# An Appraisal to Optimize SQL Queries

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

***Abstract*—Performance engineering is a vital aspect in the SQL statement crafting. SQL is a non-procedural language and gives the wrong impression that the form of the SQL statement does not matter provided it produces the accurate result set. Statements which are not cautiously designed can instigate massive performance issues. There subsists many ways of writing SQL statement to retrieve same result set, but the one which imposes minimum impact on DBMS engine is always esteemed. The proposed model is intended to serve as benchmark for the tuning SQL queries and thus improving Query performance. This paper enunciates sundry techniques that can serve as an SQL Tuning tool for programmers. Our analysis ensured reduction in time and space complexity.**

**Keywords— SQL Tuning; SQL Optimization; Database Tuning; Query Tuning; Query Optimization; Query Rewrite.**

## I. INTRODUCTION

The fascinating fact about SQL is that we retrieve data without specifying how to retrieve data (Non procedural language). Since we retrieve data with queries, it is in our hands how we retrieve data i.e. we tune our query before firing it onto DBMS engine, in order not to compromise at performance perspective.

It is guesstimated that about 80% of the problems related to database performance can be unraveled by tuning SQL statement.

This paper lists all the tuning tactics that the SQL programmers should glean at. Upcoming information of the paper is structured as follows. 2nd section deals with the associated background and related work. 3rd section deals with proposed benchmark. 4th section demonstrates experimental setup. Section 5 is concerned with Comparative analysis and finally 6th section completes the information in this paper.

## II. BACKGROUND AND RELATED WORK

*Comprehensive Database Tuning*

The comprehensive Database tuning entails entities such as Server tuning, Instance Workload tuning, Instance Object Tuning and SQL Statement tuning. Server tuning: This is concerned with tuning Network subsystem (RAID etc.) and Disk I/O in order to optimize dispatching frequency and I/O time.

Instance Workload tuning: This scenario optimizes parameters (settings), metadata (optimizer statistics) and cursor management (cursor sharing).

Instance Object Tuning: This is concerned with Row packing (disk block management), Table row location (clustering), caching and concurrency.

SQL Statement tuning: This scenario refers to optimizing SQL statements by rewriting them or enforcing static execution plans. This would engender reduction in time and space complexities.

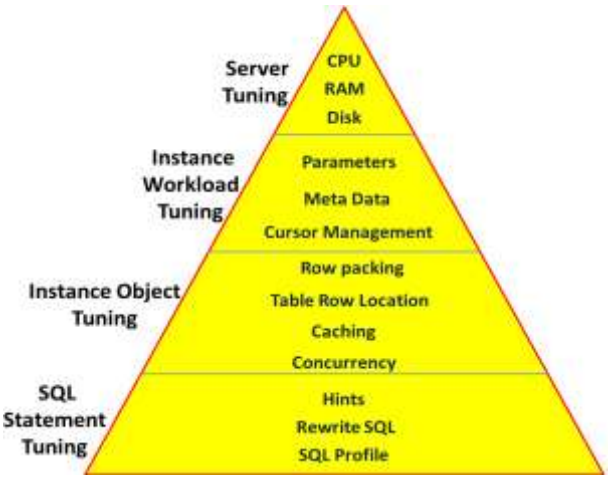


**Fig. 2.1: Comprehensive Database Tuning and Entities**

*SQL Tuning Checkpoints*

Checkpoints indicate the scenario where SQL statement performance can be determined. The following checkpoints play a major role in mitigating impact on SQL tuning.

*Rows processed*: If a query processes huge amount of rows, it imposes massive Input/output and further levies pressure on the temporary tablespace.

*Buffer get(s)*: Higher rate of buffer get(s) designate that query is resource intensive.

*Disk reads:* Higher rate of disk read obviously indicates that query is triggering much Input/output.

*Memory*: Allocation of memory for a SQL query is advantageous in recognizing queries which perform table join(s) in memory.

*Processor seconds*: It recognizes those SQL queries which exploit most of the CPU resources.

*Sort(s): Huge* Sort operations may engender a run down, specifically if those are performed on the hard disk.

*Executions:* Most recurrently processed SQL queries have to be mandatorily tweaked, as they impose great impact on overall performance.

The following portrait enunciates the scope of Database tuning.

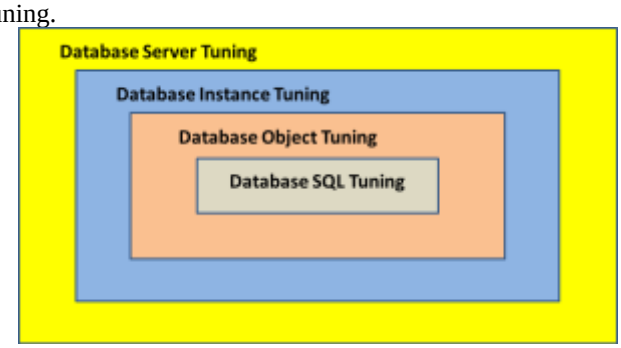


**Figure 2.2: Scope of Database Tuning**

In a nutshell the performance of SQL statement is contingent upon below factors

a) Different operations executed through the query
b) Sequence in which actions are executed
c) Algorithm utilized for executing individual operation
d) Way of retrieving data from memory/disk
*e)* The way of passing data from one operation to other through query execution.

## III. PROPOSED BENCHMARK

The proposed benchmark takes form of following tactics and thus optimizes statements. Experimental results designate that elapsed time for each tuned statement is less over non optimized statement.

*1) Thwart Full table scans:* Full table Scans(FTS) engender lot of disk I/O and drags down the database. Standard B tree indexes or Function based indexes can eliminate full table scan. Full table scan arises in the following situations.

*a)* Ommitted WHERE clause
*b)* Table statistics and Index are not updated.
*c)* WHERE clause failure to filter rows.

*2) Prevent usage of NOT IN or HAVING*: Apart from using 'HAVING' or 'NOT IN', the operator 'NOT EXISTS' with sub query will run faster.

*3) Use MINUS instead of EXISTS sub queries:* Sometimes the 'MINUS' clause in the place of 'NOT' and 'NOT IN' would outcome a speedy execution plan.

*4) SQL Analytical Functions:* Analytical function(s) have ability to perform various aggregations, on a single pass through tables, implying faster response.

*5) Rewrite NOT EXISTS sub queries and NOT EXISTS as outer joins:* Non correlated queries can be written as outer join with IS NULL.

*6) Evade the predicate LIKE:* All the time substitute 'LIKE' operator using the equality operator.

*7) Don't mix data types:* Don't use quotes if the 'WHERE' clause of column predicate is number. Always implement quotes for char index column(s).

*8) Use CASE and DECODE:* Accomplishing compound aggregations through "case" or "decode" function(s) could lessen no of times a table should to be processed.

*9) Recognizing celerity of operators:* Each operator exhibits its own speed with the index.

**Table 3.1. Celerity of Operators**

| Operator | Celerity |
|---|---|
| = | 10 |
| > >= <= < | 5 |
| LIKE | 3 |
| <> != NOT | 0 |

*10) Using Alias:* When referencing columns use fully qualified table aliases. This guides the optimizer and helps in faster retrieval.

*11) Fully Qualifying Database Object Names:* Database objects such as tables, views should be fully qualified to reduce ambiguity and guide the optimizer.

*12) Verify optimal index usage:* Ensure that index is used by the optimizer, if not apply index on another column of table.

*13) Cache small-table full-table scans:* If the table is small and frequently selects all the rows, then cache the table.

*14) Index your NULL values:* If our query frequently explores null values, then index the column with the help of function based index to index only null values.

E.g.: where mail_id IS NULL

*15) Don't use aggregation on column:* An aggregation function on a column thwarts the use of Index, unless a function based index is defined on that column.

*16) Leave column names intact:* Don't perform calculation upon index column, except a FBI (Function Based Index) is defined. *E.g. select salary + 10 from employee;*

*17) Supplant multiple queries with CASE:* In order to perform more calculations upon same rows in table, prefer CASE to multiple queries.

*18) Determine when to index:* Index a column only when the following condition holds.

Rows retrieved = Total No of rows/10

*19) Deter HAVING against WHERE:* WHERE filters rows and HAVING filters row groups. HAVING does its job, after they are grouped imposing time factor.

*20) Deter IN against EXISTS:* Exists quests for row presence whereas IN quests for actual values.

*21) Avoid IS NULL and IS NOT NULL with indexed columns:* This scenario hurts the performance instead of fructification.

*22) Deter UNION against UNION ALL:* The only difference between UNION and UNION ALL is that in later case duplicates are returned. So UNION consumes time in eliminating duplicates.

*23) Deter DISTINCT against EXISTS:* DISTINCT sorts before removing duplicate rows, so prefer EXISTS which only look for row presence.

*24) Deter CUBE against GROUPING SETS:* Grouping sets engender higher performance when compared to that of CUBE operation.

*25) Selecting adequate data type:* Choosing BIGINT over INT, even if INT is sufficient will instigate bottleneck on performance.

*26) Enforcing identical types for Primary and Foreign keys:* DBMS has to typecast when data types are not identical.

*27) Don't Over use triggers:* Trigger chaining will drag down the performance.

*28) When an Index is not invoked:* The Index is not invoked in the following cases.

a. A not equal to operator doesn't invoke index.
   WHERE age! = 10; or age <>10; or age ^= 10;
b. A function on a column doesn't invoke index, unless a function based invoke is defined.
   E.g. where upper (branch)='CS';//Thwarts Index Use
c. The 'like' operator when a prefixed wild card.
   E.g. like '%hack' (Doesn't invoke)
   E.g. like 'hack%' (Invokes Index)
d. Large IN lists.
e. When hints are used in statement.

*29) Leading index columns in WHERE clause:* Using the indexed column in where clause escalates performance, whereas using non-indexed column in where clause defeats the purpose.

*30) Prefer BETWEEN rather than AND:* Prefer BETWEEN operator in where clause rather than using AND operator.

*31) Efficient query against multiple queries:* An efficient query might work better than several queries.

*32) Restrictive expression in Join:* The joins would work faster, if literal or column is specified instead of a query.

*33) Non Column expression:* The query is driven faster if non column expressions lie on one side.

*34) Dealing with LOBs:* In order to store large object files or binary object files, place them in Operating System file system and add the respective file path into database.

*35) Make small table smaller:* Small tables reduce the rate of comparisons during join.

*36) Avoid changing the column type of Index:* The indexed column should be kept away from data type modification in order to prevent Index restructuring.

*37) Ensure that SQL standards are obeyed:*

a. Use only one case for all verbs of SQL.
b. Isolate words with single space
c. Script SQL verbs in a new line

*38) Sequence of the table names:* The DBMS engine should access the table, which eliminates most number of rows from the result set.

*39) Sequence of predicates:* The above case applies to predicates also to filters rows prior to further processing.

*40) Use IN instead of OR:* There is certitude that IN operator can be written as join. The OR operator provokes more processing by multiple query runs.

*41) Deter GROUP BY against DISTINCT:* Group By is an efficient method of filtering duplicate rows compared to DISTINCT which does sorts prior to removing duplicates.

*42) Occasionally use Hints:* Hints can be passed to optimizer, to enforce some actions, such as compelling to use a specific method or use particular index.

*43) Temporary tables escalate performance:* Temporary tables (also termed as Global temporary tables) help in faster sorting of result sets. Because initially sort results are written to temporary tables and are read back.

*44) Views accelerate the performance:* A View is a predefined stored query, which is already optimized and parsed. View obviates the tedious task of re performing the same.

*45) Avoid unnecessary columns and tables:* The lesser the number of columns (wider data) and tables, the higher the efficiency of a query.

*46) Prevent huge result set sort:* On the occasion if memory is limited, huge sorts are too expensive.

*47) Avoid indexes on frequently updated columns:* When data is updated, the overhead of updating index(s) arises.

*48) Indexing foreign keys energizes joins:* The efficiency of joins is improved on indexing foreign keys.

*49) Over Indexing is a curse:* Don't create more than four indexes for a table, which inevitably lead to performance degradation, due to frequent update and storage maintenance.

*50) Don't overlap indexes:* This case occurs when columns are indexed directly or indirectly or in a combined column approach multiple times.

*51) Deem unique indexes:* The column which entails comprehensively different values or columns used for order by or distinct should be chosen as candidate for index.

*52) Drop Indexes during bulk load:* When huge amount of data is being loaded into table, drop index and recreate it after loading gets completed.

*53) Create Function Based Indexes*: Create Function based index(s), if function(s) are frequently applied onto a column(s).

*54) Create Bit map Index:* Use Bit map index only if cardinality is low (few data values in a column). *Eg: Gender*

*55) Delete vs. Truncate:* In order to delete all the rows of table permanently, use truncate instead of delete. Because delete is a logged operation (DML) and levies significant elapsed time when table is large.

*56) Use Index Organized tables:* It's a good practice to index primary key columns, but some tables like reference tables containing fewer columns are incessantly accessed. These kind of tables should be made index organized i.e. entire table is stored in index.

*57) Recognizing the Identical statements:* Identical statements are parsed once, whereas non identical statements are parsed each time on their arrival. Parse is a resource eater,

so parsing should be reduced as much as possible. The following pairs are non-identical.

a. Different data values
   select * from t where c=1;
   select * from t where c=2;
b. Uneven spacing
   select * from t where c=1;
   select * from t where c=1;
c. Discrepancy in letter case
   select * from t where c=1;
   select * FROM t where c=1;

*58) Reusable SQL:* Bind variables do a lot in multi user environment. Most of the queries take non identical form as

select salary from e where id = 2;
select salary from e where id = 3;

Using bind variable, the query turns as

select salary from e where id=: N;

Here N is a declared bind variable.

*59) Composite Indexes:* Choose Composite index (index on more than 1 column) for higher demand of data values. If all values are indexed then table will not be queried, instead index returns the result.

*60) Char vs. Varchar2:* Prefer char over varchar2 if column width is less than 5. This avoids surplus overhead of adjustments by the SQL engine, when data is changed.

*61) Varchar vs. Varchar2:* Varchar exhibits internal behavioral change in future, so prefer varchar2 over varchar.

*62) Integers beat Chars:* Sometimes processors may process Integers faster than chars due to internal compatibility.

*63) Integers beat smallints:* Integer exhibits more compatibility than a smallint, so processors may take advantage of them.

*64) Table space management:* Locally managed table spaces offer better performance over Dictionary managed tablespaces.

*65) User management:* Ensure that user is granted default permanent tablespace (other than system) as well as temporary tablespace.

*66) Monitor Indexes:* Monitor the usage of index(s), drop them if they are not fruitful.

*67) Using Stored procedures:* Enforcing stored procedures, mitigates data sent across network and engenders certainty that the statement may be already parsed in shared SQL area. Stored procedure also ensures code sharing.

*68) Dead code elimination:* Dead code takes following form.

select name from e where 0=1 and add = 'Will not execute';

*69) Indexes with DESC:* Indexes with DESC won't fructify performance instead it hurts.

*70) Materialize aggregation for static tables:* Materialized views ensure that tables are pre-joined and pre-summarize the data. This would bolster data warehouses.

*71) Use DECODE rather than multiple evaluations:* Decode offers better performance than using multiple selections with operators.

*72) Reverse Key Indexes:* Reverse index have dissimilar beginning values and use different blocks in b-tree index structure. This mitigates concurrency on index blocks. We can't reverse bitmap index.

*73) Invisible Indexes:* Optimizer can't see Invisible indexes, so we can test query performance with and without index and determine whether to maintain or drop it.

*74) Rewriting complex sub queries with help of dynamic table:* WITH clause with its divide and conquer nature, refers sub query by name, considering it as dynamic table on fly. WITH clause can also be applied to materialized views.

*75) Peculiar full table scans:* Even though we index column(s) of a table, sometimes FTS might be fast compared to Indexed scans. This happens when the number of rows returned is higher.

*76) Transform Outer joins to Inner joins:* Some queries exhibit the certitude of rewriting outer join as efficient inner join. The latter case offers more performance.

*77) Accessing row using Row id:* Accessing row using ROWID is fastest compared to many other methods.

*78) Thwart Cartesian joins:* Cartesian or Cross joins degrade performance since they drag all the table(s) data in all possible ways.

*79) Table partitioning:* A large database environment can be tuned by partitioning tables. This partitioned tables each in turn can have individual indexed defined on them.

*80) Thwarting improper view usage*: If query uses joins on views, it might impose considerable time for execution.

*81) Enforcing Table Compression:* Tables, table partitions and materialized views can be compressed. Compression increases query performance and reduces storage.

*82) Enabling SQL Trace:* SQL trace allows tracing execution of statements. It tracks

a. Cache hit ratio
b. Statement execution plan
c. Number of logical and physical reads
d. Cost of CPU and elapsed time
e. Parse and execute count of statement

If an application begets more dynamic SQL, then tracing is ideal for tuning statements.

*83) Index key compression:* Index key compression engenders enhanced performance apart from removing duplicates and saving space.

*84) Enabling parallelism for query:* Hints can be used to exploit parallelism rate.

*85) Maximize speed of data load:* If a large amount of data needs to be transferred from one table to another.

Create table <source_tbl> as select * from <dest_tbl>;

*86) Implementing Virtual columns:* Instead of calculating value using other column(s), creating a virtual or dynamic

column offers better performance. Virtual column can be indexed, referred in where clause and is permanently defined in Database.

*87) Creating temporary tablespaces:* Temporary tablespaces help in carrying out sorting operations. They are used during index creation, ORDER BY and GROUP BY.

*88) Thwarting Balloon tactic:* Balloon tactic is the scenario where individual simple statements are converted into a single or very few complex sql statement(s). Coding simple queries against complex query ensures easier optimization and maintenance.

*89) Rebuild Indexes:* Rebuild the indexes regularly in order to reduce intra block fragmentation.

*90) Clustering the Indexes:* Cluster is a group of tables sharing common columns. They are stored in same area of disk and thus ensuring faster retrieval. To attain this, a cluster should be created and tables should be placed in that cluster after indexing the cluster.

*91) Avoid Schema level programs:* Avoid programs at schema level, instead group related code into packages. This will ensure performance as well as security.

*92) Indexing GTTs:* Sometimes indexing Global Temporary Tables may increase performance when compared to scenario with non-indexed Global temporary table.

*93) Don't overuse Bitmap Indexes:* Bitmap Indexes reduce concurrency, since it exhibits row level locking. But large number of distinct values does not engender this situation.

*94) Avoid Recursive Functions:* Recursive functions are memory hunger i.e. for each iteration variable, parameter copy and pointer onto current instruction has to be maintained and should be returned on its turn.

*95) Avoiding Row chaining:* A chained row is the one that can't fit in one block, spanning across multiple blocks. Increased block size leads to faster updating, less disk space and RAM consumption.

*96) Normalizing the data:* As the data goes through Normal forms it becomes faster, clean, organized and better. But over normalization would trigger greater space requirement.

*97) Replace PL/SQL with SQL:* There might be some occasions where SQL may be right choice against PL/SQL.

*98) Using PL/SQL Data types:* The existing data types supported by Oracle are also supported by PL/SQL. Efficient code is ensured if native PL/SQL data types are used. Specifically NUMBER and INTEGER data types are designed for portability not for performance.

*99) Partitioning tables:* Vertical Partition (column partitioning) decreases the amount of data a SQL query needs to process. Horizontal Partitioning partitions the table by data value, which decreases the amount of data a SQL query needs to process.

*100) Querying the v$sql:* The dynamic view v$sql records statements that use resources highly.

*101) Using AUTOTRACE:* Auto trace explains cost involved in processing query. It results time elapsed, cost of CPU and number of bytes accessed. Issue the below statement and continue with your normal DML statements.

SQL> set autotrace on explain

## IV. EXPERIMENTAL SETUP

Some important practices are explained, that would serve as an exemplary. Experimental results indicate that performance is increased.

1. select name from emp;

or

select name from emp where id = 532;

*Instead of*

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

11. select name from scott.emp;
13. create table tablename (col1,col2,...) cache;
19. select branch, avg(sal) from employ where branch in ('CS','IT') group by branch;

*Instead of*

select branch, avg(sal) from employ group by branch having branch in ('CS','IT');

20. select id, name from employ outer where exists (select 1 from sales inner where inner.id = outer.id);

*Instead of*

select id, name from employ where id in (select id from sales);

24. select branch, count(*) "Count", grouping(branch) from student group by cube(branch) order by branch;

*Instead of*

select branch, count(*) Count from student group by cube(branch) order by branch;

30. where price between max (price) and min (price)

*Instead of*

where price >= max (price) and price <= min (price)

31. select min(sal) min, max(sal) max, avg(sal) avg from emp;

*Instead of*

select min(sal) from emp; select max(sal) from emp;
select avg(sal) from emp;

32. select col1 from t1,t2 where t1.col1=32;

*Instead of*

select col1 from t1,t2 where t1.col1 = (t2.col2)

33. *where salary < 2500;*

*Instead of*

where salary + 1000 < 3500;

43. create global temporary table <table name> (column list);
53. create index <index_name> on <table_name> (function(column name));

Ex: create index extra on emp (salary+incentive);
select name from emp where salary+incentive > 20000;

54. create bitmap index <indx_name> on tbl_name (col_list);
55. truncate table <table name> reuse storage;
truncate table <table name> drop storage; //Default
56. create table <table_name> (col_list) organization index;
Ensure that primary key is enforced or else the above query will fail.

64. create tablespace tools datafile
'c:\ora01\dbfile\INVREP\tools1.dbf' size 100m
extent management local uniform size 128k
segment space management auto;
65. create user <user_name> identified by <passwd>
default tablespace <tablespace_name>
temporary tablespace <tablespace_name>;
66. alter index indx1 monitoring usage;
select index_name, table_name, monitoring, used from
v$object_usage;
69. select name from emp order by id desc;
// Query does not take advantage of index, unless the index is a reverse index.
70. create materialized view Strength build immediate refresh
complete as select branch, count(*) “Count” from
sys.student group by branch;
71. select branch, decode (branch, 'CS','Computer Science',
'IT', 'Info. Tech.') as “Full Form” from student;
*Instead of*
select branch, case branch when 'CS' then 'Computer
Science' when 'IT' then 'Info. Tech.' end as FullForm from
student;
72. create index on idx on product (prod_id) reverse;
To alter an index to remove reverse key
alter index <index_name> rebuild noreverse;
To alter an index to a reverse key
alter index <index_name> rebuild reverse;
73. alter index <index_name> invisible;
select index_name, visibility from dba_indexes where
index_name='<index_name>';
alter index <index_name> visible;
74. In the below query neither port_booking nor densest_port is a database object.
SQL> with
port_bookings as ( select p.port_id, p.port_name,
count(s.ship_id) ct from ports p, ships s
where p.port_id = s.home_port_id
group by p.port_id, p.port_name ),
densest_port as ( select max max_ct from port_bookings )
select port_name from port_bookings
where ct = (select max_ct from densest_port);
78. select * from student natural join hostel;
*Instead of*
select * from student, hostel;
81. create table test compress as select * from sh.sales;
82. alter session set sql_trace = true;
83. create index comp_indx on employ (id) compress;
84. select /*+ parallel (test,4) */ * from test;
Instructing optimizer to make work done with 4 parallel processes.
85. create table <table name> as select * from <target table>;
86. create table student( name varchar2(12), id number, branch as (case when id >= 500 then 'CS' when id < 500 then 'Other' end));
87. create temporary tablespace temptablespace
tempfile c:\orcl\tempf.dbf size 1G;
89. alter index <index_name> rebuild online parallel;
90. create cluster clustered_id (id varchar2(10));
create index indxed_clstr on cluster clustered_id;
create table exam(id varchar2(10) primary key,
marksnumber(3)) cluster clustered_id(id);

## V. OUTPUT SCREENS

Some of the tactics which play critical role are snapped and represented beneath.

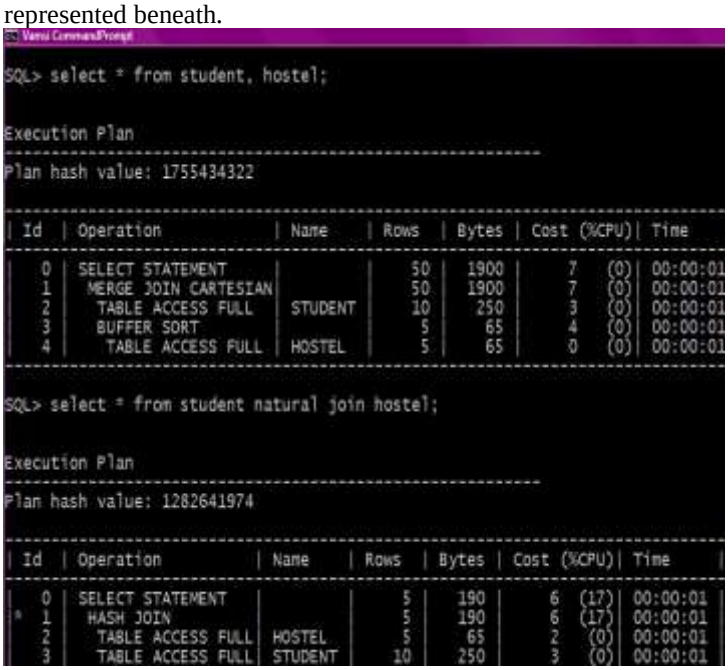


**Fig. 5.1: Cross Join vs. Indexed Join (Tactic #78)**

Above figure represents Cross Join which touches each and every block of the disk leading to excessive disk I/O.

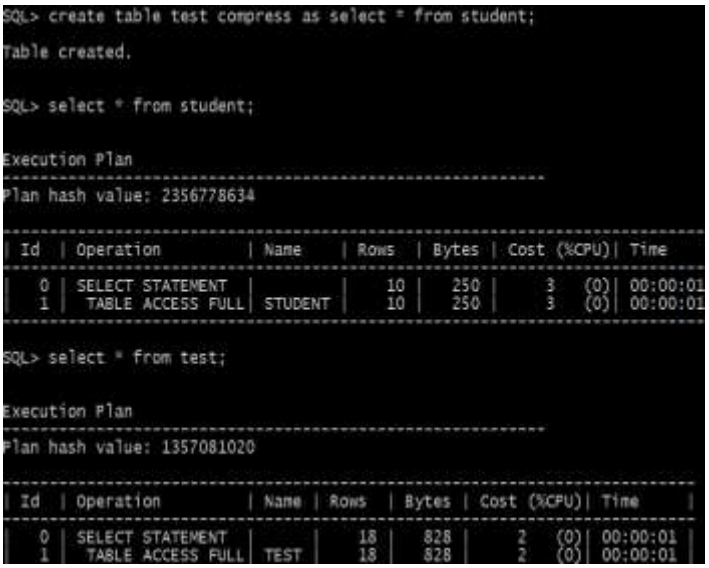


**Fig. 5.2: Normal Table vs. Compressed Table Access. (Tactic #81)**

Above figure proves that data access on compressed table is faster compared to access on normal table.

## VI. COMPARATIVE READING

The benchmark assists the SQL programmers, in many possible perspectives. The efficiency of the query can be determined by examining the system against the proposed benchmark, there by reckoning anticipated performance impact. The criticality for each practice has also been specified to analyzing the SQL statement for minimum performance anticipation. The statements exhibit more efficiency, since all the access points are overhauled against the optimization methodologies. Enforcing permanent SQL tuning, regardless of changes to environment ensures performance engineering i.e. reduction in time and space complexities.

The following results are obtained after tuning SQL statements.

| Tactic Stage | T.A.T | Bytes Accessed | CPU Cost | Runtime Memory | Sorts | Fetches | Disk Reads | B.G | R.P | E.T | P.R.B | Loads |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Before Tuning | 1.81 | 10106 | 308 | 73016 | 7 | 186 | 46 | 1098 | 974 | 580424 | 376382 | 32 |
| After Tuning | 1.12 | 5779 | 150 | 68772 | 0 | 50 | 1 | 273 | 216 | 117011 | 812 | 26 |

**Figure 6.1: Statistics Before and After Tuning**

Turn Around Time (TATi) is the sum of time taken by an individual statement at to get processed. TAT is the sum of time consumed by all queries specified above and is given by

$$TAT = TAT_1 + TAT_2 + \ldots + TAT_n.$$

Similarly rest of the parameters can be derived by

(Bytes Accessed) $BA = BA_1 + BA_2 + \ldots + BA_n.$

(CPU Cost) $C = C_1 + C_2 + \ldots + C_n.$

(Runtime Memory) $RM = RM_1 + RM_2 + \ldots + RM_n.$

(Sorts) $S = S_1 + S_2 + \ldots + S_n.$

(Fetches) $F = F_1 + F_2 + \ldots + F_n.$

(Disk Reads) $DR = DR_1 + DR_2 + \ldots + DR_n.$

(Buffer Gets) $BG = BG_1 + BG_2 + \ldots + BG_n.$

(Rows Processed) $RP = RP_1 + RP_2 + \ldots + RP_n.$

(Elapsed Time) $ET = ET_1 + ET_2 + \ldots + ETn.$

(Physical Read Bytes) $PRB = PRB_1 + PRB2 + \ldots + PRBn.$

(Loads) $L = L_1 + L_2 + \ldots + L_n.$

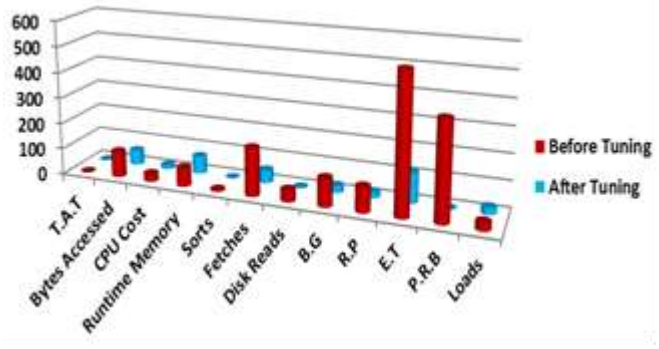


**Figure 6.2: Statistics Before and After Tuning Statements**

The above table is graphically represented as follows which delineates statistics before tuning vs. after tuning a statement. All the issues such as Elapsed Time, Number of Bytes accessed, CPU Cost and Number of Rows Processed are fructified. Amongst Tuning instigated a drastic change in Elapsed Time i.e. Time complexity as per reckoned results.

It can be perceived that a far difference exists among factors, before and after tuning just for a result set of 18 rows. Then it can be anticipated that how much performance discrepancy occurs in a distributed or complex database environment that entails millions of rows.

## VII. CLOSING COMMENTS

Performance tuning needs every bit of our ingenuity, perseverance and knowledge to keep out of harm's way. Enforcing permanent SQL tuning, regardless of changes to environment ensures performance engineering i.e. reduction in time and space complexities. Each invalid and unusable object should be investigated, identified and repaired. Indexes play a crucial role in performance, but they hurt the performance if table or view is moved or during over indexing.

This paper gazes at uplifting existing tuning engineering methodologies. Despite the innate complexity of SQL tuning, there are general guidelines that every developer should follow in order to escalate the overall performance of their systems.

The SQL query tuning may seem deceptively simple but constitutes to 80 percent of SQL tuning. This doesn't necessitate a thorough understanding of the internals of SQL or DBMS engine. Simply the query which imposes minimal disk I/O will eventually stay at optimized side. Apart from Disk I/O the operations such as huge disk sorts, wait events will engender the performance to degrade.

**Future Work:** We extend this work by developing a tool that optimizes a given query and informs user about poor coding practices. Further we bring out more appraisals and methodologies by testing and discussing with several performance engineers, DBAs and other experts.

## VIII. REFERENCES


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

[2] Vamsi Krishna Myalapalli, Pradeep Raj Savarapu, "High Performance PL/SQL Programming", 1st IEEE International Conference on Pervasive Computing, Pune, India, January 2015.

[3] Vamsi Krishna Myalapalli, "Accelerating SQL Queries", IEEE International Conference.(Unpublished).

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

[5] Vamsi Krishna Myalapalli, "An Appraisal to Overhaul Database Security Configurations", International Journal of Scientific and Research Publications, Volume 4, Issue 3, March 2014.

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

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

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

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