How To Write Efficient SQL Queries with Tips N Tricks

Greens Technologys

1. Use the appropriate Oracle Optimizer

The ORACLE optimizer has three primary modes of operation:

RULE
COST &
CHOOSE

To set the optimizer goal, you can specify RULE, COST, CHOOS or FIRST_ROWS for the OPTIMIZER_MODE parameter in the init.ora file at session level. You can override the optimizer's default operations at both the query (using Hints) & session level (using ALTER SESSION command).

The **Rule-Based Optimizer (RBO)** evaluates possible execution paths & rates the alternative execution paths based on a series of syntactical rules.

To make use of **Cost-Based Optimizer (CBO)**, you need to make sure that you run the **analyze** command frequently enough to generate statistics about the objects in your database to accurately reflect the data.

Setting OPTIMIZER_MODE to **CHOOSE** invokes CBO, if the tables have been analyzed and the RBO, if the tables have not been analyzed.

By default, ORACLE uses CHOOSE optimizer mode. To reduce the potential for unplanned full table scans, you should avoid using the CHOOSE option; either use the RBO or the CBO throughout your database.

2. Operations That Access Tables

ORACLE performs two operations for accessing the rows of a table:

TABLE ACCESS FULL

A full table scan sequentially reads each row of a table. To optimize the performance of a full table scan, ORACLE reads multiple blocks during each database read.

A full table scan is used whenever there is no **where** clause on a query.

TABLE ACCESS BY ROWID

To improve the performance of table accesses, you can use this operation that allows you to access rows by their RowID pseduocolumn values. The RowID records the physical location where the row is stored. ORACLE uses indexes to correlate data values with RowID values – and thus with physical locations of the data. And because indexes provide quick access to RowID values, they help to improve the performance of queries that make use of indexed columns.

3. Share SQL Statements

ORACLE holds SQL statements in memory after it has parsed them, so the parsing and analysis won't have to be repeated if the same statement is issued again. The single shared context area in the shared buffer pool of the System Global Area (SGA) is shared by all the users. Thus, if you issue a SQL statement, sometimes known as a cursor, that is identical to a statement another user has issued, you can take advantage of the fact that ORACLE has already parsed the statement and figured out the best execution plan for it. This represents major performance improvements and memory savings. But the cache buffering is applied only to simple tables, the multiple table queries & the joins are never cached.

The DBA must set the appropriate INIT.ORA parameters for the context areas. The larger the area, the more statements can be retained there and the more likely statements are to be shared.

Whenever you issue a SQL statement, ORACLE first looks in the context area to see if there is an identical statement there. Unfortunately, ORACLE does an extra string comparison on the new statement and the contents of the context area. To be shared, the SQL statements must truly be the same: carriage returns, spaces, and case (upper vs lower) all affect the comparison.

In order to qualify for this matching condition, all three of the following rules must be true to make use of the parsed statement in the shared area.

1. There must be a character-by-character match between the statement being examined and one already in the shared pool.

   Note:
   Before this comparison is performed, Oracle applies an internal algorithm using the new statement. It then checks the results against values of statements already in the pool. If the new value matches one already there, then only the string comparison outlined in Rule 1 is performed.

   For e.g.

   ```
   SELECT * FROM EMP;
   ```

   is not the same as any of these:

   ```
   SELECT * from EMP;
   Select * From Emp;
   SELECT      *       FROM EMP;
   ```

The following statements do not qualify because the first SQL statement is split over two lines whereas the second is on a single line.

a. Select pin from person where last_name =
'LAU';

b. Select pin from person where last_name = 'LAU';

2. The objects being referenced in the new statement are exactly the same as those objects in a statement that has passed the comparison in Rule 1.

For e.g.

Assume that for this example, the users have access to the objects as shown below:

USER	OBJECT NAME	ACCESSED VIA
Jack	sal_limit	private synonym
	work_city	public synonym
	plant_detail	public synonym
Jill	sal_limit	private synonym
	work_city	public synonym
	plant_detail	table owner

Consider the following SQL statements & why they can or cannot be shared between the two users listed above.

SQL Statement	Object Matching	WHY
select max(sal_cap) from sal_limit;	NO	Each user has a private synonym **sal_limit** - these are different objects.
select count(*) from work_city where sdesc like 'NEW%';	YES	Both users reference **work_city** by the same public synonym - the same object.
select a.sdesc, b.location from work_city a, plant_detail b where a.city_id = b.city_id;	NO	User jack references **plant_detail** by a public synonym whereas user Jill is the table owner – these are different objects.
select * from sal_limit where over_time is not null;	NO	Each user has a private synonym **sal_limit** – these are different objects.

3. If bind variables are referenced, they must have the same name in both the new & existing statements.

For e.g.

The first two statements in the following listing are identical, whereas the next two statements are not (even if the different bind variables have the same value at run time).

```
select pin, name from people where pin = :blk1.pin;
select pin, name from people where pin = :blk1.pin;

select pos_id, sal_cap from sal_limit where over_time = :blk1.ot_ind;
select pos_id, sal_cap from sal_limit where over_time = :blk1.ov_ind;
```

4. Select the Most Efficient Table Name Sequence (Only for RBO)

ORACLE parser always processes table names from right to left, the table name you specify last (driving table) is actually the first table processed. If you specify more than one table in a FROM clause of a SELECT statement, you must choose the table containing the lowest number of rows as the driving table. When ORACLE processes multiple tables, it uses an internal sort/merge procedure to join those tables. First, it scans & sorts the first table (the one specified last in the FROM clause). Next, it scans the second table (the one prior to the last in the FROM clause) and merges all of the rows retrieved from the second table with those retrieved from the first table.

For e.g.

Table TAB1 has 16,384 rows.
Table TAB2 has 1 row.

Select TAB2 as the driving table.	(Best Approach)
SELECT COUNT(*) FROM TAB1, TAB2	0.96 seconds elapsed
Now, select TAB1 as the driving table.	(Poor Approach)
SELECT COUNT(*) FROM TAB2, TAB1	26.09 seconds elapsed

If three tables are being joined, select the intersection table as the driving table. The intersection table is the table that has many tables dependent on it.

For e.g.

The EMP table represents the intersection between the LOCATION table and the CATEGORY table.

```
SELECT . . .
FROM      LOCATION L,
          CATEGORY C,
          EMP E
WHERE     E.EMP_NO BETWEEN 1000 AND 2000
AND       E.CAT_NO = C.CAT_NO
AND         E.LOCN = L.LOCN
```

is more efficient than this next example:

```
SELECT . . .
FROM        EMP E,
            LOCATION L,
            CATEGORY C
WHERE       E.CAT_NO = C.CAT_NO
AND         E.LOCN = L.LOCN
AND         E.EMP_NO BETWEEN 1000 AND 2000
```

5. Position of Joins in the WHERE Clause

Table joins should be written first before any condition of WHERE clause. And the conditions which filter out the maximum records should be placed at the end after the joins as the parsing is done from **BOTTOM to TOP.**

For e.g.

Least Efficient : (Total CPU = 156.3 Sec)

```
SELECT  . . . .
FROM            EMP E
WHERE           SAL > 50000
AND             JOB = 'MANAGER'
AND             25 < (SELECT  COUNT(*)
                      FROM     EMP
                      WHERE    MGR = E.EMPNO);
```

Most Efficient : (Total CPU = 10.6 Sec)

```
SELECT  . . . .
FROM            EMP E
WHERE           25 < (SELECT  COUNT(*)
                      FROM     EMP
                      WHERE    MGR = E.EMPNO )
AND             SAL > 50000
AND             JOB = 'MANAGER';
```

6. Avoid Using * in SELECT Clause

The dynamic SQL column reference (*) gives you a way to refer to all of the columns of a table. Do not use * feature because it is a very inefficient one as the * has to be converted to each column in turn. The SQL parser handles all the field references by obtaining the names of valid columns from the data dictionary & substitutes them on the command line which is time consuming.

7. Reduce the Number of Trips to the Database

Every time a SQL statement is executed, ORACLE needs to perform many internal processing steps; the statement needs to be parsed, indexes evaluated, variables bound, and data blocks read. The more you can reduce the number of database accesses, the more overhead you can save.

For e.g.

There are 3 distinct ways of retrieving data about employees who have employee numbers 0342 or 0291.

Method 1 (Least Efficient) :

```
SELECT    EMP_NAME, SALARY, GRADE
FROM      EMP
WHERE     EMP_NO = 0342;

SELECT    EMP_NAME, SALARY, GRADE
FROM      EMP
WHERE     EMP_NO = 0291;
```

Method 2 (Next Most Efficient) :

```
DECLARE
    CURSOR C1(E_NO NUMBER) IS
    SELECT        EMP_NAME, SALARY, GRADE
    FROM          EMP
    WHERE         EMP_NO = E_NO;
BEGIN
    OPEN  C1(342);
    FETCH C1 INTO ..., ..., ...;
    .
    .
    OPEN  C1(291);
    FETCH C1 INTO ..., ..., ...;
    CLOSE C1;
END;
```

Method 3 (Most Efficient) :

```
SELECT    A.EMP_NAME, A.SALARY, A.GRADE,
          B.EMP_NAME, B.SALARY, B.GRADE,
FROM      EMP    A,
          EMP    B
WHERE     A.EMP_NO = 0342
AND       B.EMP_NO = 0291;
```

Note:

One simple way to increase the number of rows of data you can fetch with one database access & thus reduce the number of physical calls needed is to reset the ARRAYSIZE parameter in SQL*Plus, SQL*Forms & Pro*C. Suggested value is 200.

8. Use DECODE to Reduce Processing

The DECODE statement provides a way to avoid having to scan the same rows repetitively or to join the same table repetitively.

For e.g.

```
SELECT      COUNT(*), SUM(SAL)
FROM        EMP
WHERE       DEPT_NO = 0020
AND         ENAME LIKE 'SMITH%';

SELECT      COUNT(*), SUM(SAL)
FROM        EMP
WHERE       DEPT_NO = 0030
AND         ENAME LIKE 'SMITH%';
```

You can achieve the same result much more efficiently with DECODE:

```
SELECT COUNT(DECODE(DEPT_NO, 0020, 'X', NULL)) D0020_COUNT,
       COUNT(DECODE(DEPT_NO, 0030, 'X', NULL)) D0030_COUNT,
       SUM(DECODE(DEPT_NO, 0020, SAL, NULL)) D0020_SAL,
       SUM(DECODE(DEPT_NO, 0030, SAL, NULL)) D0030_SAL
FROM   EMP
WHERE  ENAME LIKE 'SMITH%';
```

Similarly, **DECODE** can be used in **GROUP BY** or **ORDER BY** clause effectively.

9. Combine Simple, Unrelated Database Accesses

If you are running a number of simple database queries, you can improve performance by combining them into a single query, even if they are not related.

For e.g.

```
SELECT      NAME
FROM        EMP
WHERE       EMP_NO = 1234;

SELECT      NAME
FROM        DPT
WHERE       DPT_NO = 10;

SELECT      NAME
FROM        CAT
WHERE       CAT_TYPE = 'RD';
```

The above three queries can be combined as shown below:

```
SELECT      E.NAME, D.NAME, C.NAME
FROM        CAT  C, DPT  D, EMP  E, DUAL X
WHERE       NVL('X', X.DUMMY) = NVL('X', E.ROWID (+))
AND         NVL('X', X.DUMMY) = NVL('X', D.ROWID (+))
AND         NVL('X', X.DUMMY) = NVL('X', C.ROWID (+))
AND             E.EMP_NO (+) = 1234
AND             D.DEPT_NO (+) = 10
AND             C.CAT_TYPE (+) = 'RD'
```

10. Deleting Duplicate Records

The efficient way to delete duplicate records from a table is shown below. It takes advantage of the fact that a row's ROWID must be unique.

```
DELETE FROM EMP E
WHERE E.ROWID > (SELECT MIN(X.ROWID)
                 FROM    EMP X
                 WHERE X.EMP_NO = E.EMP_NO);
```

11. Use TRUNCATE instead of DELETE

When rows are removed from a table, under normal circumstances, the rollback segments are used to hold undo information; if you do not commit your transaction, Oracle restores the data to the state it was in before your transaction started.

With **TRUNCATE**, no undo information is generated. Once the table is truncated, the data cannot be recovered back. It is **faster & needs fewer resources.**

Use TRUNCATE rather than DELETE for wiping the contents of small or large tables when you need no undo information generated.

12. Issue Frequent COMMIT statements

Whenever possible, issue frequent COMMIT statements in all your programs. By issuing frequent COMMIT statements, the **performance** of the program is **enhanced** & its resource requirements are minimized as **COMMIT frees** up the following **resources:**

Information held in the rollback segments to undo the transaction, if necessary.

All locks acquired during statement processing.

Space in the redo log buffer cache

Overhead associated with any internal Oracle mechanisms to manage the resources in the previous three items.

13. Counting Rows from Tables

Contrary to popular belief, COUNT(*) is faster than COUNT(1). If the rows are being returned via an index, counting the indexed column – for example, COUNT(EMPNO) is faster still.

14. Use WHERE in Place of HAVING

Avoid including a HAVING clause in SELECT statements. **The HAVING clause filters selected rows only after all rows have been fetched.** This could include sorting, summing, etc. Restricting rows via the WHERE clause, rather than the HAVING clause, helps reduce these overheads.

For e.g.

Least Efficient :

```
SELECT      REGION, AVG(LOC_SIZE)
FROM        LOCATION
GROUP BY    REGION
HAVING      REGION != 'SYDNEY'
AND         REGION != 'PERTH'
```

Most Efficient :

```
SELECT      REGION, AVG(LOC_SIZE)
FROM        LOCATION
GROUP BY    REGION
WHERE       REGION != 'SYDNEY'
AND         REGION != 'PERTH'
```

15. Minimize Table Lookups in a Query

To improve performance, minimize the number of table lookups in queries, particularly if your statements include sub-query SELECTs or multi-column UPDATEs.

For e.g.

Least Efficient :

```
SELECT      TAB_NAME
FROM        TABLES
WHERE       TAB_NAME = (SELECT      TAB_NAME
                         FROM        TAB_COLUMNS
                        WHERE        VERSION = 604)
AND         DB_VER = (SELECT  DB_VER
                      FROM    TAB_COLUMNS
                      WHERE   VERSION = 604)
```

Most Efficient :

```
SELECT      TAB_NAME
FROM        TABLES
WHERE       (TAB_NAME, DB_VER) = (SELECT   TAB_NAME, DB_VER
                                  FROM     TAB_COLUMNS
                                  WHERE    VERSION = 604)
```

Multi-column UPDATE e.g.

Least Efficient :

```
UPDATE      EMP
```

```
SET             EMP_CAT = (SELECT MAX(CATEGORY)
                                 FROM      EMP_CATEGORIES),
                SAL_RANGE = (SELECT MAX(SAL_RANGE)
                                    FROM      EMP_CATEGORIES )
WHERE           EMP_DEPT  = 0020;
```

Most Efficient :

```
UPDATE          EMP
SET             (EMP_CAT, SAL_RANGE) =
                (SELECT MAX(CATEGORY), MAX(SAL_RANGE)
                 FROM      EMP_CATEGORIES)
WHERE           EMP_DEPT = 0020;
```

16. Reduce SQL Overheads via "Inline" Stored Functions

```
SELECT          H.EMPNO, E.ENAME,
                H.HIST_TYPE, T.TYPE_DESC,
                COUNT(*)
FROM            HISTORY_TYPE T, EMP E, EMP_HISTORY H
WHERE           H.EMPNO = E.EMPNO
AND             H.HIST_TYPE = T.HIST_TYPE
GROUP BY        H.EMPNO, E.ENAME, H.HIST_TYPE, T.TYPE_DESC;
```

The above statement's performance may be improved via an inline function call as shown below:

```
FUNCTION Lookup_Hist_Type (typ IN number) return varchar2
AS
          tdesc varchar2(30);
          CURSOR C1 IS
          SELECT          TYPE_DESC
          FROM            HISTORY_TYPE
          WHERE           HIST_TYPE = typ;
BEGIN
          OPEN C1;
          FETCH C1 INTO tdesc;
          CLOSE C1;
          return (NVL(tdesc, '?'));
END;

FUNCTION Lookup_Emp (emp IN number) return varchar2
AS
          ename varchar2(30);
          CURSOR C1 IS
          SELECT          ENAME
          FROM            EMP
          WHERE           EMPNO = emp;
BEGIN
          OPEN C1;
          FETCH C1 INTO ename;
          CLOSE C1;
          return (NVL(ename, '?'));
```

```
END;

SELECT      H.EMPNO, Lookup_Emp(H.EMPNO),
            H.HIST_TYPE,   Lookup_Hist_Type(H.HIST_TYPE),
            COUNT(*)
FROM        EMP_HISTORY H
GROUP BY    H.EMPNO, H.HIST_TYPE;
```

17. Use Table Aliases

Always use table aliases & prefix all column names by their aliases where there is more than one table involved in a query. This will reduce parse time & prevent syntax errors from occurring when ambiguously named columns are added later on.

18. Use EXISTS in Place of IN for Base Tables

Many base table queries have to actually join with another table to satisfy a selection criteria. In such cases, the EXISTS (or NOT EXISTS) clause is often a better choice for performance.

For e.g.

Least Efficient :

```
SELECT      *
FROM        EMP           (Base Table)
WHERE       EMPNO > 0
AND         DEPTNO IN (SELECT DEPTNO
                       FROM   DEPT
                       WHERE LOC = 'MELB')
```

Most Efficient :

```
SELECT      *
FROM        EMP
WHERE       EMPNO > 0
AND         EXISTS (SELECT    'X'
                    FROM      DEPT
                    WHERE     DEPTNO = EMP.DEPTNO
                    AND       LOC = 'MELB')
```

19. Use NOT EXISTS in Place of NOT IN

In sub-query statements such as the following, the NOT IN clause causes an internal sort/merge. The NOT IN clause is the all-time slowest test, because it forces a full read of the table in the sub-query **SELECT.** Avoid using NOT IN clause either by replacing it with **Outer Joins or** with **NOT EXISTS** clause as shown below:

```
SELECT . . .
FROM        EMP
WHERE       DEPT_NO NOT IN (SELECT DEPT_NO
                                        FROM     DEPT
                                        WHERE  DEPT_CAT = 'A');
```

To improve the performance, replace this code with:

Method 1 (Efficient) :

```
SELECT . . .
FROM          EMP A, DEPT B
WHERE         A.DEPT_NO = B.DEPT_NO (+)
AND           B.DEPT_NO IS NULL
AND           B.DEPT_CAT(+) = 'A'
```

Method 2 (Most Efficient) :

```
SELECT . . .
FROM          EMP E
WHERE         NOT EXISTS (SELECT       'X'
                          FROM         DEPT
                          WHERE        DEPT_NO = E.DEPT_NO
                          AND          DEPT_CAT = 'A');
```

20. Use Joins in Place of EXISTS

In general join tables rather than specifying sub-queries for them such as the following:

```
SELECT        ENAME
FROM          EMP E
WHERE         EXISTS (SELECT      'X'
                      FROM        DEPT
                      WHERE       DEPT_NO = E.DEPT_NO
                      AND         DEPT_CAT = 'A');
```

To improve the performance, specify:

```
SELECT        ENAME
FROM          DEPT D, EMP E
WHERE         E.DEPT_NO = D.DEPT_NO
AND           D.DEPT_CAT = 'A';
```

21. Use EXISTS in Place of DISTINCT

Avoid joins that require the DISTINCT qualifier on the SELECT list when you submit queries used to determine information at the owner end of a one-to-many relationship (e.g. departments that have many employees).

For e.g.

Least Efficient :

```
SELECT        DISTINCT DEPT_NO, DEPT_NAME
FROM          DEPT D, EMP E
```

```
WHERE       D.DEPT_NO = E.DEPT_NO
```

Most Efficient :

```
SELECT      DEPT_NO, DEPT_NAME
FROM        DEPT D
WHERE       EXISTS (SELECT      'X'
                    FROM        EMP E
                    WHERE       E.DEPT_NO = D.DEPT_NO);
```

EXISTS is a faster alternative because the RDBMS kernel realizes that when the sub-query has been satisfied once, the query can be terminated.

22. Identify "Poorly Performing" SQL statements

Use the following queries to identify the poorly performing SQL statements.

```
SELECT    EXECUTIONS, DISK_READS, BUFFER_GETS,
          ROUND((BUFFER_GETS-DISK_READS)/BUFFER_GETS,2)    Hit_Ratio,
          ROUND(DISK_READS/EXECUTIONS,2)    Reads_Per_Run,
          SQL_TEXT
FROM      V$SQLAREA
WHERE     EXECUTIONS > 0
AND       BUFFER_GETS > 0
AND       (BUFFER_GETS - DISK_READS) / BUFFER_GETS < 0.80
ORDER BY 4 DESC;
```

23. Use TKPROF Utility to View Performance Statistics

The SQL trace facility writes a trace file containing performance statistics for the SQL statements being executed. The trace file provides valuable information such as the number of parses, executes and fetches performed, various types of CPU & elapsed times, the number of physical & logical reads, etc, that you can use to tune your system.

To enable SQL trace, use the following query:

```
ALTER SESSION SET SQL_TRACE TRUE
```

To globally enable SQL trace, you must set SQL_TRACE parameter to TRUE in init.ora. USER_DUMP_DEST parameter specifies the directory where SQL trace writes the trace file.

24. Use EXPLAIN PLAN To Analyze SQL Statements

Explain Plan is an Oracle function that analyzes SQL statements for performance without running the queries first. The results of the Explain Plan tell you the order that Oracle will search/join the tables, the types of access that will be employed (indexed search or full table scan), and the names of indexes that will be used.

You should read the list of operations from the inside out and from top to bottom. Thus, if two operations are listed, the one that is the most indented will usually be executed first. If the two operations are at the same level of indentation, then the one that is listed first (with the lowest operation number) will be executed first.

NESTED LOOPS joins are among the few execution paths that do not follow the "read from the inside out" rule of indented execution paths. To read the NESTED LOOPS execution path correctly, examine the order of the operations that directly provide data to the NESTED LOOPS operation. Of those operations, the operation with the lowest number is executed first.

25. Use Indexes to Improve Performance

An index is a conceptual part of a database table that may be used to speed up the retrieval of data from that table. Internally, ORACLE uses a sophisticated self-balancing **B-tree index structure.**

Indexed retrieval of data from a database is almost always faster than a full-table scan. The ORACLE optimizer uses the indexes defined for a table when it figures out the most efficient retrieval path for a query or update statement. ORACLE also uses indexes in performing more efficient joins of multiple tables. Another benefit of indexes is that they provide a way to guarantee the uniqueness of the primary key in a table.

You can index any column in a table except those defined with data types of **LONG or LONG RAW.** In general, indexes are most useful when they are specified on large tables. If small tables are frequently joined, however, you'll find that performance improves when you index these tables too.

Although indexes usually provide performance gains, there is a cost to using them. Indexes require storage space. They also require maintenance. Every time a record is added to or deleted from a table and every time an indexed column is modified, the index(es) itself must be updated as well. This can mean 4 or 5 extra disk I/Os per INSERT, DELETE or UPDATE for a record. Because indexes incur the overhead of data storage & processing, you can actually degrade response time if you specify indexes that you don't use.

The maximum number of indexes is usually between 4 & 6 per table. Do keep the number of indexes over a single table to a minimum, but if an index is useful and response times can be kept below the agreed-upon limit for your site, then don't hesitate to create the index.

26. Operations That Use Indexes

ORACLE performs two operations for accessing the indexes.

INDEX UNIQUE SCAN

In most cases, the optimizer uses index via the **where** clause f the query.

For e.g.

Consider a table LODGING having two indexes on it: a unique index LODGING_PK on the Lodging column & a non-unique index LODGING$MANAGER on the Manager column.

```
SELECT      *
FROM        LODGING
WHERE       LODGING = 'ROSE HILL';
```

Internally, the execution of the above query will be divided into two steps. First, the LODGING_PK index will be accessed via an **INDEX UNIQUE SCAN** operation. The RowID value that matches the 'Rose Hill' Lodging value will be returned from the index; that RowID value will then be used to query LODGING via a **TABLE ACCESS BY ROWID** operation.

If the value requested by the query had been contained within the index, then ORACLE would not have been needed to use the TABLE ACCESS BY ROWID operation; since the data would be in the index, the index would be all that was needed to satisfy the query. Because the query selected all columns from the LODGING table, and the index did not contain all of the columns of the LODGING table, the TABLE ACCESS BY ROWID operation was necessary.

The query shown below would require only **INDEX UNIQUE SCAN** operation.

```
SELECT      LODGING
FROM        LODGING
WHERE       LODGING = 'ROSE HILL';
```

INDEX RANGE SCAN

If you query the database based on a **range of values**, or if you query using a **non-unique index**, then an INDEX RANGE SCAN operation is used to query the index.

Example 1:

```
SELECT      LODGING
FROM        LODGING
WHERE       LODGING LIKE 'M%'
```

Since the **where** clause contains a range of values, the unique LODGING_PK index will be accessed via an **INDEX RANGE SCAN** operation. Because INDEX RANGE SCAN operations require reading multiple values from the index, they are **less efficient** than INDEX UNIQUE SCAN operations. Here, INDEX RANGE SCAN of LODGING_PK is the only operation required to resolve the query as only the LODGING column was selected by the query whose values are stored in the LODGING_PK index which is being scanned.

Example 2:

```
SELECT      LODGING
FROM        LODGING
WHERE       MANAGER = 'BILL GATES';
```

The above query will involve two operations: an **INDEX RANGE SCAN of LODGING$MANAGER** (to get the RowID values for all of the rows with 'BILL GATES' values in the MANAGER column), followed by a **TABLE ACCESS BY ROWID** of the LODGING table (to retrieve the LODGING column values). Since the LODGING$MANAGER index is a non-unique index, the database cannot perform an INDEX UNIQUE SCAN on LODGING$MANAGER, even if MANAGER is equated to a single value in the query.

Since the query selects the LODGING column & the LODGING column is not in the LODGING$MANAGER index, the **INDEX RANGE SCAN** must be followed by a **TABLE ACCESS BY ROWID** operation.

When specifying a range of values for a column, an **index will not be used** if the **first character** specified is a **wildcard.** The following query **will not use** the LODGING$MANAGER index:

```
SELECT      LODGING
FROM        LODGING
WHERE       MANAER LIKE '%HANMAN';
```

Here, a full table scan (TABLE ACCESS FULL operation) will be performed.

27. Selection of Driving Table

The **Driving Table** is the table that will be read first (usually via a TABLE ACCESS FULL operation). The method of selection for the driving table depends on the optimizer in use.

If you are using the CBO, then the optimizer will check the statistics for the size of the tables & the selectivity of the indexes & will choose the path with the lowest overall cost.

If you are using the RBO, and indexes are available for all join conditions, then the driving table will usually be the table that is listed **last** in the **FROM** clause.

For e.g.

```
SELECT      A.NAME, B.MANAGER
FROM        WORKER       A,
            LODGING      B
WHERE       A.LODGING = B.LODGING;
```

Since an index is available on the LODGING column of the LODGING table, and no comparable index is available on the WORKER table, the **WORKER table** will be used as the **driving table** for the query.

28. Two or More Equality Indexes

When a SQL statement has two or more equality indexes over different tables (e.g. WHERE = value) available to the execution plan, ORACLE uses both indexes by merging them at run time & fetching only rows that are common to both indexes.

The index having a UNIQUE clause in its CREATE INDEX statement ranks before the index that does not have a UNIQUE clause. However, this is true only when they are compared against constant predicates. If they are compared against other indexed columns from other tables, such clauses are much lower on the optimizer's list.

If the two equal indexes are over two **different tables,** table sequence determines which will be queried first; the table specified last in the FROM clause outranks those specified earlier.

If the two equal indexes are over the **same table,** the index referenced first in the WHERE clause ranks before the index referenced second.

For e.g.

There is a non-unique index over DEPTNO & a non-unique index over EMP_CAT:

```
SELECT      ENAME
FROM        EMP
WHERE       DEPTNO = 20
AND         EMP_CAT = 'A';
```

Here, the DEPTNO index is retrieved first, followed by (merged with) the EMP_CAT indexed rows. The Explain Plan is as shown below:

```
TABLE ACCESS BY ROWID ON EMP
        AND-EQUAL
                INDEX RANGE SCAN ON DEPT_IDX
                INDEX RANGE SCAN ON CAT_IDX
```

29. Equality & Range Predicates

When indexes combine both equality & range predicates over the same table, ORACLE cannot merge these indexes. It uses only the **equality predicate.**

For e.g.

There is a non-unique index over DEPTNO & a non-unique index over EMP_CAT:

```
SELECT      ENAME
FROM        EMP
WHERE       DEPTNO > 20
AND         EMP_CAT = 'A';
```

Here, only the EMP_CAT index is utilized & then each row is validated manually. The Explain Plan is as shown below:

```
TABLE ACCESS BY ROWID ON EMP
        INDEX RANGE SCAN ON CAT_IDX
```

30. No Clear Ranking Winner

When there is no clear index "ranking" winner, ORACLE will use only one of the indexes. In such cases, ORACLE uses the first index referenced by a WHERE clause in the statement.

For e.g.

There is a non-unique index over DEPTNO & a non-unique index over EMP_CAT:

```
SELECT          ENAME
FROM            EMP
WHERE           DEPTNO > 20
AND             EMP_CAT > 'A';
```

Here, only the DEPT_NO index is utilized & then each row is validated manually. The Explain Plan is as shown below:

```
TABLE ACCESS BY ROWID ON EMP
        INDEX RANGE SCAN ON DEPT_IDX
```

31. Explicitly Disabling an Index

If two or more indexes have equal ranking, you can force a particular index (that has the least number of rows satisfying the query) to be used. Concatenating || '' to character column or + 0 to numeric column suppresses the use of the index on that column.

For e.g.

```
SELECT          ENAME
FROM            EMP
WHERE           EMPNO                  = 7935
AND             DEPTNO + 0             =  10
AND             EMP_TYPE || ''         = 'A';
```

This is a rather dire approach to improving performance because disabling the WHERE clause means not only disabling current retrieval paths, but also disabling all future paths. You should resort to this strategy only if you need to tune a few particular SQL statements individually.

Here is an example of when this strategy is justified. Suppose you have a non-unique index over the EMP_TYPE column of the EMP table, and that the EMP_CLASS column is not indexed:

```
SELECT          ENAME
```

```
FROM        EMP
WHERE       EMP_TYPE  = 'A'
AND         EMP_CLASS = 'X';
```

The optimizer notices that EMP_TYPE is indexed & uses that path; it is the only choice at this point. If, at a later time, a second, non-unique index is added over EMP_CLASS, the optimizer will have to choose a selection path. Under normal circumstances, the optimizer would simply use both paths, performing a sort/merge on the resulting data. However, if one particular path is nearly unique (perhaps it returns only 4 or 5 rows) & the other path has thousands of duplicates, then the sort/merge operation is an unnecessary overhead. In this case, you will want to remove the EMP_CLASS index from optimizer consideration. You can do this by recording the SELECT statement as follows:

```
SELECT      ENAME
FROM        EMP
WHERE       EMP_TYPE              = 'A'
AND         EMP_CLASS || ''       = 'X';
```

32. Avoid Calculations on Indexed Columns

If the indexed column is a part of a function (in the WHERE clause), the optimizer does not use an index & will perform a full-table scan instead.

Note :

The SQL functions **MIN & MAX are exceptions** to this rule & will utilize all available indexes.

For e.g.

Least Efficient :

```
SELECT . . .
FROM        DEPT
WHERE       SAL * 12 > 25000;
```

Most Efficient :

```
SELECT . . .
FROM        DEPT
WHERE       SAL > 25000 / 12;
```

33. Automatically Suppressing Indexes

If a table has two (or more) available indexes, and that one index is unique & the other index is not unique, in such cases, ORACLE uses the unique retrieval path & completely ignores the second option.

For e.g.

```
SELECT      ENAME
FROM        EMP
WHERE       EMPNO = 2362
```

```
AND         DEPTNO = 20;
```

Here, there is a unique index over EMPNO & a non-unique index over DEPTNO The EMPNO index is used to fetch the row. The second predicate (DEPTNO = 20) is then evaluated (no index used). The Explain Plan is as shown below:

```
TABLE ACCESS BY ROWID ON EMP
        INDEX UNIQUE SCAN ON EMP_NO_IDX
```

34. Avoid NOT on Indexed Columns

In general, avoid using NOT when testing indexed columns. The NOT function has the same effect on indexed columns that functions do. When ORACLE encounters a NOT, it will choose not to use the index & will perform a full-table scan instead.

For e.g.

Least Efficient : (Here, index will not be used)

```
SELECT . . .
FROM        DEPT
WHERE       DEPT_CODE NOT = 0;
```

Most Efficient : (Here, index will be used)

```
SELECT . . .
FROM        DEPT
WHERE       DEPT_CODE > 0;
```

In a few cases, the ORACLE optimizer will automatically transform NOTs (when they are specified with other operators) to the corresponding functions:

```
NOT >       to      <=
NOT >=      to      <
NOT <       to      >=
NOT <=      to      >
```

35. Use >= instead of >

If there is an index on DEPTNO, then try

```
SELECT      *
FROM        EMP
WHERE       DEPTNO >= 4
```

Instead of

```
SELECT      *
FROM        EMP
WHERE       DEPTNO > 3
```

Because instead of looking in the index for the first row with column = 3 and then scanning forward for the first value that is > 3, the DBMS may jump directly to the first entry that is = 4.

36. Use UNION in Place of OR (in case of Indexed Columns)

In general, always use UNION instead of OR in WHERE clause. Using OR on an indexed column causes the optimizer to perform a full-table scan rather than an indexed retrieval. Note, however, that choosing UNION over OR will be effective only if both columns are indexed; if either column is not indexed, you may actually increase overheads by not choosing OR.

In the following example, both LOC_ID & REGION are indexed.

Specify the following:

```
SELECT          LOC_ID, LOC_DESC, REGION
FROM            LOCATION
WHERE           LOC_ID = 10
UNION
SELECT          LOC_ID, LOC_DESC, REGION
FROM            LOCATION
WHERE           REGION = 'MELBOURNE'
```

instead of

```
SELECT          LOC_ID, LOC_DESC, REGION
FROM            LOCATION
WHERE           LOC_ID = 10
OR              REGION = 'MELBOURNE'
```

If you do use OR, be sure that you put the most specific index first in the OR's predicate list, and put the index that passes the most records last in the list.

Note that the following:

```
WHERE           KEY1 = 10       Should return least rows
OR              KEY2 = 20       Should return most rows
```

is internally translated to:

```
WHERE           KEY1 = 10
AND             (KEY1 NOT = 10 AND KEY2 = 20)
```

37. Use IN in Place of OR

The following query can be replaced to improve the performance as shown below:

Least Efficient :

```
SELECT . . .
FROM            LOCATION
```

```
WHERE         LOC_ID = 10
OR            LOC_ID = 20
OR            LOC_ID = 30
```

Most Efficient :

```
SELECT . . .
FROM          LOCATION
WHERE         LOC_IN IN (10,20,30)
```

38. Avoid IS NULL & IS NOT NULL on Indexed Columns

Avoid using any column that contains a null as a part of an index. ORACLE can never use an index to locate rows via a predicate such as IS NULL or IS NOT NULL.

In a single-column index, if the column is null, there is no entry within the index. For concatenated index, if every part of the key is null, no index entry exists. If at least one column of a concatenated index is non-null, an index entry does exist.

For e.g.

If a UNIQUE index is created over a table for columns A & B and a key value of (123, null) already exists, the system will reject the next record with that key as a duplicate. However, if all of the indexed columns are null (e.g. null, null), the keys are not considered to be the same, because in this case ORACLE considers the whole key to be null & null can never equal null. You could end up with 1000 rows all with the same key, a value of null !

Because null values are not a part of an index domain, specifying null on an indexed column will cause that index to be omitted from the execution plan.

For e.g.

Least Efficient : (Here, index will not be used)

```
SELECT . . .
FROM          DEPARTMENT
WHERE         DEPT_CODE IS NOT NULL;
```

Most Efficient : (Here, index will be used)

```
SELECT . . .
FROM          DEPARTMENT
WHERE         DEPT_CODE >= 0;
```

39. Always Use Leading Column of a Multicolumn Index

If the index is created on multiple columns, then the index will only be used if the leading column of the index is used in a limiting condition (where clause) of the query. If your query specifies values for only the non-leading columns of the index, then the index will not be used to resolve the query.

40. Oracle Internal Operations

ORACLE performs internal operations when executing the query. The following table shows some of the important operations that ORACLE performs, while executing the query.

Oracle Clause	Oracle Internal Operations performed
ORDER BY	SORT ORDER BY
UNION	UNION-ALL
MINUS	MINUS
INTERSECT	INTERSECTION
DISTINCT, MINUS, INTERSECT, UNION	SORT UNIQUE
MIN, MAX, COUNT	SORT AGGREGATE
GROUP BY	SORT GROUP BY
ROWNUM	COUNT or COUNT STOPKEY
Queries involving Joins	SORT JOIN, MERGE JOIN, NESTED LOOPS
CONNECT BY	CONNECT BY

41. Use UNION-ALL in Place of UNION (Where Possible)

When the query performs a UNION of the results of two queries, the two result sets are merged via UNION-ALL operation & then the result set is processed by a SORT UNIQUE operation before the records are returned to the user.

If the query had used a UNION-ALL function in place of UNION, then the SORT UNIQUE operation would not have been necessary, thus improving the performance of the query.

For e.g.

Least Efficient :

```
SELECT      ACCT_NUM, BALANCE_AMT
FROM        DEBIT_TRANSACTIONS
WHERE       TRAN_DATE = '31-DEC-95'
UNION
SELECT      ACCT_NUM, BALANCE_AMT
FROM        CREDIT_TRANSACTIONS
WHERE       TRAN_DATE = '31-DEC-95'
```

Most Efficient :

```
SELECT      ACCT_NUM, BALANCE_AMT
FROM        DEBIT_TRANSACTIONS
WHERE       TRAN_DATE = '31-DEC-95'
UNION ALL
```

```
SELECT          ACCT_NUM, BALANCE_AMT
FROM            CREDIT_TRANSACTIONS
WHERE           TRAN_DATE = ‘31-DEC-95’
```

42. Using Hints

For table accesses, there are 2 relevant hints:

FULL & ROWID

The FULL hint tells ORACLE to perform a full table scan on the listed table.

For e.g.

```
SELECT          /*+ FULL(EMP) */ *
FROM  EMP
WHERE           EMPNO = 7839;
```

The ROWID hint tells the optimizer to use a TABLE ACCESS BY ROWID operation to access the rows in the table.

In general, you should use a TABLE ACCESS BY ROWID operation whenever you need to return rows quickly to users and whenever the tables are large. To use the TABLE ACCESS BY ROWID operation, you need to either know the ROWID values or use an index.

If a large table has not been marked as a cached table & you wish for its data to stay in the SGA after the query completes, you can use the CACHE hint to tell the optimizer to keep the data in the SGA for as long as possible. The CACHE hint is usually used in conjunction with the FULL hint.

For e.g.

```
SELECT          /*+ FULL(WORKER) CACHE(WORKER) */ *
FROM            WORKER;
```

The INDEX hint tells the optimizer to use an index-based scan on the specified table. You do not need to mention the index name when using the INDEX hint, although you can list specific indexes if you choose.

For e.g.

```
SELECT          /* + INDEX(LODGING) */ LODGING
FROM            LODGING
WHERE           MANAGER = 'BILL GATES';
```

The above query should use the index without the hint being needed. However, if the index is non-selective & you are using the CBO, then the optimizer may choose to ignore the index during the processing. In that case, you can use the INDEX hint to force an index-based data access path to be used.

There are several hints available in ORACLE such as ALL_ROWS, FIRST_ROWS, RULE, USE_NL, USE_MERGE, USE_HASH, etc for tuning the queries.

43. Use WHERE Instead of ORDER BY Clause

ORDER BY clauses use an index only if they meet 2 rigid requirements.

All of the columns that make up the ORDER BY clause must be contained within a single index in the **same sequence.**

All of the columns that make up the ORDER BY clause must be defined as NOT NULL within the table definition. Remember, null values are not contained within an index.

WHERE clause indexes & ORDER BY indexes cannot be used in parallel.

For e.g.

Consider a table DEPT with the following fields:

```
DEPT_CODE    PK       NOT NULL
DEPT_DESC             NOT NULL
DEPT_TYPE             NULL

NON UNIQUE INDEX (DEPT_TYPE)
```

Least Efficient : (Here, index will not be used)

```
SELECT       DEPT_CODE
FROM         DEPT
ORDER BY     DEPT_TYPE
```

Explain Plan:

```
SORT ORDER BY
        TABLE ACCESS FULL
```

Most Efficient : (Here, index will be used)

```
SELECT       DEPT_CODE
FROM         DEPT
WHERE        DEPT_TYPE > 0
```

Explain Plan:

```
TABLE ACCESS BY ROWID ON EMP
        INDEX RANGE SCAN ON DEPT_IDX
```

44. Avoid Converting Index Column Types

ORACLE automatically performs simple column type conversion or casting, when it compares two columns of different types.

Assume that EMPNO is an **indexed numeric column.**

```
SELECT . . .
FROM        EMP
WHERE       EMPNO = '123'
```

In fact, because of conversion, this statement will actually be processed as:

```
SELECT . . .
FROM        EMP
WHERE       EMPNO = TO_NUMBER('123')
```

Here, even though a type conversion has taken place, index usage is not affected.

Now assume that EMP_TYPE is an **indexed CHAR column.**

```
SELECT . . .
FROM        EMP
WHERE       EMP_TYPE = 123
```

This statement will actually be processed as:

```
SELECT . . .
FROM        EMP
WHERE       TO_NUMBER(EMP_TYPE) = 123
```

Indexes cannot be used, if they are included in a function. Therefore, this internal conversion will keep the index from being used.

45. Beware of the WHEREs

Some SELECT statement WHERE clauses do not use indexes at all. Here, are some of the examples shown below:

In the following example, **the != function cannot use an index.** Remember, indexes can tell you what is in a table, but not what is not in a table. All references to **NOT, != and <> disable index** usage:

Do Not Use:

```
SELECT      ACCOUNT_NAME
FROM        TRANSACTION
WHERE       AMOUNT != 0;
```

Use:

```
SELECT      ACCOUNT_NAME
FROM        TRANSACTION
WHERE       AMOUNT > 0;
```

In the following example, || is the concatenate function. It, like other functions, disables indexes.

Do Not Use:

```
SELECT        ACCOUNT_NAME, AMOUNT
FROM          TRANSACTION
WHERE         ACCOUNT_NAME || ACCOUNT_TYPE = 'AMEXA';
```

Use:

```
SELECT        ACCOUNT_NAME, AMOUNT
FROM          TRANSACTION
WHERE         ACCOUNT_NAME = 'AMEX'
AND           ACCOUNT_TYPE = 'A';
```

In the following example, addition (+) is a function and disables the index. The other arithmetic operators (-, *, and /) have the same effect.

Do Not Use:

```
SELECT        ACCOUNT_NAME, AMOUNT
FROM          TRANSACTION
WHERE         AMOUNT+ 3000 < 5000;
```

Use:

```
SELECT        ACCOUNT_NAME, AMOUNT
FROM          TRANSACTION
WHERE         AMOUNT < 2000;
```

In the following example, indexes cannot be used to compare indexed columns against the same index column. This causes a full-table scan.

Do Not Use:

```
SELECT        ACCOUNT_NAME, AMOUNT
FROM          TRANSACTION
WHERE         ACCOUNT_NAME = NVL(:ACC_NAME, ACCOUNT_NAME);
```

Use:

```
SELECT        ACCOUNT_NAME, AMOUNT
FROM          TRANSACTION
WHERE         ACCOUNT_NAME LIKE NVL(:ACC_NAME, '%');
```

46. CONCATENATION of Multiple Scans

If you specify a list of values for a column's limiting condition, then the optimizer may perform multiple scans & concatenate the results of the scans.

For e.g.

```
SELECT        *
FROM          LODGING
```

```
WHERE        MANAGER IN ('BILL GATES', 'KEN MULLER');
```

The optimizer may interpret the query as if you had provided two separate limiting conditions, with an **OR** clause as shown below:

```
SELECT       *
FROM         LODGING
WHERE        MANAGER = 'BILL GATES'
OR           MANAGER = 'KEN MULLER';
```

When resolving the above query, the optimizer may perform an INDEX RANGE SCAN on LODGING$MANAGER for each of the limiting conditions. The RowIDs returned from the index scans are used to access the rows in the LODGING table (via TABLE ACCESS BY ROWID operations). The rows returned from each of the TABLE ACCESS BY ROWID operations are combined into a single set of rows via the CONCATENATION operation.

The Explain Plan is as shown below:

```
SELECT STATEMENT Optimizer=CHOOSE
  CONCATENATION
    TABLE ACCESS (BY INDEX ROWID) OF LODGING
      INDEX (RANGE SCAN) OF LODGING$MANAGER (NON-UNIQUE)
    TABLE ACCESS (BY INDEX ROWID) OF LODGING
      INDEX (RANGE SCAN) OF LODGING$MANAGER (NON-UNIQUE)
```

47. Use the Selective Index (Only For CBO)

The Cost-Based Optimizer can use the selectivity of the index to judge whether using the index will lower the cost of executing the query.

If the index is highly selective, then a small number of records are associated with each distinct column value.

For example, if there are 100 records in a table & 80 distinct values for a column in that table, then the selectivity of an index on that column is 80/100 = 0.80 The higher the selectivity, the fewer the number of rows returned for each distinct value in the column.

If an index has a low selectivity, then the many INDEX RANGE SCAN operations & TABLE ACCESS BY ROWID operations used to retrieve the data may involve more work than a TABLE ACCESS FULL of the table.

48. Avoid Resource Intensive Operations

Queries which uses DISTINCT, UNION, MINUS, INTERESECT, ORDER BY and GROUP BY call upon SQL engine to perform resource intensive sorts. A DISTINCT requires one sort, the other set operators requires at least two sorts.

For example, a **UNION** of queries in which each query contains a **group by** clause will require nested sorts; a sorting operation would be required for each of the queries, followed by the SORT UNIQUE operation required for the **UNION.** The sort operation required for the **UNION** will not be able to begin until the sorts for the **group by** clauses have completed. The more deeply nested the sorts are, the greater the performance impact on your queries.

Other ways of writing these queries should be found. Most queries that use the set operators, UNION, MINUS and INTERSECT, can be rewritten in other ways.

49. GROUP BY & Predicate Clauses

The performance of GROUP BY queries can be improved by eliminating unwanted rows early in the selection process. The following two queries return the same data, however, the second is potentially quicker, since rows will be eliminated before the set operators are applied.

For e.g.

Least Efficient :

```
SELECT      JOB, AVG(SAL)
FROM        EMP
GROUP BY    JOB
HAVING      JOB = 'PREDIDENT'
OR          JOB = 'MANAGER'
```

Most Efficient :

```
SELECT      JOB, AVG(SAL)
FROM        EMP
WHERE       JOB = 'PREDIDENT'
OR          JOB = 'MANAGER'
GROUP BY    JOB
```

50. Using Dates

When using dates, note that, if more than 5 decimal places are added to a date, the date is actually rounded up to the next day !

For e.g.

```
SELECT      TO_DATE('01-JAN-93') + .99999
FROM        DUAL;
```

returns:

```
'01-JAN-93 23:59:59'
```

And,

```
SELECT      TO_DATE('01-JAN-93') + .999999
FROM        DUAL;
```

returns:

‘02-JAN-93 00:00:00’

51. Use Explicit Cursors

When implicit cursors are used, two calls are made to the database, once to fetch the record and then to check for the TOO MANY ROWS exception. Explicit cursors prevent the second call.

52. Tuning EXPort & IMPort

Run Export & Import with a large buffer size, say 10 MB (10,240,000) **to speed up the process.** Oracle will acquire as much as you specify and will not return an error if it can 't find that amount. Set this value to at least as large as the largest table column value, otherwise the field will be truncated.

53. Table and Index Splitting

Always create separate tablespaces for your tables & indexes and never put objects that are not part of the core Oracle system in the system tablespace. Also ensure that data tablespaces & index tablespaces reside on separate disk drives.

The reason is to allow the disk head on one disk to read the index information while the disk head on the other disk reads the table data. Both reads happen faster because one disk head is on the index and the other is on the table data. If the objects were on the same disk, the disk head would need to reposition itself from the index extent to the data extent between the index read and the data read. This can dramatically decrease the throughput of data in a system.

54. CPU Tuning

Allocate as much real memory as possible to the shared pool & database buffers (SHARED_POOL_SIZE & DB_BLOCK_BUFFERS in init.ora) to permit as much work as possible to be done in memory. Work done in memory rather than disk does not use as much CPU.

Set the SEQUENCE_CACHE_ENTRIES in init.ora high. (Default is 10 - try setting it to 1000).

Allocate more than the default amount of memory to do sorting (SORT_AREA_SIZE); memory sorts not requiring I/O use much less CPU.

On multi-CPU machines, increase the LOG_SIMULTANEOUS_COPIES to allow one process per CPU to copy entries into the redo log buffers.

55. Use UTLBstat & UTLEstat to Analyze Database Performance

Oracle supplies two scripts UTLBstat.sql & UTLEstat.sql to gather a snapshot of ORACLE performance over a given period of time.

UTLBstat gathers the initial performance statistics. It should not be run immediately after the database has started or it will skew your results as none of the system caches are loaded initially.

UTLEstat gathers performance statistics at the end of your observations period. This script must be run at the end of the period for which you want to tune performance. It then generates a report of the complete information.

In order that all the statistics to be populated during a UTLBstat/UTLEstat session, you must set TIMED_STATISTICS=TRUE in init.ora

You must run UTLBstat from sqldba, because it does a connect internal to start the collection of statistics.

```
Sqldba> @utlbstat.sql
```

The output from UTLBstat/UTLEstat is placed in report.txt

56. Interpreting the Output (report.txt) from UTLBstat/UTLEstat

Library Cache

This cache contains parsed & executable SQL statements. An important key to tuning the SGA is ensuring that the library cache is large enough so Oracle can keep parsed & executable statements in the shared pool.

RELOAD represents entries in the library cache that were parsed more than once. You should strive for zero RELOADs. The solution is to increase SHARED_POOL_SIZE parameter. Alternatively, you can calculate this ratio by using the following query.

```
SELECT      SUM(pins), SUM(reloads),
            SUM(reloads) / (SUM(pins)+SUM(reloads)) * 100
FROM        V$LIBRARYCACHE
```

If the ratio is above 1%, increase the SHARED_POOL_SIZE in init.ora

GETHITRATIO & PINHITRATIO should always be greater than 80%. If you fall below this mark, you should increase the value of SHARED_POOL_SIZE.

Hit Ratio

Determine the Hit Ratio using the following formulae:

```
Logical Reads = Consistent Gets + DB Block Gets
Hit Ratio = (Logical Reads - Physical Reads) / Logical Reads
```

Hit Ratio should be greater than 80%. If the Hit Ratio is less than 80%, increase the value of DB_BLOCK_BUFFERS (data cache). The larger the data cache, the more likely the Oracle database will have what it needs in memory. The smaller the cache, the more likely Oracle will have to issue I/Os to put the information in the cache.

Buffer Busy Wait Ratio

The goal is to eliminate all waits for resources. Determine the ratio using the following formulae.

Logical Reads = Consistent Gets + DB Block Gets
Buffer Busy Wait Ratio = Buffer Busy Waits / Logical Reads

A ratio of greater than 4% is a problem.

Sorts

The sorts (disk) row tells you how many times you had to sort to disk; that is a sort that could not be handled by the size you specified for SORT_AREA_SIZE parameter.

The sorts (memory) row tells you how many times you were able to complete the sort using just memory. Usually, 90% or higher of all sorting should be done in memory.

To eliminate sorts to disk, increase SORT_AREA_SIZE parameter. The larger you make SORT_AREA_SIZE, the larger the sort that can be accomplished by Oracle in memory. Unlike other parameters, this is allocated per user. This is taken from available memory, not from the Oracle SGA area of memory.

Chained Blocks

Eliminate Chained Block. If you suspect chaining in your database & it is small enough, export and import the entire database. This will repack the database, eliminating any chained blocks.

Dictionary Cache

Dictionary Cache contains data dictionary information pertaining to segments in the database (e.g. indexes, sequences, and tables) file space availability (for acquisition of space by object creation & extension) and object privileges.

A well-tuned database should report an average dictionary cache hit ratio of over 90% by using following query:

```
SELECT        (1- (sum(getmisses) /
              (SUM(gets)+SUM(getmisses))))*100 "Hit Ratio"
FROM          V$ROWCACHE
```

Database Buffer Cache

A Cache Hit means the information required is already in memory.
A Cache Miss means Oracle must perform disk I/O to satisfy a request.

The secret when sizing the database buffer cache is to keep the cache misses to a minimum.