER BY amount_sold DESC)
  3     largest_sale_customer FROM sh.sales GROUP BY prod_id;
Elapsed: 00:00:00.00

Execution Plan
----------------------------------------------------------
Plan hash value: 4109827725

-----------------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |       |    72 |  1008 |   535  (11)| 00:00:07 |       |       |
|   1 |  SORT GROUP BY       |       |    72 |  1008 |   535  (11)| 00:00:07 |       |       |
|   2 |   PARTITION RANGE ALL|       |  918K|    12M|   494   (3)| 00:00:06 |     1 |    28 |
|   3 |    TABLE ACCESS FULL | SALES |  918K|    12M|   494   (3)| 00:00:06 |     1 |    28 |
-----------------------------------------------------------------------------------------
```

**Fig 5.2: Efficient Filtering and Sorting with Analytic Function DENSE_RANK and FIRST Function (#11)**

The figure 5.2 demonstrates the progression of processing rows through analytical functions and efficiently filtering rows before sorting huge set of rows and thus consuming less memory. Statistics: CPU Cost – 2,058; CPU Time – 26ms.

```
SQL> SELECT cust_id, time_id, SUM (amount_sold) daily_amount, SUM (
  2   SUM (amount_sold)) OVER (PARTITION BY cust_id ORDER BY time_id
  3    RANGE BETWEEN INTERVAL '7' DAY PRECEDING AND CURRENT ROW)
  4     weekly_amount FROM sh.sales GROUP BY cust_id, time_id ORDER BY cust_id, time_id DESC;
Elapsed: 00:00:00.00

Execution Plan
----------------------------------------------------------
Plan hash value: 2853367057

-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |       |  918K|    15M|       | 11038   (2)| 00:02:13 |       |       |
|   1 |  SORT ORDER BY       |       |  918K|    15M|    24M| 11038   (2)| 00:02:13 |       |       |
|   2 |   WINDOW SORT        |       |  918K|    15M|    24M| 11038   (2)| 00:02:13 |       |       |
|   3 |    PARTITION RANGE ALL|       |  918K|    15M|       | 11038   (2)| 00:02:13 |     1 |    28 |
|   4 |     SORT GROUP BY    |       |  918K|    15M|    24M| 11038   (2)| 00:02:13 |       |       |
|   5 |      TABLE ACCESS FULL| SALES |  918K|    15M|       |   499   (4)| 00:00:06 |     1 |    28 |
-------------------------------------------------------------------------------------------------
```

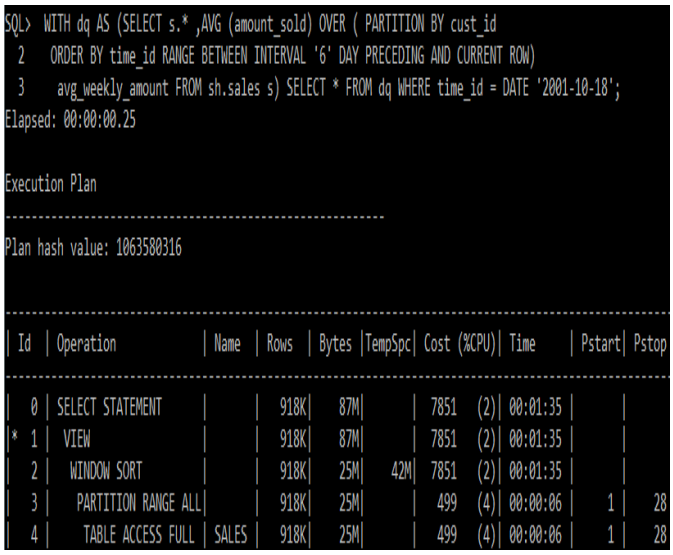
**Fig 5.3: Representing Duplicate Sorts (#12)**

The figure 5.3 exhibits a query engendering a double sort. Statistics: CPU Cost- 55,689; CPU Time – 10Sec71ms.

```
SQL> SELECT cust_id, time_id, SUM (amount_sold) daily_amount ,SUM (
  2  SUM (amount_sold)) OVER (PARTITION BY cust_id ORDER BY time_id DESC
  3  RANGE BETWEEN CURRENT ROW AND INTERVAL '7' DAY FOLLOWING)
  4  weekly_amount FROM sh.sales GROUP BY cust_id, time_id ORDER BY cust_id, time_id DESC;
Elapsed: 00:00:00.00

Execution Plan
----------------------------------------------------------
Plan hash value: 2450472871

-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |       |  918K|    15M|       |  5769   (2)| 00:01:10 |       |       |
|   1 |  WINDOW SORT         |       |  918K|    15M|    24M|  5769   (2)| 00:01:10 |       |       |
|   2 |   PARTITION RANGE ALL|       |  918K|    15M|       |  5769   (2)| 00:01:10 |     1 |    28 |
|   3 |    SORT GROUP BY     |       |  918K|    15M|    24M|  5769   (2)| 00:01:10 |       |       |
|   4 |     TABLE ACCESS FULL| SALES |  918K|    15M|       |   499   (4)| 00:00:06 |     1 |    28 |
-------------------------------------------------------------------------------------------------
```

**Fig 5.4: Eliminated Duplicate Sort (#12)**

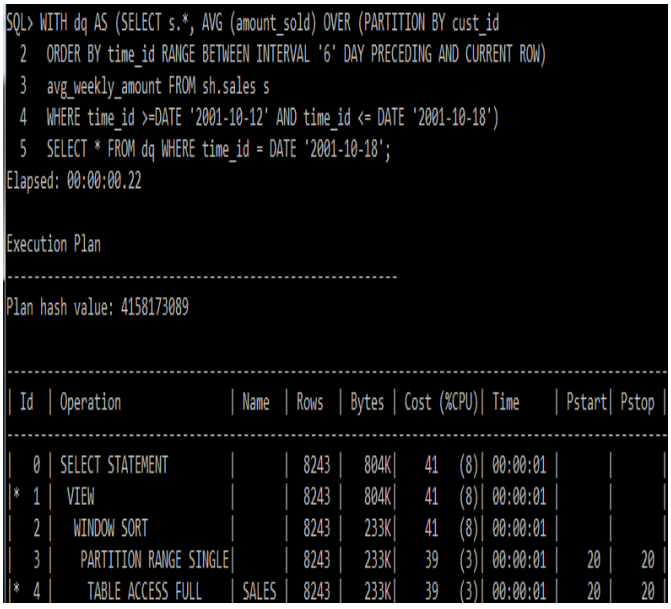
The figure 5.3 exhibits a query obviating the redundant sort. Statistics: CPU Cost – 23,575; CPU Time – 4Sec46ms.



```
SQL> WITH dq AS (SELECT s.* ,AVG (amount_sold) OVER ( PARTITION BY cust_id
  2    ORDER BY time_id RANGE BETWEEN INTERVAL '6' DAY PRECEDING AND CURRENT ROW)
  3     avg_weekly_amount FROM sh.sales s) SELECT * FROM dq WHERE time_id = DATE '2001-10-18';
Elapsed: 00:00:00.25

Execution Plan
----------------------------------------------------------

Plan hash value: 1063580316


-------------------------------------------------------------------------------------------------------------
| Id  | Operation           | Name  | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |   918K|    87M|       |  7851   (2)| 00:01:35 |       |       |
|*  1 |  VIEW               |       |   918K|    87M|       |  7851   (2)| 00:01:35 |       |       |
|   2 |   WINDOW SORT       |       |   918K|    25M|   42M|  7851   (2)| 00:01:35 |       |       |
|   3 |    PARTITION RANGE ALL|       |   918K|    25M|       |   499   (4)| 00:00:06 |     1 |    28 |
|   4 |     TABLE ACCESS FULL | SALES |   918K|    25M|       |   499   (4)| 00:00:06 |     1 |    28 |
```

**Fig 5.5: Sorting Rows before Filtering (#13)**

The figure 5.5 explains inefficiency as sorting entire SALES table is happening before choosing the records. Statistics: CPU Cost – 24,551; CPU Time – 3Sec117ms.



```
SQL> WITH dq AS (SELECT s.*, AVG (amount_sold) OVER (PARTITION BY cust_id
  2    ORDER BY time_id RANGE BETWEEN INTERVAL '6' DAY PRECEDING AND CURRENT ROW)
  3    avg_weekly_amount FROM sh.sales s
  4    WHERE time_id >=DATE '2001-10-12' AND time_id <= DATE '2001-10-18')
  5    SELECT * FROM dq WHERE time_id = DATE '2001-10-18';
Elapsed: 00:00:00.22

Execution Plan
----------------------------------------------------------

Plan hash value: 4158173089


-----------------------------------------------------------------------------------------------
| Id  | Operation             | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |       |  8243 |   804K|    41   (8)| 00:00:01 |       |       |
|*  1 |  VIEW                 |       |  8243 |   804K|    41   (8)| 00:00:01 |       |       |
|   2 |   WINDOW SORT         |       |  8243 |   233K|    41   (8)| 00:00:01 |       |       |
|   3 |    PARTITION RANGE SINGLE|       |  8243 |   233K|    39   (3)| 00:00:01 |    20 |    20 |
|*  4 |     TABLE ACCESS FULL | SALES |  8243 |   233K|    39   (3)| 00:00:01 |    20 |    20 |
```

**Fig 5.6: Sorting Few Rows Using Second Predicate (#13)**

The figure 5.5 explains that after adding second predicate the data being sorted is limited.
Statistics: CPU Cost – 201; CPU Time – 5ms.

## VI. CONCLUSION

Though, the Cost Based Optimizer will often choose a suitable plan for execution, it is also quite usual for it to select a severely suboptimal plan, even if statistics are gathered precisely and are up to date. In a nutshell, the CBO might pick a plan which consumes much longer time to run even when an alternative plan exists.

Without the precise set of extended statistics cardinality errors will arise and cardinality errors will entirely confuse the CBO. Nevertheless, cardinality errors are not the only things that confuse the CBO. CBO has no way to assess the merits of scalar sub-query caching nor has it any way of perceiving which blocks will stay in the cache when the query executes. Hence if the DBA or developer is aware of underlying physics of a query, he/she can construct an efficient query compared to that of re-written query performed by the optimizer.

This paper explained several query rewriting techniques to ensure cost based optimization, as this kind of tuning is architecture and platform independent. All the rewriting methodologies that are implemented witnessed significant optimization in terms of time and resources consumed.

## REFERENCES


[1] Vamsi Krishna Myalapalli and Pradeep Raj Savarapu, "High Performance SQL", 11th IEEE International Conference on Emerging Trends in Innovation and Technology, December 2014, Pune, India.

[2] Vamsi Krishna Myalapalli and Muddu Butchi Shiva, "An Appraisal to Optimize SQL Queries", IEEE International Conference on Pervasive Computing, January 2015, Pune, India.

[3] Vamsi Krishna Myalapalli, Thirumala Padmakumar Totakura and Sunitha Geloth, "Augmenting Database Performance via SQL Tuning", IEEE International Conference on Energy Systems and Application, October 2015, Pune, India.

[4] Vamsi Krishna Myalapalli and Bhupati Lohit Ravi Teja, "High Performance PL/SQL Programming", IEEE International Conference on Pervasive Computing, January 2015, Pune, India.

[5] Vamsi Krishna Myalapalli and Karthik Dussa, "Optimizing SQL Queries in OLAP Database Systems", IEEE International Conference on Information Processing, 2015, Pune, India.

[6] Vamsi Krishna Myalapalli, "Wait Event Tuning in Database Engine", Springer International Journal of Advances in Computing, November 2015.

[7] Vamsi Krishna Myalapalli, Keesari Prathap Reddy and ASN Chakravarthy, "Accelerating SQL Queries by Unravelling Performance Bottlenecks in DBMS Engine", IEEE International Conference on Energy Systems and Application, October 2015, Pune, India.

[8] Vamsi Krishna Myalapalli, "Augmenting Database Performance via SQL Tuning", IEEE International Conference on Energy Systems and Application, October 2015, Pune, India.

[9] Vamsi Krishna Myalapalli and ASN Chakravarthy, "A Unified Model for Cherishing Privacy in Database System", 1st IEEE International Conference on Networks and Soft Computing, August 2014, India .

[10] Vamsi Krishna Myalapalli and Karthik Dussa, "Overhauling PL/SQL Applications for Optimized Performance", IJIRSET Journal, Volume 4, Issue 9, September 2015.