INTRODUCTION TO

PERFORMANCE TUNING

1.0 Introduction

This paper discusses provides basics of performance tuning. This paper should be read in conjunction with the power point presentation associated with this paper.

This paper is NOT designed as a step-by-step approach, rather as a guideline. Every effort has been taken to reproduce the test results. It is possible for the test results to differ slightly due to version/platform differences.

2.0 Tracing SQL execution

First step to resolve any performance issue is to identify the issue. SQL performance tuning is no different and first we need to identify the execution step consuming elapsed time.

Sessions are typically in one of two states:

1. Either they are executing on CPU
2. Waiting for a resource to be available in the database.

There is a third state in which the session can wait for CPU resource in the host Operating System. We will ignore that state for now. You need to understand if the time is spent in the CPU or waiting for an event. You need to trace the SQL executions to breakdown the response time.

SQL statements can be traced with waits using few methods. Most common methods are

```
alter session set events '10046 trace name context forever, level 8';
exec dbms_session.session_trace_enable(waits => true);
```

Trace event level 10046 has various levels and the level is a 4-bit integer:

Trace level is a 4-bit integer:

- Level 1 : Lowest level, same as sql_trace=true
- level 4 : Level 1 + captures bind variables
- level 8 : Level 1 + captures wait events at SQL level
- level 12: level 1 + captures bind and wait events at SQL level

2.1 Example

Following few lines shows few lines from the trace file after executing a SQL statement. SQL statement is printed below. After parsing the statement various trace lines are printed indicating the wait event and amount of time spent in each wait event in micro-seconds (from 10g onwards). There are more information printed in the trace file too, such as the specific file_id, block_id read etc.

We will discuss ways to summarize the trace file in a more human readable format later in this paper.

Listing 1-1: Trace file example

```
PARSING IN CURSOR #3 len=83 dep=0 uid=173 oct=3 lid=173 tim=12972441295985
hv=855947554 ad='fa2e5530'
select transaction_id from mtl_transaction_accounts where transaction_id=1682944981
END OF STMT
```

```
PARSE #3:c=10000,e=5080,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=12972441295971
EXEC #3:c=0,e=198,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=12972441296397
WAIT #3: nam='SQL*Net message to client' ela= 8 driver id=1650815232 #bytes=1 p3=0
obj#=38149 tim=12972441296513
WAIT #3: nam='db file sequential read' ela= 8337 file#=803 block#=213006 blocks=1
obj#=38172 tim=12972441305307
WAIT #3: nam='db file sequential read' ela= 12840 file#=234 block#=65037 blocks=1
obj#=38172 tim=12972441318487
WAIT #3: nam='db file sequential read' ela= 4095 file#=803 block#=213474 blocks=1
obj#=38172 tim=12972441324278
```

2.2 Tracing with bind variables

SQL Trace also provides facility to print bind variables. Bind variables can be useful to reproduce a performance issue in a production environment. For programs designed with excessive loop based logic, tracing with bind variable can generate huge trace files and increase performance issues.

Tracing SQL statements with Binds and waits:

```
alter session set events '10046 trace name context forever, level 12';
exec dbms_session.session_trace_enable(waits => true, binds => true);
```

3.0 Analyzing trace files

Raw trace files are very hard to read. As humans, we would like to know which SQL statement is causing the problem as a summary. Oracle Corporation supplies a utility tkprof to summarize the trace files in to a more human readable format.
In Listing 2-1, a trace file devl_ora_123.trc file is summarized using tkprof utility. In a typical program, many SQL statements can be executed. You need to identify which SQL statement is causing the performance issue. Sort option is essential to achieve that; Sort option sorts the SQL statement based upon various options.
In most cases, you are more interested to see what are the top SQL statements by elapsed time in your program. Sort option exeela and fchela will summarize the trace files and sort the summary by Execution elapsed time and Fetch Elapsed time. So, Top SQL statements by elapsed time are printed in the beginning of the tkprof output file so that you can focus on those costly SQL statements.

Listing 2-1: tkprof example

```
tkprof  trace_file  outfile  sort=option explain=user/pwd  sys=no

tkprof  devl_ora_123.trc /tmp/devl_ora_tkp.out sort=exeela,fchela
```

3.1 Explain option

tkprof utility also provides an explain option. If you specify explain option in tkprof, that session will login to the local database, execute "explain plan for <SQL>.." and print in the trace file. It is probably not a good idea. SQL execution itself will print the execution plan. That plan is more accurate than the output provided with explain plan option.
If the execution plan is not printed properly, then it is possible that trace file is not closed properly. If you logoff from the database after executing the program, then trace file will be closed and complete execution plan will be printed.

4.0 Understanding Execution plan

Understanding execution plan is another important step in resolving performance issues. You specifically need to understand where the execution starts and how it proceeds step by step.

Listing 4-1 shows an example how you can easily query the execution plan. It is always better to execute the statement with a proper bind variable, but in some cases, it is easier to just review the execution plan using the method described in Listing 4-1. Execution sequence is also enumerated in the Listing 4-1 thoroughly.

Listing 4-1: Understanding execution plan

```
explain plan for
select t1.id, t1.vc10 , t2.id
From  t1,t2 where t1.id=t2.id and t2.id <1000;
```

```
Select * from table(dbms_xplan.display);

-------------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |   975 | 22425 |   512   (1)| 00:00:02 |
|*  1 | HASH JOIN                    |       |   975 | 22425 |   512   (1)| 00:00:02 |
|*  2 |  INDEX RANGE SCAN            | T2_N1 |   976 |  5856 |     5   (0)| 00:00:01 |
|   3 |  TABLE ACCESS BY INDEX ROWID | T1    |  1000 | 17000 |   506   (1)| 00:00:02 |
|*  4 |   INDEX RANGE SCAN           | T1_N1 |  1000 |       |     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
 1 - access("T1"."ID"="T2"."ID")
 2 - access("T2"."ID"<1000)
 4 - access("T1"."ID"<1000)
```

Execution sequence

(i) Index T2_N1 was scanned with access predicate t2.id <1000 [Step 2]

(ii) Index T1_N1 was scanned with access predicate t1.id <1000 [Step 4]

(iii) Table T1 is accessed to retrieve non-indexed column using rowids returned from step 2. [Step 3].

(iv) Rows from step 2 and 3 are joined to create the final result set.[Step 1]

4.1 Predicates

Predicates are very useful to understand the execution plan. Generally speaking, Access predicates shows how the indexes were accessed, Filter predicates shows how the rows are filtered after accessing a table or in a join operation.

Cardinality estimates are good guess to understand the performance issue. As a developer or DBA, you can quickly point out that cardinality estimate is incorrect in the execution plan. Incorrect cardinality estimates are most common cause for optimizer choosing inefficient execution plan. Remember that it is only an estimate, not actual number of rows.

In the Listing 4-2, cardinality estimates

4.2 Use of predicates: An example

Listing 4-2: Predicates and cardinality at step 5 is 20. But, after applying filter predicates at table level, cardinality estimates went down to 1. Essentially, optimizer expects to get 20 ROWIDs in step 5, retrieve 20 rows from the table, and then after applying filter predicates, estimate is down to 1. Filter predicates are reducing the work and reviewing filter predicates at step 4 we see that predicates on column request_id and rma_number are applied in step 4. Is it possible to apply the filter predicates earlier, before accessing the table? That will reduce number of rows retrieved from the table.

Listing 4-2: Use of predicates

```
Set lines 120 pages 0

Select * from table(dbms_xplan.display);

-------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name                         | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                              |     1 |   214 |     9  (23)|
|   1 |  SORT ORDER BY                 |                              |     1 |   214 |     9  (23)|
|*  2 |   TABLE ACCESS BY INDEX ROWID  | RCV_STAGING_LINES            |     1 |   155 |     4  (25)|
|   3 |    NESTED LOOPS                |                              |     1 |   214 |     8  (13)|
|*  4 |     TABLE ACCESS BY INDEX ROWID| RCV_STAGING_HEADERS          |     1 |    59 |     5  (20)|
|*  5 |      INDEX RANGE SCAN          | RCV_STAGING_HEADERS_N5       |    20 |       |     4  (25)|
|*  6 |     INDEX RANGE SCAN           | RCV_STAGING_LINES_N3         |     1 |       |     3  (34)|
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("RSL"."RMA_NUMBER"=:Z AND "RSL"."RMA_LINE_NUMBER"=TO_NUMBER(:Z))

   4 - filter("RSH"."RMA_NUMBER"=:Z AND "RSH"."REQUEST_ID"=TO_NUMBER(:Z))
   5 - access("RSH"."STATUS"='PROCESSED')

   6 - access("RSH"."SOURCE_HEADER_ID"="RSL"."SOURCE_HEADER_ID")
```

One way to filter early is to add those filtered columns to the index. Listing 4-3 shows the execution plan after adding those filtered columns to the index. Note that my intention is not to advocate adding more indexes, rather it is to show how filter and access predicates can be effectively used. Also note that indexes need to be maintained during DML operations and too many indexes can hamper DML performance. After applying the predicates at the index step (5), cardinality estimates went down to 1.

Listing 4-3: Use of predicates and indexes

```
create index apps.rcv_staging_headers_n5 on  apps.RCV_STAGING_HEADERS
  (status, rma_number, request_id ) compress 1 compute statistics;

Select * from table(dbms_xplan.display);

-------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name                         | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                              |     1 |   206 |     9  (23)|
|   1 |  SORT ORDER BY                 |                              |     1 |   206 |     9  (23)|
|*  2 |   TABLE ACCESS BY INDEX ROWID  | RCV_STAGING_LINES            |     1 |   155 |     4  (25)|
|   3 |    NESTED LOOPS                |                              |     1 |   206 |     8  (13)|
|   4 |     TABLE ACCESS BY INDEX ROWID| RCV_STAGING_HEADERS          |     1 |    51 |     5  (20)|
|*  5 |      INDEX RANGE SCAN          | RCV_STAGING_HEADERS_N5       |     1 |       |     4  (25)|
|*  6 |     INDEX RANGE SCAN           | RCV_STAGING_LINES_N3         |     1 |       |     3  (34)|
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("RSL"."RMA_NUMBER"=:Z AND "RSL"."RMA_LINE_NUMBER"=TO_NUMBER(:Z))
   5 - access("RSH"."STATUS"='PROCESSED' AND "RSH"."RMA_NUMBER"=:Z AND
              "RSH"."REQUEST_ID"=TO_NUMBER(:Z))
   6 - access("RSH"."SOURCE_HEADER_ID"="RSL"."SOURCE_HEADER_ID")
```

5.0 DBMS_XPLAN

In section 4.0, I gently introduced dbms_xplan package and it is a versatile package with many options. We will discuss some of the options here, please review documentation for all facilities available in this package, all options are not thoroughly discussed here.
Listing 5-1 provides short summary of options available in dbms_xplan package.
Listing 5-1: Short summary of dbms_xplan package

To Print execution plan for recently explained SQL statement:
dbms_xplan.display;
To Print execution plan for a SQL statement executed by passing sql_id or hash_value:
dbms_xplan.display_cursor ('&sqlid', '', '');
To Print execution plan for a SQL statement captured by AWR report
dbms_xplan.display_awr ('&sqlid', '', '');

5.1 Autotrace deprecated

From 10g onwards, autotrace is deprecated and you should plan on using dbms_xplan package calls instead of using autotrace. Listing 5-2 provides a short example of dbms_xplan package call replacing autotrace. Executing dbms_xplan.display_Cursor without any parameter prints the execution plan of the recently *executed* statement.

Listing 5-2: Autotrace deprecated

```
select t1.id, t1.vc10 , t2.id
From  t1,t2 where t1.id=t2.id and t2.id <1000;


Select * from table (dbms_xplan.display_cursor);

SQL_ID  cd6h52abfgfg3, child number 0
-------------------------------------
select t1.id, t1.vc10 , t2.id From  t1,t2 where t1.id=t2.id and t2.id <1000

Plan hash value: 3286489634
--------------------------------------------------------------------------
| Id  | Operation          | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |       |       |   512 (100)|          |
|*  1 |  HASH JOIN         |       |   975 | 22425 |   512   (1)| 00:00:02 |
...
```

Following are few key salient points about dbms_xplan.display_cursor pertaining to the listing 5-2.

1. dbms_xplan accesses vsql, vsql_plan_statistics_all and v$sql_plan internally. So, you need to have select access on these fixed views to execute dbms_xplan.
2. Executing dbms_xplan.display_cursor without specifying any sql_id retrieves the execution plan for the query executed last.
3. From 10g onwards, serveroutput is on by default. You need to disable serveroutput to see the execution plan correctly:

   Select * from table (dbms_xplan.display_cursor);

   SQL_ID 9babjv8yq8ru3, child number 1
   BEGIN DBMS_OUTPUT.GET_LINES(:LINES, :NUMLINES); END;

5.2 display_cursor

Package dbms_xplan provides facility to print the execution plan of a SQL in the shared pool. In Listing 5-3, sql_id is passed to the dbms_xplan.display_cursor to print the execution plan.

Listing 5-3: Display_cursor

```
select * from table(dbms_xplan.display_cursor ( '&sql_id','',''));

SQL_ID  fqmkvmdyb043r, child number 0
-------------------------------------
select t1.id, t1.vc10 , t2.id From  t1,t2 where t1.id=t2.id and t2.id <1000

Plan hash value: 3286489634
---------------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |       |       |   512 (100)|          |
|*  1 |  HASH JOIN                   |       |   975 | 22425 |   512   (1)| 00:00:02 |
|*  2 |   INDEX RANGE SCAN           | T2_N1 |   976 |  5856 |     5   (0)| 00:00:01 |
|   3 |   TABLE ACCESS BY INDEX ROWID| T1    |  1000 | 17000 |   506   (1)| 00:00:02 |
...
```

6.0 Statistics_level

In all the examples we saw so far, we are missing a key component. Tkprof utility provides top SQL statements to tune, but it hasn't provided step level statistics so far. A parameter statistics_level=all must be set in your session or system level for the step level statistics to be collected.
In the Listing 6-1, an execution plan is printed with number of rows returned in each step. It is not possible to identify the step that consumed more time. You need to know where the time is spent so that step can be tuned properly.

Listing 6-1: Where is the time spent?

```
 Rows     Row Source Operation
 -------  ---------------------------------------------------
   49996  WINDOW SORT
  644538   NESTED LOOPS
  704891    NESTED LOOPS
  704891     NESTED LOOPS
  704891      NESTED LOOPS OUTER
  704891       NESTED LOOPS
  704891        NESTED LOOPS
       1         TABLE ACCESS BY INDEX ROWID HR_ALL_ORGANIZATION_UNITS
       1          INDEX UNIQUE SCAN HR_ORGANIZATION_UNITS_PK
  704891         TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_DETAILS
  704891          INDEX RANGE SCAN WSH_DELIVERY_DETAILS_N8
  704891        TABLE ACCESS BY INDEX ROWID MTL_SYSTEM_ITEMS_B
  704891         INDEX RANGE SCAN MTL_SYSTEM_ITEMS_B_U1
  644783       VIEW PUSHED PREDICATE
  644783        NESTED LOOPS
  704891         TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUE_SETS
  704891          INDEX UNIQUE SCAN FND_FLEX_VALUE_SETS_U2
  644783         TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUES
  644783          INDEX RANGE SCAN FND_FLEX_VALUES_N1
  704891      TABLE ACCESS BY INDEX ROWID OE_ORDER_LINES_ALL
  704891       INDEX UNIQUE SCAN OE_ORDER_LINES_U1
  704891     TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_ASSIGNMENTS
  704891      INDEX RANGE SCAN WSH_DELIVERY_ASSIGNMENTS_N3
  644538    TABLE ACCESS BY INDEX ROWID WSH_NEW_DELIVERIES
  704891     INDEX UNIQUE SCAN WSH_NEW_DELIVERIES_U1
```

After turning on the parameter statistics_level at session level, tkprofed the trace file, you can see that step level execution statistics are printed. Now, you can see that how time is being spent in each step. You would focus on the step that consumed more time to improve the performance of this SQL statement.

Listing 6-2: Statistics_level=all prints valuable step level information.

```
 Rows     Row Source Operation
 -------  ---------------------------------------------------
   49996  WINDOW SORT (cr=11333940 r=66442 w=46044 time=327835520 us)
  644538   NESTED LOOPS  (cr=11333940 r=41631 w=0 time=302305095 us)
  704891    NESTED LOOPS  (cr=9219265 r=34793 w=0 time=263965257 us)
  704891     NESTED LOOPS  (cr=7093885 r=27898 w=0 time=217008667 us)
  704891      NESTED LOOPS OUTER (cr=4979210 r=460 w=0 time=80832751 us)
  704891       NESTED LOOPS  (cr=2219750 r=460 w=0 time=46780270 us)
  704891        NESTED LOOPS  (cr=51405 r=0 w=0 time=15862881 us)
       1         TABLE ACCESS BY INDEX ROWID HR_ALL_ORGANIZATION_UNITS (cr=3 r=0 w=0 time=79 us)
       1          INDEX UNIQUE SCAN HR_ORGANIZATION_UNITS_PK (cr=2 r=0 w=0 time=42 us)(object id 43498)
  704891         TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_DETAILS (cr=51402 r=0 w=0 time=15456581 us)
  704891          INDEX RANGE SCAN WSH_DELIVERY_DETAILS_N8 (cr=2692 r=0 w=0 time=1677304 us)(object id 5124003)
  704891        TABLE ACCESS BY INDEX ROWID MTL_SYSTEM_ITEMS_B (cr=2168345 r=460 w=0 time=26259264 us)
  704891         INDEX RANGE SCAN MTL_SYSTEM_ITEMS_B_U1 (cr=1409795 r=213 w=0 time=15069327 us)(object id 38017)
  644783       VIEW PUSHED PREDICATE  (cr=2759460 r=0 w=0 time=30859890 us)
  644783        NESTED LOOPS  (cr=2759460 r=0 w=0 time=29181767 us)
  704891         TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUE_SETS (cr=1409784 r=0 w=0 time=11031089 us)
  704891          INDEX UNIQUE SCAN FND_FLEX_VALUE_SETS_U2 (cr=704893 r=0 w=0 time=6257393 us)(object id 33768)
  644783         TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUES (cr=1349676 r=0 w=0 time=13839752 us)
  644783          INDEX RANGE SCAN CCW_FND_FLEX_VALUES_N1 (cr=704893 r=0 w=0 time=8683861 us)(object id 5153800)
  704891      TABLE ACCESS BY INDEX ROWID OE_ORDER_LINES_ALL (cr=2114675 r=27438 w=0 time=133292520 us)
  704891       INDEX UNIQUE SCAN OE_ORDER_LINES_U1 (cr=1409784 r=2025 w=0 time=14863664 us)(object id 42102)
  704891     TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_ASSIGNMENTS (cr=2125380 r=6895 w=0 time=42893004 us)
  704891      INDEX RANGE SCAN WSH_DELIVERY_ASSIGNMENTS_N3 (cr=1413108 r=2580 w=0 time=24181814 us)(object id
46295)
  644538    TABLE ACCESS BY INDEX ROWID WSH_NEW_DELIVERIES (cr=2114675 r=6838 w=0 time=35307346 us)
  704891     INDEX UNIQUE SCAN WSH_NEW_DELIVERIES_U1 (cr=1409784 r=362 w=0 time=7381948 us)(object id 46306)
```

7.0 SQL Monitor

There is an inherent problem in the method discussed so far. SQL statement must complete at least once so the execution step level statistics are printed properly. In a production environment, in many cases, you don't have an option as a statement can run for days before completion and the business would want to have the program complete in the next hour.

In 11gR2, a new feature SQL Monitor has been introduced. This feature collects the execution step details while the execution is proceeding. You don't need to wait for the execution to complete and the step level execution metrics are available in real time. Further, this feature is completely integrated with dbms_sqltune package and Enterprise Manager screens.

Listing 7-1 shows an example how SQL stats for a running query can be retrieved using dbms_sqltune package. Enterprise Manager also screens showing this more elegantly.

Listing 7-1: SQL Monitor

```
set lines 180
col rpt format a180
set  long 1000000
select dbms_sqltune.report_sql_monitor('&sqlid') rpt from dual;
SQL Monitoring Report

SQL Text
------------------------------
select count(a.n1), count(distinct(a.v1)) from rs.hugetable a , rs.hugetable b ,
rs.hugetable c

Error: ORA-1013
------------------------------
ORA-01013: user requested cancel of current operation

Global Information
------------------------------
 Status              :  DONE (ERROR)
 Instance ID         :  1
 Session             :  SYS (48:21)
 SQL ID              :  8jwwx25pq37j1
```

```
 SQL Execution ID       :  16777217
 Execution Started      :  03/17/2011 20:56:05
 First Refresh Time     :  03/17/2011 20:56:09
 Last Refresh Time      :  03/17/2011 20:56:59
 Duration               :  54s
 Module/Action          :  sqlplus@solrac1 (TNS V1-V3)/-
 Service                :  SYS$USERS
 Program                :  sqlplus@solrac1 (TNS V1-V3)
 Fetch Calls            :  1

Global Stats
=========================================================================================
| Elapsed |   Cpu   |    IO    | Application |  Other   | Fetch | Buffer | Read | Read  |
| Time(s) | Time(s) | Waits(s) |  Waits(s)   | Waits(s) | Calls |  Gets  | Reqs | Bytes |
=========================================================================================
|      54 |      33 |     4.51 |        0.00 |       16 |     1 |  26913 | 1718 | 210MB |
=========================================================================================

SQL Plan Monitoring Details (Plan Hash Value=1443272992)
=========================================================================================
========================================================================
| Id |          Operation           |   Name     |  Rows    | Cost |    Time      | Start   |
Execs |    Rows     | Read  | Read    |  Mem   | Activity |    Activity Detail      |
|    |                              |            | (Estim)  |      | Active(s)  | Active |
| (Actual) | Reqs  | Bytes | (Max)  |    (%)     |     (# samples)        |
=========================================================================================
========================================================================
|  0 | SELECT STATEMENT             |            |          |      |            |        |
1 |          |       |       |       |          |                       |
|  1 |   SORT AGGREGATE             |            |        1 |      |            |        |
1 |          |       |       |       |          |                       |
|  2 |    VIEW                      | VW_DAG_0   |    40293 |   6T |            |        |
1 |          |       |       |       |          |
...
```

8.0 Access methods and Joins

There are various access methods and joins available in Oracle database. In this section we will explore few access methods.

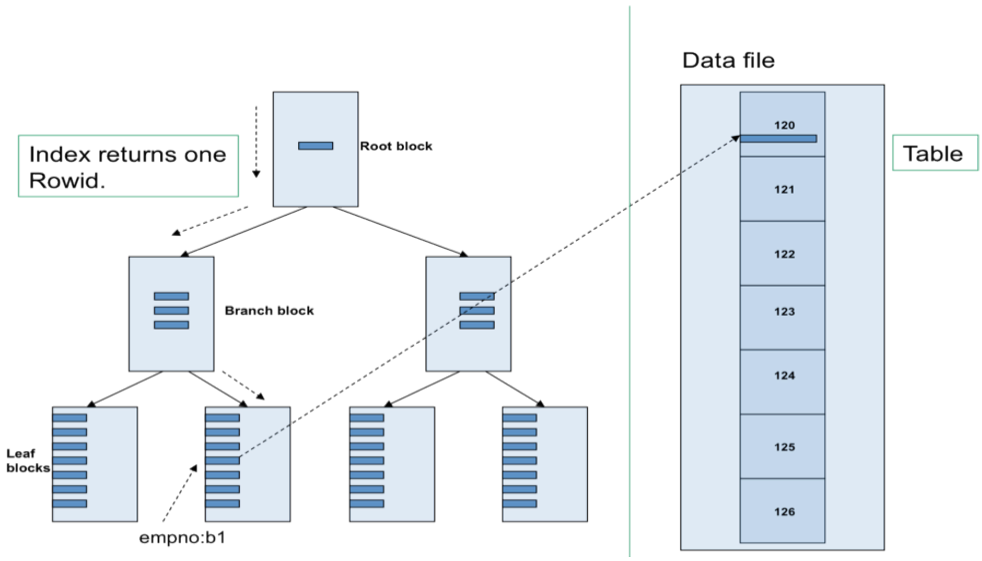
8.1 Index unique scan

In this access method, index is accessed using an unique column, a ROWID retrieved from the unique index using the column value lookup. Using the rowed, table block is accessed to retrieve the row (optionally).

Listing 8-1: Index Unique Scan

```
explain plan for select ename from demo1.emp where empno=:b1
------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |     1 |    10 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |     1 |    10 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_PK |     1 |       |     0   (0)| 00:00:01 |
------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPNO"=TO_NUMBER(:B1))
```



8.2 Index Range scan

Index is searched with non-unique column values and Multiple ROWIDs may be retrieved from the index. For every ROWID returned, table block is accessed to retrieve other selected columns. In the example of listing 8-2, index EMP_PK is accessed using a between operator, which can return man rows. Table block is accessed using those ROWIDs. Even if there is just one row returned, non-unique indexes will use index range scan.

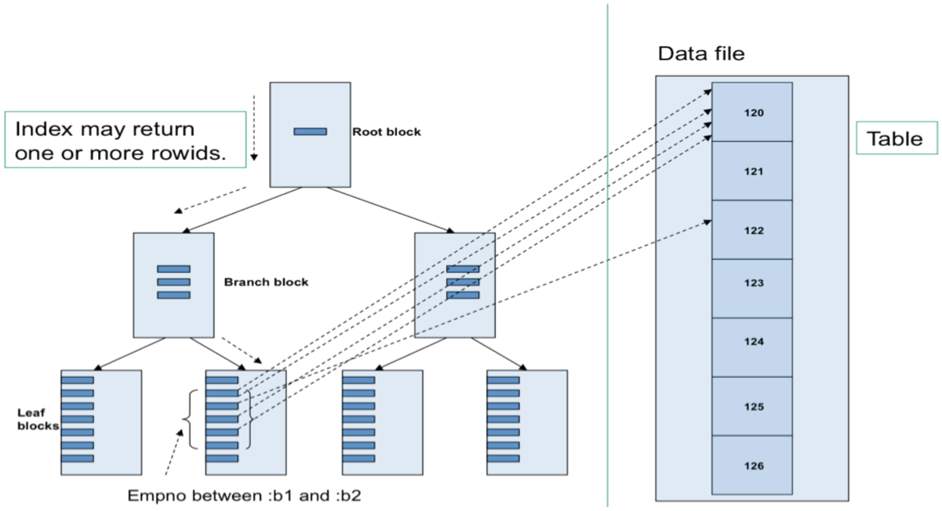
Listing 8-2: Index Range Scan

```
explain plan for select ename from demo1.emp where empno between :b1 and :b2;

 --------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     1 |    10 |     2   (0)| 00:00:01 |
|*  1 |  FILTER                      |         |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP     |     1 |    10 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | EMP_PK  |     2 |       |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(TO_NUMBER(:B1)<=TO_NUMBER(:B2))
   3 - access("EMPNO">=TO_NUMBER(:B1) AND "EMPNO"<=TO_NUMBER(:B2))
```

Search for the column key values starts at root block, traverses to branch block, traverses to leaf block. Index leaf block entries (ROWIDs) matching with the column values are returned. Using these ROWIDs, table block is accessed to retrieve the complete row piece.



8.3 Index Skip scan

Index is searched with non-leading columns of the index. Generally, performs better if the leading column values have few distinct values.

In the Listing 8-3, leading column of the index EMP_C2 is not specified, only ename is specified. Since there are very few distinct values for the leading columns in this table data, INDEX SKIP SCAN is chosen.

Listing 8-3: Index Skip Scan

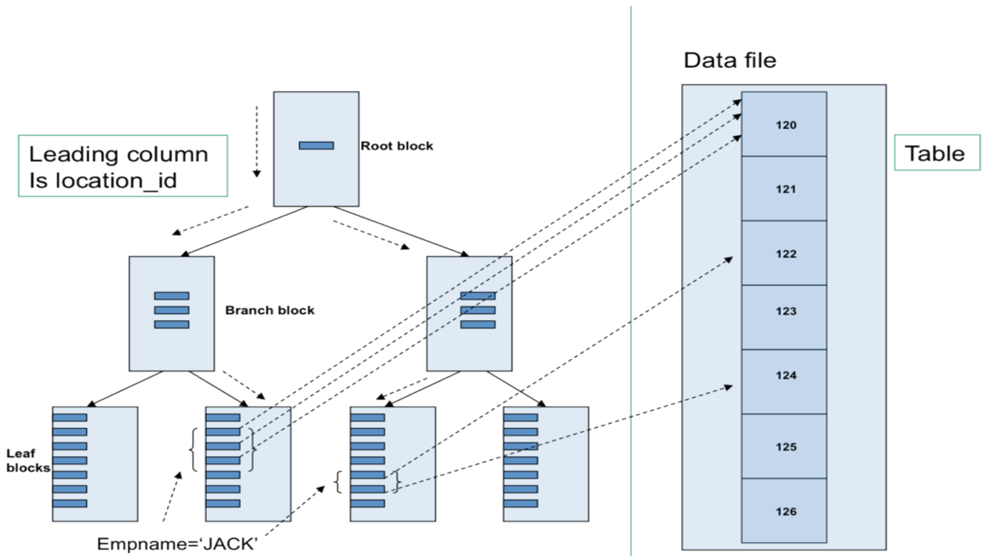
```
Index           Column                         Type
------          -----------                    -------------
EMP_C2          LOCATION_ID                  1 NUMBER(22)
                ENAME                        2 VARCHAR2(10)

select  *  from emp a  where ename='JACK';

--------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     1 |    43 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP     |     1 |    43 |     3   (0)| 00:00:01 |
|*  2 |   INDEX SKIP SCAN            | EMP_C2  |     1 |       |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("ENAME"='JACK')
       filter("ENAME"='JACK')
```

For each distinct value of the leading column, index is searched using the non-leading column. In this example, search will start with location_id=10, search for ename='JACK'. After last JACK entry in the index leaf block, values will be searched with location_id=20 for ename='JACK' etc.

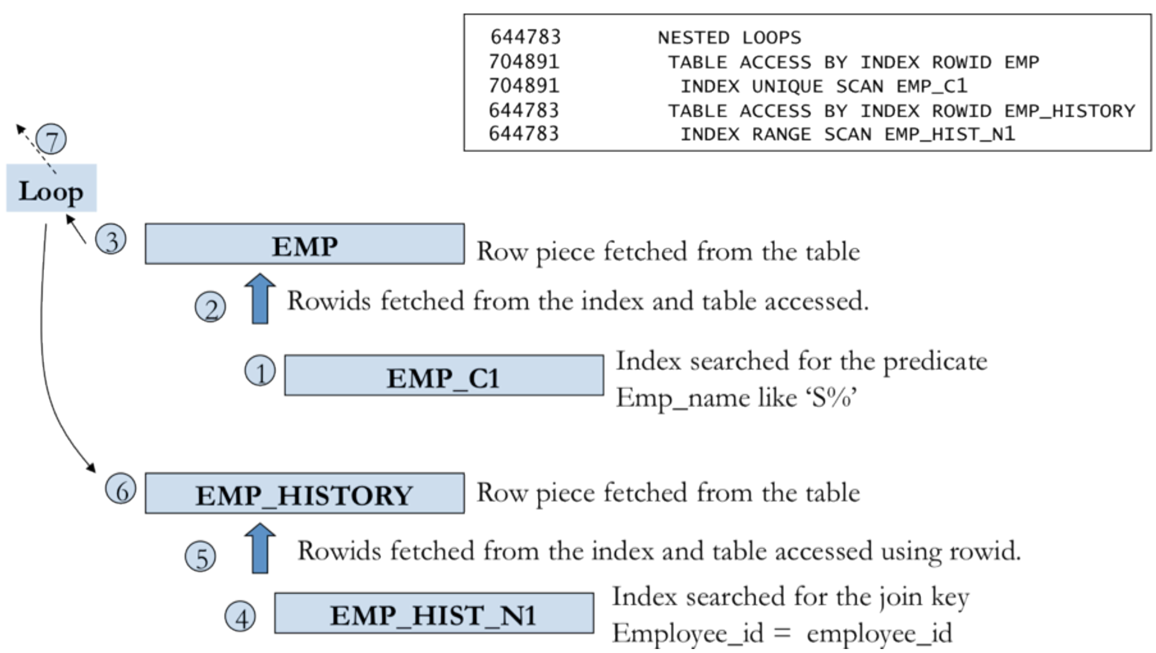


8.4 Nested Loops Join

For every row from outer row source, inner row source is probed for a matching row satisfying join key condition. This join technique, generally, performs better if number of rows from the inner row source is much lower.

Listing 8-4: Nested Loops Join

```
644783          NESTED LOOPS
704891           TABLE ACCESS BY INDEX ROWID EMP
704891            INDEX UNIQUE SCAN EMP_C1
644783           TABLE ACCESS BY INDEX ROWID EMP_HISTORY
644783            INDEX RANGE SCAN EMP_HIST_N1
```



8.5 Hash Join

Rows from two row sources fetched and then joined. In the Listing 8-5, rows from table T1 is retrieved and a hash lookup (in memory) created. Then the rows from Table T2 are retrieved and joined using an hashing algorithm to determine if there is a match. Matched rows are returned for further processing. Also note that Bloom filtering techniques can be used to pre-filter the rows.

Hash join is quite complex and not all details are discussed in this paper.

Listing 8-5: Hash join

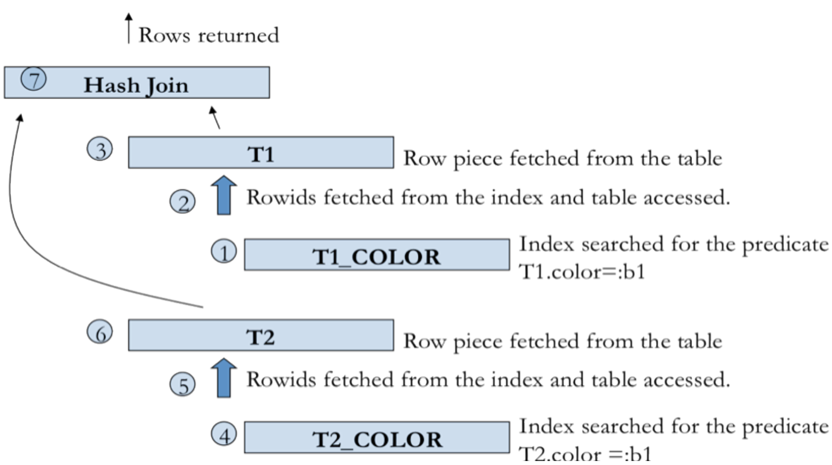
```
explain plan for select t1.color, t2.color, t1.shape   from
t1, t2 where t1.color_id = t2.color_id and t1.color=:b1 and t2.color=:b1
/
Explained.

SQL> select * from table (dbms_xplan.display);

Plan hash value: 2339531555

-----------------------------------------------------------------d-----------
------------
| Id  | Operation                    | Name     | Rows  | Bytes | Cost (%CPU)|
Time     |
----------------------------------------------------------------------------
-----------
|   0 | SELECT STATEMENT             |          |   333 |  9990 |     8  (13)|
00:00:01 |
|*  1 |  HASH JOIN                   |          |   333 |  9990 |     8  (13)|
00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1       |   333 |  6660 |     3   (0)|
00:00:01 |
|*  3 |    INDEX RANGE SCAN          | T1_COLOR |   333 |       |     1   (0)|
00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| T2       |   500 |  5000 |     4   (0)|
00:00:01 |
|*  5 |    INDEX RANGE SCAN          | T2_COLOR |   500 |       |     2   (0)|
00:00:01 |
----------------------------------------------------------------------------
-----------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("T1"."COLOR_ID"="T2"."COLOR_ID")
   3 - access("T1"."COLOR"=:B1)
   5 - access("T2"."COLOR"=:B1)

19 rows selected.
```



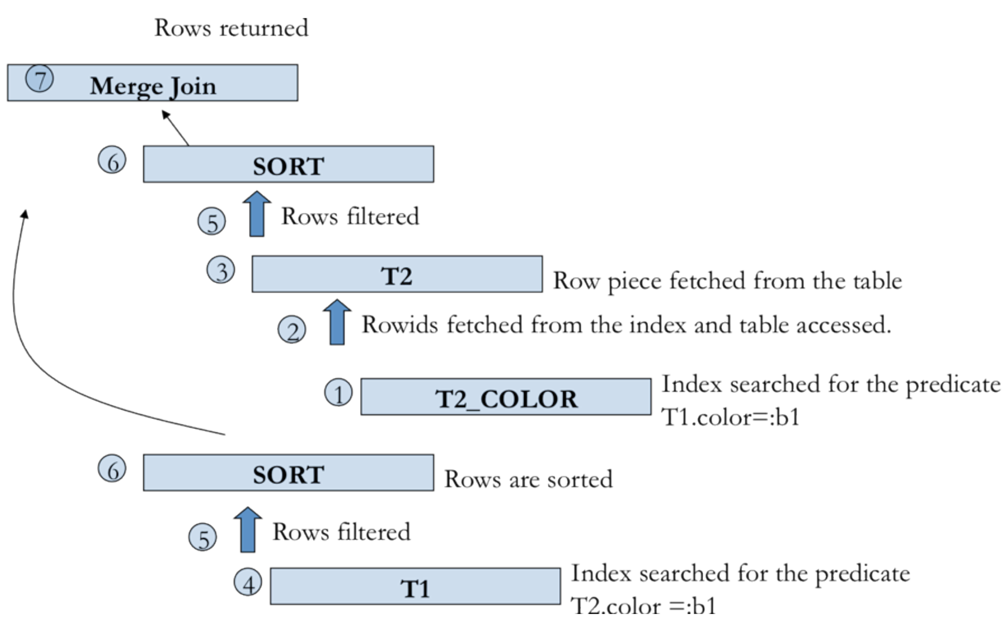
8.6 Merge Join

Rows from two row sources are sorted and joined using merge join technique. There is an additional overhead of sorting involved in the merge join. Generally, hash join will outperform instead of merge join. Cartesian Merge join is a special case in which every row from one row source is joined to every row in another row source.

```
select /*+ use_merge (t1, t2) */  t1.color, t2.color, t1.shape   from
t1, t2 where t1.color_id = t2.color_id and t1.color='black' and
t2.color='black'
-----------------------------------------------------------------------------
------------
| Id  | Operation                     | Name     | Rows  | Bytes | Cost
(%CPU)| Time     |
-----------------------------------------------------------------------------
------------
|   0 | SELECT STATEMENT              |          |       |       |    10
(100)|          |
|   1 |  MERGE JOIN                   |          |   500 | 15000 |    10
(20)| 00:00:01 |
|   2 |   SORT JOIN                   |          |   500 |  5000 |     5
(20)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| T2       |   500 |  5000 |     4
(0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | T2_COLOR |   500 |       |     2
(0)| 00:00:01 |
|*  5 |   SORT JOIN                   |          |   900 | 18000 |     5
(20)| 00:00:01 |
|*  6 |    TABLE ACCESS FULL          | T1       |   900 | 18000 |     4
(0)| 00:00:01 |
-----------------------------------------------------------------------------
------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("T2"."COLOR"='black')
   5 - access("T1"."COLOR_ID"="T2"."COLOR_ID")
       filter("T1"."COLOR_ID"="T2"."COLOR_ID")
   6 - filter("T1"."COLOR"='black')
```



9.0 Common problems in execution plan

Few common problems resulting in inefficient execution plans:

1. Optimizer choosing incorrect cardinality. Use cardinality feedback method to correct cardinality issues.
2. Incorrect choice of leading table in a join.
3. Optimizer estimating very small cardinality due to uniform and disjoint selectivity.
4. View merging did not happen.

10.0 Indexing, Partitioning and more

Indexes and partitioning scheme can be used to improve the performance of SQL queries. Following is few key points to consider while designing indexes:

1. Choose columns frequently used in the predicates of critical queries to be the leading columns of the index.
2. Indexes on join key columns are useful to improve performance of critical queries.
3. Selectivity of the columns is important. For b-tree indexes, columns should be very selective, meaning access through the index should return minimal amount of rows. Primary key or Unique key columns are highly selective; Columns such as Sex column with very few distinct values are not good candidate for b-tree indexes. Although, bitmap indexes can be considered.
4. Do not index columns that are updated heavily.
5. There is more than just one index type. Use Index Organized Table, Compressed Indexes, Bitmap indexes etc matching with the workload of application table.

Conclusion

We introduced basic concepts about performance tuning. There is wealth of information in Documentation and performance tuning guide is a must for any body to know more about performance tuning.

About the author

Riyaj Shamsudeen has 18+ years of experience in Oracle and he specializes in RAC and Performance. He is also an EBusiness DBA. He is a proud member of OakTable Network and Oracle ACE. He has co-authored few books on Oracle database performance and SQL. He blogs regularly at http://orainternals.wordpress.com

References

1. Oracle support site. Metalink.oracle.com. Various documents

2. Internal's guru Steve Adam's website
   www.ixora.com.au

3. Jonathan Lewis' website
   www.jlcomp.daemon.co.uk

4. Julian Dyke's website
   www.julian-dyke.com

5. 'Oracle8i Internal Services for Waits, Latches, Locks, and Memory'
   by Steve Adams

6. Tom Kyte's website
   Asktom.oracle.com

7. My blog
   Orainternals.wordpress.com