High Performance SQL

Vamsi Krishna Myalapalli
Oracle Corporation,
500 Oracle Parkway,
Redwood City, California, USA.
vamsikrishna.vasu@gmail.com

***Abstract*— In the contemporary world the size of database(s) is rising exponentially. Increased size of database inevitably necessitates performance maintenance in order that performance is enhanced or sustained. The suggested model is envisioned to serve as benchmark for High Performance Database Administration via High performance SQL Query Tuning and thus improving database performance. The strategy of tuning falls on how effectively we can tune the single problem queries and database memory. This paper elucidates sundry tactics to escalate query as well as database performance and can function as a utility for Database Administrators, SQL programmers and Data Center Managers. Experimental outcomes of our analysis designate that performance is enhanced by saving CPU and I/O.**

Index Terms—*Query Optimization; Query Tuning; SQL Best Practices; SQL Optimization; SQL Tuning; DBA Best Practices.*

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.

Data Definition Language and Data Control Language abbreviated for DDL and DCL remain far away from performance engineering, compared to that of Transaction Control Language (TCL) and Data Manipulation Language (DML).

There exist two methods for performance engineering reactive and proactive. “Reactive” methodology indicates performing action when or after a problem occurs itself. “Proactive” approach indicates recognizing pending issues even before they engender problems. Apparently, the second case is the ideal approach to curtail the impact of problems on end users. On the other hand reactive monitoring is also indispensable in several cases.

Proactive performance engineering is delineated in this paper in the form of tactics that the SQL developers should glean at. There are four main areas of performance tuning:
SQL Tuning – Programmer responsibility.
Database Tuning – Database Administrator responsibility.
System Tuning – System Administrator responsibility.
Network Tuning – Network / LAN / WAN Administrator Responsibility.

Further paper is organized in the following manner. Section 2 explains the related work and background. Section 3 deals with proposed benchmark i.e. the tactics. Section 4 demonstrates experimental setup i.e. implementing tactics. Section 5 explains comparative analysis and finally section 6 concludes the paper.

II. BACKGROUND AND RELATED WORK

Optimization of a statement can be either Rule Based or Cost based referred as Rule Based Optimization (RBO) or Cost Based Optimization (CBO) respectively. RBO uses predefined set of precedence rules to find out the path it will use to access the database. RBO is driven by 20 condition rankings which are referred as "Golden rules."

Rank	Condition
1	ROWID = Constant
2	Cluster Join with Unique or Primary Key = Constant
3	Hash Cluster key with Unique or Primary Key = Constant
4	Entire Unique Concatenated Index = Constant
5	Unique Indexed Column = Constant
6	Entire Cluster key = Corresponding Cluster key of another table in same Cluster
7	Hash Cluster Key = Constant
8	Entire Cluster Key = Constant
9	Entire non-UNIQUE CONCATENATED Index = Constant
10	Non-UNIQUE Index Merge
11	Entire Concatenated Index = Lower Bound
12	Most leading column(s) of concatenated Index = Constant
13	Indexed column between low value & high value or Indexed column LIKE "ABC%" (Bounded range)
14	Non-UNIQUE Indexed column between low value & high value or Indexed column like `ABC%' (Bounded range)
15	UNIQUE Indexed column or Constant (Unbounded range)
16	Non-UNIQUE Indexed column or Constant (Unbounded range)
17	Equality on Non-Indexed = column or Constant (Sort/Merge Join)
18	MAX or MIN of single Indexed columns
19	ORDER BY entire Index
20	Full Table Scan

Figure 2.1: RBO Condition Rankings (Golden Rules)

The above rules initiate the optimizer to determine execution path for a statement, when to prefer one index over another and when to make a FTS (Full Table Scan).
On the other hand CBO determines the best execution path for a statement, using database information such as CPU cost, table size, number of rows, key spread etc., rather than going through rigid rules.
A query will default to CBO if any one of the tables involved in the statement has been analyzed. The CBO’s functionality is loosely broken down into the subsequent steps:

1. Query Parsing.
2. Generate all potential execution plans list.
3. Calculate each execution plan cost using all available object statistics.
4. Selecting execution plan of the lowest cost.

III. PROPOSED BENCHMARK

There exist sundry approaches for SQL tuning and this paper explains a holistic tactics of SQL tuning where we optimize the queries.

1) Deter Full Table Scan: (FTS) Full Table Scan retrieves each and every bit of data from the table i.e. triggers a lot of disk I/O. Full Table Scan may arise due to

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

2) Deter Foreign Key Constraints: Though Integrity of data is ensured with Foreign Key constraint(s) it costs performance.So if the primary goal is performance we should push the rules of data integrity into application layer. A fair design of database evades foreign key constraints. Many of the major RDBMS have a set of tables referred as system tables, which hold meta data regarding databases of user. There will be no foreign key relationship, though there exists relationships among tables.

3) Celerity of Operators: The operator(s) present in WHERE clause exhibits their own speed with the associated index. The operators with speed 0 doesn't invoke index.

Table 3.1: Celerity of Operators

Operator	Swiftness
=	10
> >= <= <	5
LIKE	3
<> != NOT	0

Fastest method to Delete Data from Large Table (4-7)

4) Amount of rows that will be deleted: If more than 30-50% of the rows are deleted in a very large table, it is fastest to use CTAS (Create Table As Select) to delete data from a table rather than to do a specific delete and a reorganizing the table blocks and a rebuilding the indexes and constraints.

5) Discharge the space consumed by deleted rows: If we know that the empty space will be re-claimed by subsequent DML then we should leave the empty space within the table. On the other hand, if we want to release the space back onto the tablespace then we should reorganize the table.

6) Table Partitioning: Partitioning impressively enhances delete performance.

7) Specific Delete: (Deleting entries in 24*7 Environment) On a super-large table, a delete statement needs a dedicated rollback segment (UNDO log) & in certain cases, delete is so large that it must be written in PL/SQL with a COMMIT. Parallel DML allows parallelizing large SQL deletes. But a standard SQL delete may result in honey comb fragmentation & also places pages onto the freelist that have used row space.

8) Isolating Table and Indexes: Using different tablespace(s) for table and index will reduce I/O contention.

Fastest Way to Rebuild Index (9-19)

9) Determining when to Index: Index a column if and only if the following condition holds.

Rows retrieved = Total No of rows/10

10) Rebuild Indexes: Rebuild the indexes regularly to reduce intra block fragmentation. Index rebuilding is 100% benign & cost free in a scheduled maintenance downtime.

11) ONLINE keyword for index rebuild: When the online keyword is used table will not be locked i.e. If table is not locked it will not distrurb active users.

12) No logging with an Index rebuild: NOLOGGING option can be used for super fast rebuild of index. On the occasion if nologging is used it neccessitates re-running the create index query, if we do a roll-forward recovery of database. Using "nologging" along with create index could ensure index rebuilding upto 30% quicker.

13) Parallel Scan: Index rebuild with a full-scan can be parallelized based on 'cpu_count' parameter.

14) Partition the index: If index partitioning is enabled, we could rebuild a local partitioned index faster compared to a single large index.

15) Solid State Disk: Depending on the environment fleeting SSD can be used to bring up the speed of early index writes & transfer index in future to platter disk storage.

16) Parallel Index Rebuilds: If there exists indexes and spare CPU on dissimilar disks, index rebuild jobs can be submitted concurrently. Upon a capable server dozens of indexes can be rebuilt concurrently, each using concurrent query. This is a kind of a concurrent concurrence for faster index rebuilds.

17) Recognizing which Indexes to Rebuild: There will be no change in execution plan after rebuild, but an index scan that require block reads will be minimized. Index rebuild is required in following cases

a) Systems that do batch DML jobs.
b) System that never update or delete rows.
c) When large datasets are added and removed.
d) Deleted entries are more than 20% of existing.
e) Depth of Index exceeds by 4 levels.
f) To release free space

18) Large Index Block Size reduces I/O: A bigger block size upshots more space for key storage in the branch nodes of B-tree indexes, which reduces index height and improves indexed queries performance. We can use the large blocksize (16-32 K) and isolated data caches to improve response time under specific situations. Bigger size of block consequences logical Input/Output minimization with larger blocksizes. Apart from this external implementations (Storage Area

Network, Network Access Storage, Level of RAID, Stripe Size) can effect time of response.

19) Index Leaf Block contention Tuning: Contention occurs if records are injected depending on a user generated key (i.e. a sequence). This is due to the reason that sequence key is every time high order key. Every insertion of low-level index tree node should spread upwards towards high-key indicators in B-tree index. We should perform these tuning tasks only if their exists delay events which are straightly associated with index block contention.The following can reduce contention.

a) Adjusting the index block size
b) Hash partitioning global indexes
c) Sequences with the noorder and cache options
d) Reverse Key indexes

20) Recognizing Queries with high execution time: The following tables can be queried to find long running queries.

a) v$active_session_history: The sql_exec_start column will find slow SQL statements
b) v$sql: The elapsed_time column displays slow execution time for current queries.
c) stats$sqlstat: This displays slow queries if we install STATSPACK.
d) dba_hist_sqlstat : If we have purchased the necessary extra cost packs, this table will gather data on slow SQL queries over time.

21) Parameter hash_area_size: The tendency of the optimizer to invoke a hash join is greatly controlled by the setting for the hash_area_size parameter. The higher the value, the higher hash joins the optimizer will invoke.

22) Parameter pga_aggregate_target: This parameter indicates memory limit assigned for particular user. Higher memory enables sorting and other operations faster.

Tuning Physical Read Waits (23-32)

If wait events are to be tuned, the objects that are prone to physical read waits should be recognized and if they do so, the problem should be resolved by tweaking them.

23) Identifying Wait events: The views v$system_event, v$session_wait, v$segment_statistics can be queried to find out the wait events on onjects. v$system_event dynamic view gives statistics about the whole amount of I/O associated waits contained in the database, but it will not tell us the definite object which is responsible.

The dynamic view v$session_wait suggests complete file and block data & we can extract object using block number. Wait events happen quickly & it's tough to retrieve information except when we run query at the precise moment the wait event is experienced in database. Therefore, we should formulate a method for using the view v$session_wait in order that we can catch fleeting physical I/O waits.

24) Change Indexes: Enforcing INDEX Hint or creating FBIs (Function Based Index) can mark SQL less I/O demanding through making use of high selective index.

25) Alter table join approaches: Frequently, nested loop joins require less I/O waits compared to hash joins, specifically for successive read(s). We could alter table join approaches by SQL hints (Example: 'USE_NL' Hint).

26) Alter table join order: In the case of successive read waits, SQL can be tweaked to alter the sequence in which tables are joined (i.e. making use of ORDERED hint).
Hints can be passed to optimizer, to enforce some actions, such as compelling to use a specific method or use particular index.

27) Re-scheduling contentious SQL: Once the frequent trends of reiterating disk waits are identified, the SQL execution can be rescheduled at another time, to relieve physical delays.

28) Increasing size of Data buffer cache: Higher the data blocks we preserve in Main Memory, the lesser the possibility of read delays.

29) Using KEEP pool: To minimize scattered reads, implement KEEP pool. The small-table full-table scans must be located in KEEP pool to minimize dispersed read delays.

30) Share Disk Input/Output over additional spindles: Occasionally physical read delays triggers contention in disk channel. If disk delays occur due to contention in hardware, we can strip the offending objects over several disk spindles by rearranging the object & using the MINEXTENTS and NEXT parameters to stripe the object over several data files on several disks or using volume manager or I/O subsystem provided striping mechanisms.

31) Re-analyze the schema using dbms_stats: In certain cases, non-representative or stale statistics produced by the dbms_utility.analyze_schema package can cause sub optimal SQL execution plans, resulting in needless disk waits. The solution is to make use of dbms_stats package to analyze our schema. In addition if column data values are skewed, adding histograms may also be essential.

32) Tracing Input/Output Waits over particular Indexes and Tables: We have to translate block number and file number to a particular index or table name using the view dba_extents inorder to find out the beginning & ending block for each extent in each table. Making use of dba_extents to find object and its data block borders, is to read over our new table & find those particular objects undergoing buffer busy or delays.

33) Composite Key vs. Surrogate Key: Replacing ugly, big composite keys and natural keys with tight integer surrogate key is bound to enhance join performance. The storage requirements are minimized & index lookups would appear to be simpler. A Surrogate Key could be a system created sequence number or a grouping of parts of a column that serve to make the row unique.

34) Tightly Packing Table Rows onto Data Blocks: If we have read-only tables (i.e. no inserts, deletes or updates) & there is desire to compactly pack the records as compact as probable onto data blocks optmially. To lessen space wastage & guarantee fast retrieval through index consider following

a) Compression: Compression increases query performance.

b) Tablespace fragmentation: Don't allow fragmentation

c) PCTFREE: This is the amount of space (in %) to leave free on succeeding updates or inserts updates. The default is 20%. Inorder to compact data firmly, fix PCTFREE value to minimum.

d) Size of NEXT extent: Less size of NEXT extent results in minimizing wastage.

e) Space Management pages: Locally managed tablespace and Automatic Segment Space Management consume less pages inside a tablespace for managing space.

f) Blocksize: Enforce smaller block size to reduce wasting space, nonetheless we have to remember that optimum size of block is a job of row length i.e. small sized blocks will outcome less wastage of space, except our avg_row_len exceeds block size.

35) Tuning Workload: We should tune our workload comprehensively by enhancing parameters of the optimizer such as db_file_multiblock_read_count, optimizer_mode, optimizer_index_cost_adj etc before tuning SQL queries.

36) CBO Statistics: CBO statistics may not be always accurate. Utilizing GIGO (garbage-in, garbage-out) norm, we can re-analyze indexes & tables with 'dbms_stats' package.

37) Using statistical diagrams to tune: Several SQL problems (such as suboptimal join order of table) are instigated due to low cardinality approximations. Enforce histograms providently (when there is requisite) in assisting the optimizer to calculate intermediary rowset operations size.

38) Decompose complex SQL: We can use the Global Temporary Tables and WITH clause to flatten-out compound sub queries & result in faster execution.

39) Find out anamolous issues: Implementations such as 'where rownum=1' are risky and may engender abrupt results.

40) Deter the HAVING clause: We can break down a compound query using WITH clause in order to prevent the implementation of costlier HAVING clause.

41) Evaluate with RULE hint: RULE hint suits excellent to evaluate if a sub-optimal SQL query is doomed due to imprecise CBO statistics or missing index. In several issues, the RULE hints easiness could assist tuning SQL queries quicker.

42) Don't invalidate Columns: Altering the LHS predicate of a WHERE clause can trigger massive performance issues.

43) Utilizing views cautiously: Views are aimed to support end-users. They disguise compound statements "seem" like a distinct table. So, issuing production queries against views will instigate problems of optimization in mass quantity.

44) Index Partioning: Partitioning Indexes offer several performance benefits such as injecting data through DDL & DML queries, probing data through SELECT statements and maintenance benefits such as rebuilding indexes and setting indexes unusable or invisible at a level of partition.

45) Detecting Long Running Queries: Analyze the V$SESSION_LONGOPS dynamic performance view's elapsed_seconds column to detect long running operations.

46) Predicting DBMS Engine Performance: (AWR) Automatic Workload Repository & STATSPACK are some of the best optimal tools for predicting Database performance using temporal i.e time-series data tracked by them. Utilizing this temporal data, a DBA can recognize concealed performance issues and predict the performance problems even prior they harm the database.

By utilizing verified predicting methods, we can create self-tuning DBMS engine that predicts and expects performance variations and enforces just-in-time tweaking methods to repair upcoming performance problems prior they happen.

47) CBO vs. RBO: If a tie up exists when evaluating performance of multiple queries using CBO, then evaluate queries using RBO analysis.

48) Anonymous vs. Stored PL/SQL Code: Anonymous PL/SQL code is less efficient than stored PL/SQL code and also triggers problems with source code administration, as the code may be spread across several machines.

49) Using Hash Clusters: Hash clusters is a best way to minimize I/O on some tables, but they exhibit their downside.

a) If very less space is alloted for each key or if the cluster is created with too little hash keys, then each key will split over multiple blocks negating the advantages of cluster.

b) If excess space is alloted for each key or if the cluster is created with too many hash keys, then the cluster will encompass thousands of empty blocks that slow down Full Table Scans.

50) High Water Mark (HWM) problem: HWM is the highest numbered block in table that ever contained a row. If an INSERT query can't find a free block to occupy, it goes straight to the HWM & allocates a new block. When rows are deleted, the High Water Mark does not come down. This isn't a problem, except for FTSs, because a FTS must hunt every block beneath the HWM for data. If table is incessantly shrinking & growing (through regular deletes), it can be partitioned, so that instead of deleting rows we can drop partitions or truncate the table. On the occasion if table is not going to grow back to full size in medium term, then it should be dropped & rebuilt to move HWM back down to a correct level. Shrink operation relocates HWM & ensures lesser disk I/O and higher space availability.

IV. EXPERIMENTAL SETUP

In this section the tactics specified in earlier section are explained via queries that serve in the form of exemplary.
The enhanced queries are tuned as per CBO or RBO statistics.

1. select name from employee where id = 532;

Instead of

select * from employee; // Results in Full Table Scan

4. create table <source_table> as select <col_list> from <destination_table>;

10. alter index “myindex” rebuild tablespace “mytablespace” storage (initial 0 next 1M);

12. alter index <index_name> rebuild online nologging;

13. show parameter cpu_count;

14. create index dob_idx on emp (dob)
local
(partition dob_idx1), (partition dob_idx2) ;

15. alter index <indx_name> rebuild online parallel nologging;

16. alter index indx_name rebuild parallel 6;
// 6 is the number of processes to be used & ideally it should be greater (double) the number of processors.

17. select index_name, blevel, leaf_blocks, pct_free from dba_indexes where table_name=’<table_name>’;
analyze index <index_name> compute statistics;
analyze index <index_name> validate structure;

18. show parameter db_block_size;
This parameter can be set only during database creation.

19. Detecting Contention
select sid, sql_text from v$session a, v$sql b where sid in
(select sid from v$session where state in ('WAITING') event='enq: TX - index contention'
and wait_class != 'Idle' and
and (b.sql_id = a.prev_sql_id or b.sql_id = a.sql_id));

20. select sql_text from v$sql where elapsed_time > 100000;

21. show parameter hash_area_size;
alter session set hash_area_size=<hash_area_size>;

22. show parameter pga_aggregate_target;
alter session set pga_agggregate_target = <size>;

23. select event, wait_time from v$session_wait;

26. select /*+ ordered use_nl(bonus) parallel(e, 4) */
e.name, hiredate, b.comm from emp e, bonus b
where e.name = b.name ;

28. show parameter db_cache_size
If buffer memory is managed automatically 0 is returned.

29. show parameter db_keep_cache_size

32. select file_id, block_id from dba_extents;

33. create index surrogate_index on emp (substr (name,1,3)||substr(city,1,3));
// Creates index on surrogate key containing first 3 letters from name column and city column.

34. alter tablespace <tablespace_name> coalesce;
//Combines all fragments of tablespace into a single unit.

35. show parameter optimizer_mode;
If Throughput is aimed then “ALL_ROWS” is returned.

36. analyze index <indx_name> compute statistics;

42. where substr(city,1,3) = 'isa';
where trunc(dob) =trunc(sysdate));

44. create index <index_name> on <table_name>
(column_list)
[tablespace <tablespace_name>] local;

45. select elapsed_seconds from v$session_longops where sql_plan_hash_value = <plan_value>;

50. alter table <table_name> enable row movement;
alter table <table_name> shrink space compact;

V. OUTPUT SCREENS

Some of the results are represented beneath in the form of Output screens.

In order to get statistics for each query, we fired some queries on database prior to tuning.

SQL> set timing on // Displays elapsed time
SQL> set feedback on // Metadata for output
SQL> set autotrace on explain // Tracing execution plan

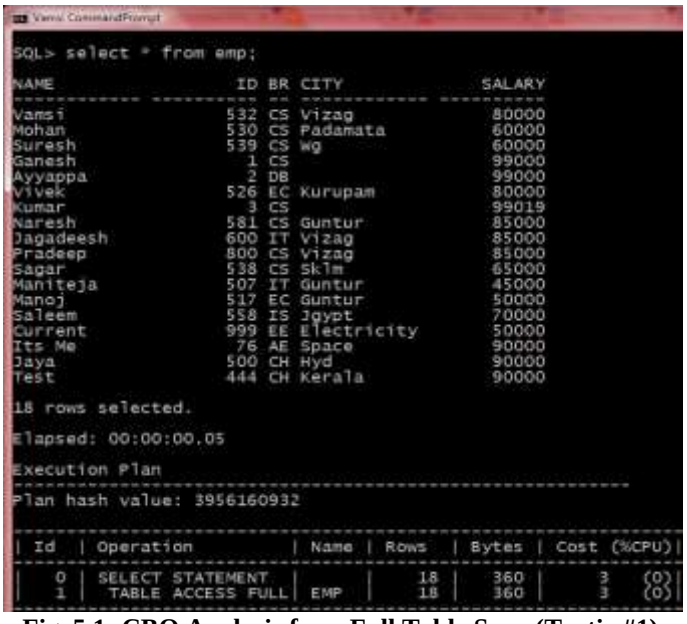

Fig. 5.1: CBO Analysis for a Full Table Scan (Tactic #1)

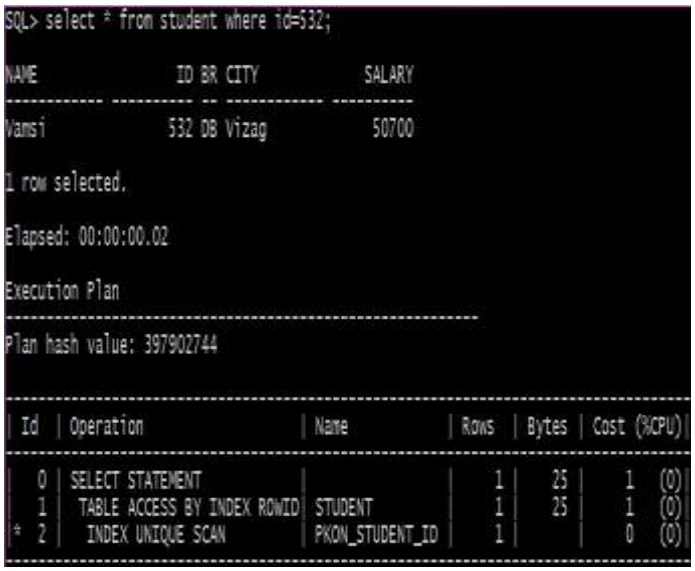

Fig. 5.2: CBO Analysis for Index Unique Scan (Tactic #1)

The figure 5.1 specifies that a Full Table Scan triggers excessive I/O compared to that of IUS in figure 5.2.

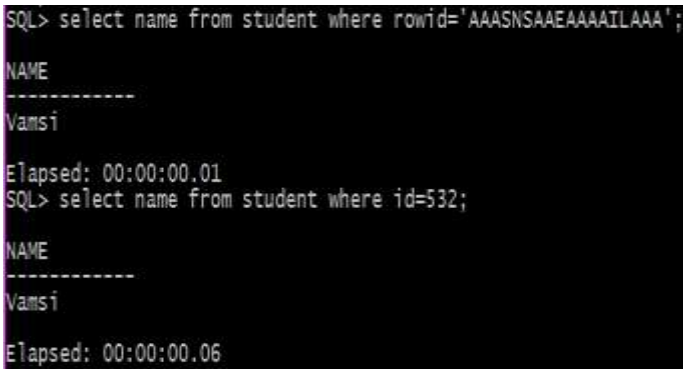

Fig. 5.2: RBO Analysis for Row Access by User Row Id vs. Index Unique Scan (Rule 1 (Fig.2.1)).

In this scenario the 'id' column of 'student' table is indexed. The above screen designates that row access is extremely faster with ROWID compared to Index Unique Scan i.e. using 'id' column.

VI. COMPARATIVE STUDY

On enforcing the tactics tuning can be ensured at Database engine as well as Statement level. As per our results queries exhibited different behavior on different systems, but the Processor Cost i.e. CPU Cost as per Cost Based analysis remained constant. In few cases tie up raised i.e. same CPU Cost remained even after tuning, but tuning is ensured as per Rule Base Analysis ranking (Fig. 2.1).

VII. CONCLUSION

Database tuning necessitates each bit of our ingenuity to evade harmful way. Implementing perpetual SQL tuning, irrespective of modifications to environment ensures performance tuning i.e. minimizing time & space complexities. Every invalid & unusable object should be recognized & fixed.

Evaluating real-time physical I/O delays is a significant move in enhancing performance. Tuning project(s) necessitates the capability to explore and correct DBMS physical read delay events. The maximum of tasks in a DBMS engine encompasses fetching information. Hence this kind of amendment(s) will result in massive, optimistic influence upon performance.

When performing delay investigation, it is crucial to remember that every DBMS engine(s) undergo delay events and that the existence of delay(s) may not always upshot a problem. In fact, every fine-tuned DBMS engines possess certain bottleneck. A computation demanding DBMS engine will be processor bound and a data warehouse is inevitable by disk-read delays. A Database performs well if, access to hardware resources related to delays are enhanced.

VIII. REFERENCES

[1] Vamsi Krishna Myalapalli "An Appraisal to Optimize SQL Queries", IEEE International Conference on Pervasive Computing, Pune, India.

[2] Vamsi Krishna Myalapalli "High Performance PLSQL Programming", IEEE International Conference on Pervasive Computing, Pune, India.

[3] Vamsi Krishna Myalapalli "High Performance JAVA Programming", IEEE International Conference on Pervasive Computing, Pune, India.

[4] Vamsi Krishna Myalapalli, "An Appraisal to Overhaul Database Security Configurations", IJSRP, Volume 4, Issue 3, March 2014.

[5] Donald K. Burleson, Joe Celko. John Paul Cook, Peter Gulutzan, "Advanced SQL Database Programmer Handbook", August 2003 1st Edition, Rampant Tech press.

[6] Sanjay Mishra and Alan Beaulieu, "Mastering Oracle SQL", April 2002 1st Edition, O'Reilly Publishing.

[7] Ciro Fiorillo, "Oracle Database 11gR2 Performance Tuning Cookbook", January 2012 1st Edition, Packt Publishing.

[8] Edward Whalen, "Oracle Performance Tuning and Optimization", 1996 1st Edition, Sams Publishing.

[9] Mark Gurry, "Oracle SQL Tuning Pocket Reference", January 2002, 1st Edition, O'Reilly Publishing.

[10] Andy Oppel, "SQL Demystified", 2005, 1st Edition, McGraw-Hill Publishing.

[11] Advanced SQL Database Programmer Handbook. Donald K. Burleson, Joe Celko. John Paul Cook, Peter Gulutzan. Rampant Tech press.