DBFastCollect Review

Nov 2007

NOTICE OF PROPRIETARY INFORMATION

Table of Contents

1. TUning Tips

1. Oracle SQL Coding Tips

Tips for SQL Coding in the context of Oracle are in the below file. These tips can improve performance and reduce resource consumption.

1. 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;**
2. 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
```

3. 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';
```

4. 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.

5. 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.

6. 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.

7. 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;
```

8. 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;
```

9. 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')
```

10. 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');
```

11. 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.

12. 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;
```

13. 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 of 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.

14. 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.

15. 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
```

16. 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
```

17. 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
```

18. 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';
```

19. 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;
```

20. 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
```

21. 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 >
```

22. 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.

23. 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)
```
24. 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)
```
25. 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;
```

26. 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.

27. 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. UNION clause removes the duplicates and UNION ALL keeps duplicates records so it skips the step of Sorting and removing duplicates of UNION clause, so **IF WE ARE SURE** that the tables which we are using in UNION does not produce duplicate records then we should always go for UNION ALL instead of UNION.

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'
```

28. 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.

29. 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
```

30. 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, '%');
```

31. 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.

32. 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.

33. 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
```

35. 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.

36. 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.

37. 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.

38. 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

39. 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.