**Tutorial Week 6: SQL Tuning**

Objective is to understand how SQL tuning done in Oracle. To start first finding how your SQL statement is executed and how much time it is taken for execution. You have to use the “**SH**” schema or user.

Now from the same terminal you started Oracle, typing **SQLPLUS** to start.

In your current session, type following:

**set echo on**

**set timing on**

**set autotrace on**

After that before you do each part of this tutorial, flush the cached data using:

**alter system flush shared\_pool;**

**alter system flush buffer\_cache;**

To create an index: <http://www.techonthenet.com/oracle/indexes.php>

Tutorial on how to use ***hints*** in oracle: <http://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements006.htm#SQLRF50702>

**Q1. Type following select statements and compare their execution plans and time. What is difference you find?**

1. select cust\_first\_name, cust\_last\_name from customers where cust\_id = 1030;
2. select cust\_first\_name, cust\_last\_name from customers where cust\_id <> 1030;
3. select cust\_first\_name, cust\_last\_name from customers where cust\_id < 20000;
4. select cust\_first\_name, cust\_last\_name from customers where cust\_id between 70000 and 80000;

The results of these queries show that the Oracle optimizer CAN use an index for EQUALITY search (=) and CANNOT use the NOT EQUAL (<>) search.

Moreover, the optimizer sometimes has a problem with the index.

For example, the database may be unable to use an index when handling a range predicate: unbounded range (<=, <, >, and >=). It means that the database will perform a full table scan, ignoring your index.

**Q2. Create an index on the** **CUST\_CREDIT\_LIMIT column of the CUSTOMERS table. After that, compare and discuss the execution plan and timing of following queries**

1. select cust\_id from customers where cust\_credit\_limit\*1.10=11000;
2. select cust\_id from customers where cust\_credit\_limit=11000/1.10;

Execute following SQL statement to create an index on the CUST\_CREDIT\_LIMIT column of the CUSTOMERS table:

Create index CUST\_CREDIT\_INDX on CUSTOMERS (CUST\_CREDIT\_LIMIT);

The results of these queries show that although the CUST\_CREDIT\_LIMIT column is indexed, this index cannot be used appropriately in the statement (a) because of an expression (**\*1.10**) on the CUST\_CREDIT\_LIMIT column.

To improve performance, the indexed column should appear clean in the WHERE clause (avoid using function or expression).

By rewriting the query, the index is used in the statement (b) because of the CUST\_CREDIT\_LIMIT column is clean.

**Q3. Create another index on the CUST\_LAST\_NAME column of the CUSTOMERS table. After that, compare the execution plan and timing of following queries**

1. select cust\_id from customers where substr(cust\_last\_name,1,1) = 'S';
2. select cust\_id from customers where cust\_last\_name like 'S%';
3. select cust\_last\_name from customers where cust\_last\_name like 'S%';

Execute following SQL statement to create an index on the CUST\_LAST\_NAME column of the CUSTOMERS table:

Create index CUST\_LNAME\_INDX on CUSTOMERS (CUST\_LAST\_NAME);

(a) Although the CUST\_LAST\_NAME column is indexed, this index cannot be used in an appropriate manner in the statement (a) because of the function (**SUBSTR**) on the CUST\_LAST\_NAME column. Optimiser will use ‘Fast Full Index Scan’ which is equivalent to a ‘Full Table Scan’. By rewriting the query (a) as (b) can allow index to be used.

Queries (a) and (b) are logically equivalent, but in ( c ) selected column is same as search column. Therefore, you will notice change in execution plan.

In summary, you need to understand that the indexed column should be clean. As soon as you apply any function, index usage is impossible.

**Q4. Check the execution plan and reason why index is not used.**

1. select cust\_id from customers where cust\_last\_name like '%S%'

The index is unusable, because the search pattern starts with a wildcard. This makes sense. The index is useless if you do not specify a leading part for the search.

1. select cust\_last\_name from customers where cust\_id like '7%'

This is because the CUST\_ID column is a numeric column, so the optimizer must apply an implicit data type conversion and rewrites the WHERE clause to read. This will lead to full scan of index. In other words, index will become unusable. To solve the problem in this query, you need to create a new index for TO\_CHAR(cust\_id) and rewrite the query like following:

select cust\_last\_name from customers where TO\_CHAR(cust\_id) like '7%'

**Q5. In ‘ORDER BY’ clause, in general there is sorting done on all the rows. Try to run following select statement and again check the execution plan:**

select cust\_first\_name, cust\_last\_name, cust\_credit\_limit from customers order by cust\_credit\_limit;

Now create an appropriate index on a column and again run above select statement. Do you find any change in performance and execution plan?

You create an index on the CUST\_CREDIT\_LIMIT column using the following statement.

Create index CUST\_CREDIT\_INDX on CUSTOMERS (CUST\_CREDIT\_LIMIT);

Create an index on CUST\_ID column of CUSTOMERS table **if it does not.** Now execute following SQL statement and compare the execution plan of this query with above.

select cust\_first\_name, cust\_last\_name, cust\_credit\_limit from customers order by cust\_id;

In general index is used when you are doing ‘order by’ on an indexed column. You will notice that the index is used in later case but not in previous case. The difference between both queries is that the CUST\_ID column is mandatory (NOT NULL) and the CUST\_CREDIT\_LIMIT column is not. The index on CUST\_CREDIT\_LIMIT cannot be guaranteed to contain an entry for each customer, because NULL-values are not stored in regular indexes. Thus, only way to find all rows containing NULL values is to perform a full-table scan.

**Q6. Compare the execution plan for following statements and explain the difference:**

select max(cust\_credit\_limit) from customers;

select max(cust\_credit\_limit\*2) from customers;

select max(cust\_credit\_limit+1000) from customers;

This shows that an index can be useful to retrieve a maximum value (and a minimum value). If no index is available, the optimizer must scan the full table and perform a sort. However, in some databases, for second query index will not be used but index is used in last one. The reason is that optimizer can find equivalent query for last query so that index can be used but optimizer cannot find for second query.

*max(cust\_credit\_limit + :x) is always equivalent to*

*max(cust\_credit\_limit ) + :x*

*max(cust\_credit\_limit \* :x) is not always equivalent to max(cust\_credit\_limit ) \* :x*

**Q7. Compare the execution plan and time of following two statements.**

select count(\*) from products p where prod\_list\_price<1.15 \* (select avg(unit\_cost) from costs c where c.prod\_id=p.prod\_id);

select count(\*) from products p, (select prod\_id, avg(unit\_cost) ac from costs group by prod\_id) c where p.prod\_id=c.prod\_id and p.prod\_list\_price <1.15\*c.ac;

Second query performs better as in the first query as subquery runs for every row found in the query. You may notice, your optimizer may rewrite the query to have join rather than subquery.

**Q8. Analyse following SQL statement and identify the best join operation (and join sequence, when relevant), using the hints (USE\_MERGE, USE\_NL, USE\_HASH) to instruct the optimizer.**

select c.cust\_last\_name, c.cust\_year\_of\_birth, co.country\_name from customers c, countries co where c.country\_id = co.country\_id and co.country\_region = 'Americas';

You have to use hint like following

select /\*+ USE\_MERGE(c co) \*/ c.cust\_last\_name, c.cust\_year\_of\_birth, co.country\_name from customers c, countries co where c.country\_id = co.country\_id and co.country\_region = 'Americas';

For other hints you have to do in similar way. While comparing execution plan, compare their ‘cost’ to see which one is better than other.

**Q9. Analyse the SQL statement**

select c.cust\_last\_name, s.time\_id, s.prod\_id from customers c, sales s where c.cust\_id <> s.cust\_id and s.prod\_id = 2595 and s.time\_id = '01-JAN-98';

Which join operations can be used to execute this join? Experiment with different join orders by using an ORDERED hint, then try a LEADING hint, and find the best choice. There is a significant difference in performance? Do the same with different join operations.

Nested loop is only possibility here due to non-equality. Here you may find, for different types of join hints, oracle may still use most appropriate one which may be different from the one specified in join hint. However, you may find difference when using ORDERED and LEADING. Having CUSTOMERS as outer table will results in more cost, so in this case, there will be very high chance for SALES to be an outer table for join.

**Q10. Consider following SQL statement**

select c1.cust\_first\_name, c1.cust\_last\_name, c1.cust\_year\_of\_birth from customers c1 where c1.cust\_year\_of\_birth = (select max(c2.cust\_year\_of\_birth) from customers c2 where c1.country\_id = c2.country\_id);

This statement retrieves the customers with the oldest birth year in every country. This type of statement can be found in many real-life situations. What is happening in the execution plan? Are the results satisfactory?

Create an index on the COUNTRY\_ID column, and measure the performance improvement. Is creating the index a better choice? Can you rewrite the query to use join operation? Is there any performance improvement.

***Hint: Move the correlated subquery to the FROM clause*.**

You create an index on the COUNTRY\_ID column using the following statement.

Create index CUST\_COUNTRY\_INDX on CUSTOMERS (COUNTRY\_ID);

Query can be rewritten like following:

select c1.cust\_first\_name, c1.cust\_last\_name, c1.cust\_year\_of\_birth from customers c1, (select max(c2.cust\_year\_of\_birth)as maxyear, c2.country\_id as country from customers c2 group by c2.country\_id) temp where c1.cust\_year\_of\_birth=temp.maxyear and c1.country\_id=temp.country;

**In following questions, try to understand different possibilities you need to consider before deciding for which column index needs to be created. When bitmap index to be used? When combined index to be used?**

**Q11. Create indexes on the following columns: cust\_gender, cust\_postal\_code**, **cust\_credit\_limit**

Execute following SQL statements to create three indexes:

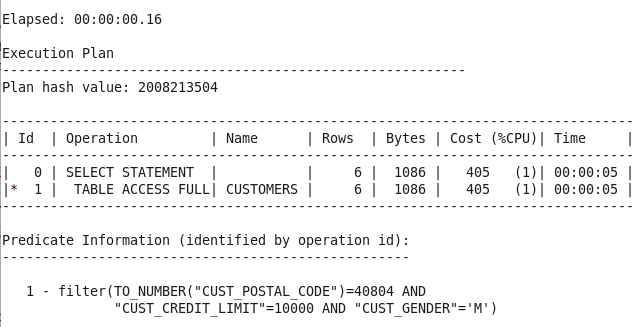
Create Index CUST\_GENDER\_INDX On Customers(cust\_gender);

Create Index CUST\_POST\_INDX On Customers(cust\_postal\_code);

Create Index CUST\_CREDIT\_INDX On Customers(cust\_credit\_limit);

The following statement contains WHERE clause with three predicates. Execute this statement, and take notes about the indexes used, the cost of the execution plan, and the amount of I/O performed.

select c.\* from customers c where cust\_gender = 'M' and cust\_postal\_code = 40804 and cust\_credit\_limit = 10000;



Run above statement again with different indexes by giving HINT as follow and take notes about the performance results.

select /\*+ INDEX (c CUST\_CREDIT\_INDX) \*/ c.\* from customers c where cust\_gender = 'M' and cust\_postal\_code = 40804 and cust\_credit\_limit = 10000;

Elapsed: 00:00:00.18

Execution Plan

----------------------------------------------------------

Plan hash value: 3171069297

-----------------------------------------------------------------------------------------------------

| Id | Operation | Name Rows | Bytes | Cost (%CPU) | Time |

-----------------------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 6 | 1086 |452 (1) | 00:00:06 |

|\* 1 | TABLE ACCESS BY INDEX ROWID | CUSTOMERS| 6 | 1086 |452 (1) | 00:00:06 |

| 2 | BITMAP CONVERSION TO ROWIDS| | | | | |

| 3 | BITMAP AND | | | | | |

|\* 4 | BITMAP INDEX SINGLE VALUE | CUST\_CREDIT\_INDX | | | | |

| 5 | BITMAP CONVERSION FROM ROWIDS| | | | | |

|\* 6 | INDEX RANGE SCAN | CUST\_GENDER\_INDX | | | 51 (0) | 00:00:01 |

-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter(TO\_NUMBER("CUST\_POSTAL\_CODE")=40804)

4 - access("CUST\_CREDIT\_LIMIT"=10000)

6 - access("CUST\_GENDER"='M')

Run above statement again with different indexes by giving HINT as follow and take notes about the performance results.

select /\*+ INDEX (c CUST\_GENDER\_INDX CUST\_POST\_INDX CUST\_CREDIT\_INDX) \*/ c.\* from customers c where cust\_gender = 'M' and cust\_postal\_code = 40804 and cust\_credit\_limit = 10000;

Elapsed: 00:00:00.09

Execution Plan

----------------------------------------------------------

Plan hash value: 743140819

----------------------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

----------------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | 6 |1086 | 217 (0) | 00:00:03 |

|\* 1 | TABLE ACCESS BY INDEX ROWID| CUSTOMERS | 6 |1086 | 217 (0) | 00:00:03 |

|\* 2 | INDEX FULL SCAN | CUST\_POST\_INDX| 89 | | 133 (0) | 00:00:02 |

----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter("CUST\_CREDIT\_LIMIT"=10000 AND "CUST\_GENDER"='M')

2 - filter(TO\_NUMBER("CUST\_POSTAL\_CODE")=40804)

Drop above indexes and create a new combined index and run above statement without any hints again. Compare the performance results with previous ones. Which is best?

Execute following SQL statement to create a new combined index:

Create Index C\_INDX On Customers(cust\_gender, cust\_postal\_code, cust\_credit\_limit);

Elapsed: 00:00:00.09

Execution Plan

----------------------------------------------------------

Plan hash value: 1353691094

-----------------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

-----------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 7 | 1267 | 93 (0) | 00:00:02 |

| 1 | TABLE ACCESS BY INDEX ROWID| CUSTOMERS | 7 | 1267 | 93 (0) | 00:00:02 |

|\* 2 | INDEX RANGE SCAN | C\_INDX | 6 | | 87 (0) | 00:00:02 |

-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("CUST\_GENDER"='M' AND "CUST\_CREDIT\_LIMIT"=10000)

filter(TO\_NUMBER("CUST\_POSTAL\_CODE")=40804 AND "CUST\_CREDIT\_LIMIT"=10000)

**Q12. Oracle provides several index types.**

Hint: to create an index.

*http://docs.oracle.com/cd/B28359\_01/appdev.111/b28843/tdddg\_creating.htm*

*http://www.orafaq.com/wiki/Bitmap\_index*

In this exercise will try to see how they differ:

Execute following SQL statement to see all existing indexes:

SELECT index\_name FROM user\_indexes;

1. Drop previously created indexes in this tutorial. Create two bitmap indexes on the following columns of the CUSTOMERS table:

cust\_year\_of\_birth

cust\_credit\_limit

Execute following SQL statements to drop indexes:

Drop Index CUST\_YOB\_INDX;  
Drop Index CUST\_CREDIT\_INDX;

Execute following SQL statements to create two bitmap indexes:

Create Bitmap Index CUST\_YOB\_INDX On Customers(cust\_year\_of\_birth);

Create Bitmap Index CUST\_CREDIT\_INDX On Customers(cust\_credit\_limit);

Execute following SQL statements and compare their execution plan and performance:

SELECT /\*+ INDEX\_COMBINE(c) \*/ c.\* FROM customers c

WHERE c.cust\_year\_of\_birth = 1953 OR c.cust\_credit\_limit = 10000;

The indexes are used because the INDEX hint is applied to force the optimizer to use it.

SELECT c.\* FROM customers c WHERE c.cust\_year\_of\_birth = 1953 OR c.cust\_credit\_limit = 10000;

The indexes are not used.

1. Drop all the indexes on the CUSTOMERS table except its primary key index.   
   After this, create a concatenated B-tree index on the following columns of the

CUSTOMERS table, and in the order here:

cust\_last\_name

cust\_first\_name

Check all existing indexes of the CUSTOMERS table and drop all except its primary key index.

Drop Index column\_names;

Compare results of following query before and after creating the index:

SELECT c.cust\_last\_name, c.cust\_first\_name FROM customers c;

A composite index, also called a concatenated index, is an index on multiple columns in a table. Columns in a composite index should appear in the order that makes the most sense for the queries that will retrieve data and need not be adjacent in the table.

Execute following SQL statement to create a concatenated B-tree index:

Create Index CUST\_NAME\_INDX On Customers(cust\_last\_name, cust\_first\_name);

The results of the above query show that an index fast full is used when the concatenated B-tree index created.

Delete the concatenated B-tree index and create two new indexes:

Create Index CUST\_LAST\_NAME\_INDX On Customers(cust\_last\_name);

Create Index CUST\_FIRST\_NAME\_INDX On Customers(cust\_first\_name);

The results of the above query show that two above indexes are not used.

Now run following statement and again compare the results:

SELECT /\*+ INDEX\_JOIN(c CUST\_LAST\_NAME\_INDX CUST\_FIRST\_NAME\_INDX) \*/ c.cust\_last\_name, c.cust\_first\_name FROM customers c;

The indexes are used because the INDEX\_JOIN hint is applied to force the optimizer to combine and use it.

1. (i) Drop all the previously created indexes on the CUSTOMERS table except its primary key index. Create one B-tree index on the following column of the CUSTOMERS table: cust\_credit\_limit

Drop Index column\_names;

Create Index CUST\_CREDIT\_INDX On Customers(cust\_credit\_limit);

Execute the following query, note down the plan and performance statistics.

SELECT count(\*), cust\_credit\_limit FROM customers WHERE cust\_credit\_limit = 10000 GROUP BY cust\_credit\_limit;

Elapsed: 00:00:00.01

Execution Plan

----------------------------------------------------------

Plan hash value: 636859273

-----------------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

-----------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | 4 | 14 (0) | 00:00:01 |

| 1 | SORT GROUP BY NOSORT | | 1 | 4 | 14 (0) | 00:00:01 |

|\* 2 | INDEX RANGE SCAN | CUST\_CREDIT\_INDX | 6938 | 27752 | 14 (0) | 00:00:01 |

-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1. - access("CUST\_CREDIT\_LIMIT"=10000)

(ii) Drop all the indexes on the CUSTOMERS table except its primary key index. Then, create one bitmap index on the following column of the CUSTOMERS table: cust\_credit\_limit

Drop Index column\_names;

Create Bitmap Index CUST\_CREDIT\_INDX On Customers(cust\_credit\_limit);

Execute the following query, note down the plan and performance statistics.

SELECT count(\*), cust\_credit\_limit FROM customers WHERE cust\_credit\_limit = 10000 GROUP BY cust\_credit\_limit;

Elapsed: 00:00:00.04

Execution Plan

----------------------------------------------------------

Plan hash value: 2388706395

------------------------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

------------------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | 4 | 1 (0) | 00:00:01 |

| 1 | SORT GROUP BY NOSORT | | 1 | 4 | 1 (0) | 00:00:01 |

| 2 | BITMAP CONVERSION COUNT | | 6938 | 27752 | 1 (0) | 00:00:01 |

|\* 3 | BITMAP INDEX SINGLE VALUE | CUST\_CREDIT\_INDX | | | | |

------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

3 - access("CUST\_CREDIT\_LIMIT"=10000)

(iii) Compare the results of (i) and (ii). Which index is better and why?

A B-Tree index is used in columns with high data sparsity (many different values relative to the total number of rows), while a bitmap index is used in columns with low data sparsity.