SQL Query Performance Tuning Case Study

Tao Zuo, NPD Group

Introduction

SQL tuning is a phenomenally complex subject. There are many books and papers on this topic. From release to release, Oracle built more and more intelligence into its query optimizer to run a query in an optimal manner. That also says, to understand oracle’s internal on how a query being run becomes more and more difficult. There are many tuning software and utilities in the market to try to make SQL tuning “a click away”. However you still find times with feeling of helpless, and SQL tuning is still an art work.

Through this paper, I would like to share two cases encountered in real life practice that you may find the fundamental utilities are real gifts to crack the hard shell.

There are many factors affect a SQL performance: Hardware, Network, OS, DB, Application, etc. Bear in mind that system level tuning could benefit many queries. So don’t forget your tuning path is from top to bottom: from system to query itself. Ensure the objects statistics is up to date is essential, as some query maybe very sensitive to the small changes on object statistics and most of the time refreshing the statistics solve the query performance problem immediately. If statistics staleness is not the issue, further SQL tuning analysis shall begin.
Case Study 1

Here is an example. We received a client’s complaint that their report is running in a hour glass for almost 45 minutes. We exam database system load, and all other applications responses, all appeared normal, the issue was narrowed down to a specific query. Which query contributes to the hour glass then? In our practice, we found gv$sql_monitor is pretty handy:

```sql
SELECT inst_id,
       status,
       sid,
       session_serial#,
       username,
       module,
       action,
       service_name,
       client_identifier,
       client_info,
       program,
       cpu_time,
       queuing_time,
       elapsed_time, -- Elapsed time (in microseconds); updated as the statement executes
       sql_id,
       sql_text
FROM gv$sql_monitor
WHERE elapsed_time > 1000000 * 60 * 60
ORDER BY elapsed_time DESC;
```

By the elapsed_time, username, client_info, client_identifier, program, sql_text, etc. we may quickly locate the problematic query. It can also provide us some useful information on the INST_ID (instance ID for RAC database) that the query runs, its SID, etc. for later further analysis.

With the SID, and INST_ID gathered in the previous query, we may find SPID (OS process ID) from the following v$process and v$session:

```sql
select s.username, s.sid, s.osuser, s.program, s.machine, p.spid, s.status
from v$session s, v$process p
where s.paddr = p.addr
and s.sid in (922)
;
--------
30765
```

Note: with INST_ID, we can directly execute the above query in that instance.
With the SPID, we may take a look at how that process behaves on the server, using TOP utility:

```
top - 20:16:49 up 14 days, 11:55, 3 users, load average: 1.92, 1.14, 0.92
Tasks: 730 total, 3 running, 727 sleeping, 0 stopped, 0 zombie
Cpu(s): 13.4%us, 0.2%sy, 0.0%ni, 86.4%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 1920846656 total, 1512162756 used, 388692k free, 269432k buffers
Swap: 296712868k total, 6900232k used, 188675548k free, 118970888k cached

  PID USER      PR  NI  VIRT  RES   SHR S %CPU %MEM     TIME+  COMMAND
30765 oracle   20   0  88.4g 1.5g  1.4g R 100.0  1.2   0:50.22 oracle
30268 oracle   20   0  69.4g  2.3g  1.3g R  99.9  1.6   1:19.81 oracle
23132 oracle   16   0  88.5g  5.0m  50m  S  4.3  0.4  0:106.22 oracle
3112 Oracle    15   0  88.5g  1.4g  1.4g S  2.0  0.1  0:119.43 oracle
```

In this case, the process 30765 is sitting on the first line of the TOP output. It is taking 100% of a single CPU. However overall server load is pretty low: load average is between 0.92 and 1.92, or there is almost no process needs to wait in the CPU run queue. In another word, the server does give sufficient resource to this process, and it still can’t return with results in an expected time. From here we learned that this process is a CPU bound process.

This is how it looks in Oracle Enterprise Manager Session Detail page. We learned that the process was burning a whole CPU from 9:04PM - 9:59PM, over 50 minutes, with no return. From our experience, it is very possible that the query is running in a suboptimal plan with nested loops used in the wrong place killing the CPU.
SQL Tuning Advisor has an interface embedded in Oracle Enterprise Manager. If you have Tuning Packs turned on, you may use it to tune the problematic SQL in your database.

To find out if Tuning Pack is turned on, use this:

```
SQL> show parameter control_management_pack_access
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>control_management_pack_access</td>
<td>string</td>
<td>DIAGNOSTIC+TUNING</td>
</tr>
</tbody>
</table>

Since we licensed with Diagnostic+Tuning Packs, and it is just a click away, it would be awesome if this click can get the problem solved.

After running the SQL advisor, we got the recommendation as the following:

Well, it found a better execution plan, which may improve the query performance by about 32%.

It also provided a nice graph on how the system resource usage would be reduced after applying its recommendation:
So we implement the recommendation, and run the query again. Here is how it looks from OEM:

![Query Execution Plan]

Query ran from 11:15AM till 12:08PM, about 53 minutes, no returns. Similar to its original execution plan, major time was pinning on CPU, no performance improvement. No silver bullet!

Here comes the SQL tuning fundamentals. 

**Format the query**

Wait, does query format have anything to do with SQL performance? Well, No. Oracle can parse a query in any format, and execute it. However making the code readable is importance to appeal its structure, so we understand the logic and to make the tuning work easier. Either SQL developer or TOAD provides the feature of “Format the code” for the developers, so formatting the code is really just a click away.

![Oracle SQL Developer]

The query that we are dealing with has 92 lines, many layers of subqueries, and a few views in addition to plenty tables.
Before formatting, the query looks cumbersome, and the logic is hard to catch.

After we format the code, the query becomes 110 lines, however, the query structure is much more obvious. We can clearly see each subquery layer, and not being confused by those repeat used subquery block aliases.
Here the 3 alias of “e” and 2 “a” represent different query blocks. This needs to be corrected, not only for the code readability, but also the query tuning. Since we have learned this query has 4 layers, therefore each layer block is labeled with the layer# as its alias’s suffix:
Explain Plan

Now, with the query formatted, we use the following to generate the query explain plan:

explain plan set statement_id='f' for
<QUERY>
;

And We use the following to show the explain plan output:

select * from table(dbms_xplan.display);

Here is the query explain plan:

Plan hash value: 3464502560

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>TempSpc</th>
<th>Time</th>
<th>Cost (%)</th>
<th>Pstart</th>
<th>Pstop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>94916</td>
<td>480M</td>
<td></td>
<td>00:00:01</td>
<td>287K</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY GLOBAL INDEX ROWID</td>
<td>XYZ_CODES</td>
<td>1</td>
<td>25</td>
<td>3</td>
<td>00:00:01</td>
<td>ROWID</td>
<td>ROWID</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS BY GLOBAL INDEX ROWID</td>
<td>XYZ_CODES</td>
<td>1</td>
<td>25</td>
<td>3</td>
<td>00:00:01</td>
<td>ROWID</td>
<td>ROWID</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>XYZ_RET999ERS</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>XYZ_RET999ERS</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>TABLE ACCESS BY GLOBAL INDEX ROWID</td>
<td>XYZ_CODES</td>
<td>1</td>
<td>25</td>
<td>3</td>
<td>00:00:01</td>
<td>ROWID</td>
<td>ROWID</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>XYZ_SOPDATASTATUSVALUES</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>PARTITION LIST SINGLE</td>
<td>SYS_C002645765</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>TABLE ACCESS BY LOCAL INDEX ROWID</td>
<td>XYZ_CODES</td>
<td>1</td>
<td>44</td>
<td>2</td>
<td>00:00:01</td>
<td>KEY</td>
<td>KEY</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>INDEX RANGE SCAN</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td>59</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PARTITION LIST SINGLE</td>
<td></td>
<td>1</td>
<td>28</td>
<td>2</td>
<td>00:00:01</td>
<td>KEY</td>
<td>KEY</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>TABLE ACCESS BY LOCAL INDEX ROWID</td>
<td>XYZ_CODES</td>
<td>1</td>
<td>28</td>
<td>2</td>
<td>00:00:01</td>
<td>KEY</td>
<td>KEY</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td>59</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>PARTITION LIST SINGLE</td>
<td></td>
<td>1</td>
<td>27</td>
<td>2</td>
<td>00:00:01</td>
<td>KEY</td>
<td>KEY</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>TABLE ACCESS BY LOCAL INDEX ROWID</td>
<td>XYZ_CODES</td>
<td>1</td>
<td>27</td>
<td>2</td>
<td>00:00:01</td>
<td>KEY</td>
<td>KEY</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td>59</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>TABLE ACCESS BY GLOBAL INDEX ROWID</td>
<td>XYZ_CODES</td>
<td>1</td>
<td>25</td>
<td>3</td>
<td>00:00:01</td>
<td>ROWID</td>
<td>ROWID</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The last section list all the predicate information. We wanted to make sure the effective filter has applied in the early stage of the query as much as possible, so the query cost can be as small as possible.

Let’s look at the explain plan output from top to bottom. Line ID0 gives out the query last cumulative time in estimation. Here it says it is 1.07hr. We navigate down to find the last line with time close to this, which is line ID111, with estimation query time 52:15min. According to this, if we can make this query block execution time dropped, the whole query time shall be dropped. Therefore we know this is where we need to start to work with.

We learned that there are 4 tables participated in this query block. First things first, we want to verify the 4 tables statistics are not stale. With dictionary view DBA_TAB_MODIFICATIONS, we can check if these tables are updated since last statistics gathered. If the table statistics is stale for some reason, we can refresh it with DBMS_STATS package. By default, oracle use 10% change rule to define if an object has stale statistics or not. However, in our practice, that is just a general rule of thumb, you have to learn from your system, every object could have different stale statistics threshold.

After we clear the tables’ statistics staleness, we move on to verify if the optimizer is correct by executing that query block.

**Analysis**

Now the question is what part of the query equivalent to this query block between line ID111 to line ID126 in the explain plan? Benefiting from the query formatting, this becomes much easier. The only little problem is those 4 tables don’t show up in the query. We know there are a few views being used in the query, so they must embedded in those views.
With this query, we can easily pull the view that those tables constructed with:

```sql
BEGIN
  FOR V
  IN (SELECT VIEW_NAME, TEXT
       FROM DBA_VIEWS
       WHERE OWNER = 'DICT_MGR'
       AND VIEW_NAME IN (UPPER ('vw_djw_categdoghierarchy_8_0'),
                          UPPER ('vw_djw_out101hierlist_8_0'),
                          UPPER ('vw_djw_sopout101s'),
                          UPPER ('vw_djw_exceptions')))
  LOOP
    IF INSTR (UPPER (V.TEXT), 'SOPDATALOADSTATUS') > 0
    AND INSTR (UPPER (V.TEXT), 'SOPUNMATCHEDDATA') > 0
    AND INSTR (UPPER (V.TEXT), 'SOP_EXCEPTIONS') > 0
    AND INSTR (UPPER (V.TEXT), 'SOPOUT101ITEMS') > 0
    THEN
      DBMS_OUTPUT.PUT_LINE (V.VIEW_NAME);
    END IF;
  END LOOP;
END;
/---------------
VW_DJW_EXCEPTIONS
```

With this we can find the query block (layer4) and execute it:

```sql
SELECT COUNT(*)
FROM (SELECT a4.1
      FROM vw_djw_exceptions a4
      WHERE sopout101code = 253355
      AND status NOT IN (91, -2)
      AND status NOT IN (99, 95, 94, 97)
      AND status NOT IN (11, 12, 15)
      AND hidden_exp = 0
      AND cutoff_exp = 0
      AND ppm_month = 549)
; COUNT(*)
----------
229021 rows
Elapsed: 23 sec.
```

Well, looks like optimizer’s estimation and the real scenario has a big discrepancy. This query block’s execution is fine.
We expend to its outer block:

```sql
SELECT CONT(*) FROM (  
SELECT lrowid,  
polid,  
  ...  
FROM  
  (SELECT *  
  FROM  
    (<layer4>  
      )  
  )  
GROUP BY lrowid,  
polid,  
  ...  
);  
COUNT(*)  
---------  
162869 rows  
Elapsed: 19sec
```

Again, the problem is not in the layer.

We extended to its outer block (layer 3):

```sql
SELECT COUNT(*) FROM(  
SELECT e3.*,  
  ...  
    h.out101code,  
  ...  
FROM  
  (<layer3>) e3,  
  (SELECT * FROM vw_djw_sopout101s WHERE sopout101code = 253355  
  ) h  
WHERE h.sopout101code(+) = e3.sopout101code  
AND h.out101division(+) = NVL (e3.out101division, '**')  
AND h.out101de8888mmm(+)= NVL (e3.out101de8888mmm, '**')  
AND h.out101subde8888mmm(+)= NVL (e3.out101subde8888mmm, '**')  
AND h.out101class(+)   = NVL (e3.out101class, '**')  
AND h.out101subclass(+) = NVL (e3.out101subclass, '**')  
);  
COUNT(*)  
---------  
162869 rows  
Elapsed: 6sec
```

This block runs fine.
Further extended out (layer2):

```
SELECT COUNT(*) FROM ( 
SELECT 
(SELECT code 
FROM xyz_codes 
WHERE bus466ssid = 471 
AND codeset = 'djw_a_subcategdog' 
AND NAME = categdogname 
) AS selected_categdog_code, 
e2.*, 
cat.*, 
OUT.* 
FROM (layer 2) 
)e2, 
vw_djw_categdoghierarchy_8_0 cat, 
vw_djw_out101hierlist_8_0 OUT 
WHERE 1 = 1 
AND e2.categdogcode = cat.code(+)
AND e2.out101code = OUT.out101_code(+) 
) ;
```

```
COUNT(*) 
----------
162869 rows
```

Elapsed: 7sec

Now we learn up to this query block there is no performance problem. The problem is sitting on the top layer.

### 10046 Trace and tkprof

Let’s generate a 10046 trace for the query. It provides the statistics for the real run. Before we start the trace, we need to make sure timed_statistics is turned to true. The statistics_level is set to typical. To make the trace file easier to be found, we set the tracefile_identifier. After that, we run the query. The trace file is generated in the same directory where the database instance alert.log is.

- ALTER SESSION SET TIMED_STATISTICS = TRUE;
- ALTER SESSION SET STATISTICS_LEVEL=TYPICAL;
- alter session set tracefile_identifier = 'SQL_badsql';
- alter session set events '10046 trace name context forever, level 12';
- <run query>
- alter session set events '10046 trace name context off';
- $ORACLE_BASE/diag/rdbms/<db>/<inst>/trace/<inst>_ora_xxxx_SQL_badsql.trc

Tkprof is the utility to interpret 10046 trace file. To get more details of this tool, you may refer to metalink doc: “tkprof Interpretation (9i and above)” (Doc ID 760786.1).
Here is how you get the tkprof output:

$ tkprof <inst>_ora_xxxx_SQL_badsql.trc  <inst>_ora_xxxx_SQL_badsql.out

This is the plan statistics captured of this query in the tkprof output:

```
Line: 1179, Top layer: elapsed time=1912696184 us or 32 min.
Line: 1180, 2nd layer: elapsed time=68106257 us or 1 min.

From the tkprof output, we can see the top layer in this run took 32min, while the second layer only took 1 min. This matches what we have found in the earlier analysis.

So now we know where the problem is. What is the fix? Obviously the join method of using nested loop is a wrong choice. We add a hint to force it to use hash join method. Since hint is using the query block alias, we can see the importance of labeling different query block with different alias.
```
SELECT /*+ USE_HASH(e a) */
  e.poiid  AS poiid,
FROM  
  (SELECT
e2.*,
cat.*,
OUT.*
FROM  
  (SELECT e3.*,  
    FROM  
    (SELECT lrowid,
poiid,
    FROM  
    (SELECT *
    FROM  
    (SELECT a4.*
    FROM vw_djw_exceptions a4  
    ) e4  
    )) e3,
    vw_djw_categdoghierarchy_8_0 cat,
    vw_djw_out101hierlist_8_0 OUT  
WHERE 1            = 1
AND e2.categdogcode = cat.code(+)
AND e2.out101code   = OUT.out101_code(+)
) e,
xyz_sopout101item_attributes a
WHERE e.sopout101code = a.sopout101(+)
AND e.poiid           = a.poiid(+);

Now we check the explain plan again:

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>TempSpc</th>
<th>Time</th>
<th>Pstart</th>
<th>Pstop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>118</td>
<td>11132 (2)</td>
<td></td>
<td>00:02:36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>118</td>
<td>11132 (2)</td>
<td></td>
<td>00:02:36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 2</td>
<td>HASH JOIN RIGHT OUTER</td>
<td></td>
<td>1498</td>
<td>172K</td>
<td>11132 (2)</td>
<td>00:02:36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 3</td>
<td>HASH JOIN OUTER</td>
<td></td>
<td>1498</td>
<td>172K</td>
<td>11132 (2)</td>
<td>00:02:36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>VIEW</td>
<td>vw_djw_out101hierlist_8_0</td>
<td>5</td>
<td>35</td>
<td></td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The estimation Time is 2:36 min., much more promising than the old one with Time as 1:07hr.
Then we run the query with the hint we added. It returns in a half minute. We can compare the 10046 trace file tkprof output here.

- **Tkprof - After**

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.51</td>
<td>0.58</td>
<td>0</td>
<td>48</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>2</td>
<td>16.18</td>
<td>31.17</td>
<td>16905</td>
<td>578400</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>4</td>
<td>16.69</td>
<td>31.76</td>
<td>16905</td>
<td>578448</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Rows (1st) Rows (avg) Rows (max) Row Source Operation

- **Tkprof - before**

<table>
<thead>
<tr>
<th>Line</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1179:</td>
<td>162869</td>
<td>162869</td>
<td>162869</td>
<td>15</td>
<td>SORT AGGREGATE (cr=578400 pr=16905 pw=6690 time=31172401 us)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1180:</td>
<td>162869</td>
<td>162869</td>
<td>162869</td>
<td>135</td>
<td>HASH JOIN OUTER (cr=578400 pr=16905 pw=6690 time=31278389 us cost=9967 size=171949 card=1607)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1181:</td>
<td>135</td>
<td>135</td>
<td>135</td>
<td>VIEW VW_DJW_OUT101HIERLIST_8_0 (cr=16 pr=0 pw=0 time=5953 us cost=3 size=28 card=4)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here is the new execution activities, some db file sequential read, some db file scattered read and some cpu used. Looks healthy compare to the previous activity, which is nothing but burning cpu!

---

**SQL Profile and Hint**

You may say, this is great, but what if we can’t change the query with hint?
We can use dbms_sqltune package to avoid code changes. (need diag+tuning pack licenses)

The idea is to create a sql profile with the hint we added, and attach the sql profile to the query.

This time we use a slight different syntax to retrieve the explain plan output.

```
explain plan set statement_id='f' for
<QUERY hinted>
    select * from table(dbms_xplan.display(null,null,'ADVANCED'));

or

<Run the query hinted>
select * from table(dbms_xplan.display _cursor(format => 'ADVANCED'));
```

We should find the outline section from the output report:

```
Outline Data
-----------------
/*+ BEGIN_OUTLINE_DATA
  IGNORE_OPTIM_EMBEDDED_HINTS
  OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
  DB_VERSION('11.2.0.4')
  OUTLINE_LEAF(@"SEL$19")
  OUTLINE_LEAF(@"SEL$43F09110")
  USE_HASH_AGGREGATION(@"SEL$43F09110")
  INDEX_RS_ASC(@"SEL$19" "XYZ_RET999ERS"@"SEL$19" ("XYZ_RET999ERS"."RET999ER_ID")
END_OUTLINE_DATA */
```

Then we use dbms_sqltune package to create a sql profile with the outline.

```
DECLARE
  l_sql               clob;
BEGIN
  l_sql := q'!select count(*) from ( 
    SELECT
      e
      .poiid  AS poiid,
      ...
    )' !;
  dbms_sqltune.import_sql_profile( sql_text => l_sql,
    profile => sqlprof_attr( 
      q'! IGNORE_OPTIM_EMBEDDED_HINTS!',
      q'! OPTIMIZER_FEATURES_ENABLE('11.2.0.3')!',
      q'! USE_HASH_AGGREGATION(@"SEL$43F09110")!',
      q'! INDEX_RS_ASC(@"SEL$19" "XYZ_RET999ERS"@"SEL$19" ("XYZ_RET999ERS"."RET999ER_ID")
    ),
    force_match => true
  );
end;
/*
Let's verify that the sql profile is generated and does work for the original query. As you can see, dbms_sqltune is doing a great job.
```
SQL> select * from table(dbms_xplan.display);

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>TempSpc</th>
<th>Cost (%CPU)</th>
<th>Time</th>
<th>Pstart</th>
<th>Pstop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1118</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_RPT_PPUS_PSTEDT_FLT_VALS_UNIQUE</td>
<td>1498</td>
<td>153K</td>
<td>21</td>
<td>1</td>
<td>00:00:36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>HASH JOIN RIGHT OUTER</td>
<td></td>
<td>1498</td>
<td>153K</td>
<td>21</td>
<td>1</td>
<td>00:00:36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>VIEW</td>
<td>VW_DJW_OUT101HIERLIST_8_0</td>
<td>5</td>
<td>35</td>
<td>1</td>
<td>6 (17)</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note
-- SQL profile "SQLPROFILE_01" used for this statement

Case Study 2

The Query has 107 lines, and it calls several functions.

Here is the query explain plan:

Plan hash value: 1228438049

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>TempSpc</th>
<th>Cost (%CPU)</th>
<th>Time</th>
<th>Pstart</th>
<th>Pstop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>9</td>
<td>100</td>
<td>21</td>
<td>1</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_TYPES</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>INDEX UNIQUE SCAN</td>
<td>LTE_TYPES_PK</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SORT UNIQUE</td>
<td></td>
<td>1</td>
<td>21</td>
<td>3</td>
<td>34</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_PPUS_CODE_DEFS</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>INDEX RANGE SCAN</td>
<td>LTE_PPUS_CODE_DEFS_PK</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_TYPES</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>INDEX UNIQUE SCAN</td>
<td>LTE_TYPES_PK</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>22</td>
<td>3</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_RPT_PROGRESSION_LEVELS</td>
<td>1</td>
<td>13</td>
<td>2</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>INDEX UNIQUE SCAN</td>
<td>LTE_RPT_PROGRESSION_LEVELS_PK</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_PPUS_FIELD_DEFS</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>INDEX UNIQUE SCAN</td>
<td>LTE_PPUS_FIELD_DEFS_PK</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>TABLE ACCESS BY GLOBAL INDEX ROWID</td>
<td>XYZ_CODES</td>
<td>1</td>
<td>25</td>
<td>3</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>INDEX UNIQUE SCAN</td>
<td>XYZ_CODES_UK01</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SORT ORDER BY</td>
<td></td>
<td>9</td>
<td>3528</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>00:00:02</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>HASH JOIN OUTER</td>
<td></td>
<td>9</td>
<td>3528</td>
<td>99</td>
<td>0</td>
<td>00:00:02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>NESTED LOOPS</td>
<td></td>
<td>3</td>
<td>1029</td>
<td>96</td>
<td>0</td>
<td>00:00:02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>HASH JOIN OUTER</td>
<td></td>
<td>1</td>
<td>289</td>
<td>94</td>
<td>0</td>
<td>00:00:02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>237</td>
<td>82</td>
<td>0</td>
<td>00:00:02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>NESTED LOOPS</td>
<td></td>
<td>8</td>
<td>237</td>
<td>82</td>
<td>0</td>
<td>00:00:02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>NESTED LOOPS</td>
<td></td>
<td>8</td>
<td>1792</td>
<td>74</td>
<td>0</td>
<td>00:00:02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>82</td>
<td>4</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>73</td>
<td>3</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>61</td>
<td>2</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_PPUS_FIELD_DEFS</td>
<td>1</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>INDEX UNIQUE SCAN</td>
<td>LTE_PPUS_FIELD_DEFS_PK</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_RPT_PROGRESSION_RULES</td>
<td>1</td>
<td>56</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>INDEX UNIQUE SCAN</td>
<td>LTE_RPT_PROGRESSION_RULES_PK</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>INDEX RANGE SCAN</td>
<td>LTE_RPT_RULESET_VERSION_PK</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_RULESETS</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>INDEX UNIQUE SCAN</td>
<td>LTE_RULESETS_PK</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>TABLE ACCESS FULL</td>
<td>LTE_RPT_PPUS_PSTEDT_RULES</td>
<td>8</td>
<td>1156</td>
<td>78</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>INDEX UNIQUE SCAN</td>
<td>LTE_RPT_PROGRESSION_LEVELS_PK</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_RPT_PROGRESSION_LEVELS</td>
<td>1</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>TABLE ACCESS FULL</td>
<td>LTE_RPT_PPUS_PSTEDT_CASES</td>
<td>5781</td>
<td>2093</td>
<td>12</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_RPT_PPUS_PSTEDT_FLT_VALUES</td>
<td>3</td>
<td>162</td>
<td>2</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>INDEX RANGE SCAN</td>
<td>PPUS_PSTEDT_FLT_VALS_UNIQUE</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>LTE_RPT_PPUS_PSTEDT_FIELD_DEFS</td>
<td>3</td>
<td>147</td>
<td>3</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

<table>
<thead>
<tr>
<th>Operations</th>
<th>Name</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LTE_RPT_PPUS_PSTEDT_FLT_VALS_UNIQUE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LTE_RPT_PPUS_PSTEDT_CASES</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LTE_PPUS_CODE_DEFS</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LTE_RPT_PROGRESSION_LEVELS</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LTE_PPUS_FIELD_DEFS</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LTE_PPUS_FIELD_DEFS_PK</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>LTE_PPUS_FIELD_DEFS</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LTE_PPUS_FIELD_DEFS_PK</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>XYZ_CODES_UK01</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>LTE_RPT_PROGRESSION_LEVELS</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LTE_PPUS_CODE_DEFS</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>LTE_PPUS_FIELD_DEFS</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>LTE_PPUS_FIELD_DEFS_PK</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>XYZ_CODES</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>LTE_PPUS_FIELD_DEFS</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>LTE_PPUS_FIELD_DEFS_PK</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>LTE_PPUS_CODE_DEFS</td>
<td></td>
</tr>
</tbody>
</table>

Document classification: Internal Only
Here is 10046 trace:

```
<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.02</td>
<td>0.02</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>36</td>
<td>0.17</td>
<td>0.17</td>
<td>0</td>
<td>1529</td>
<td>11404</td>
<td>35</td>
</tr>
<tr>
<td>total</td>
<td>38</td>
<td>0.20</td>
<td>0.20</td>
<td>0</td>
<td>1532</td>
<td>11404</td>
<td>35</td>
</tr>
</tbody>
</table>
```

The Query plan doesn’t show any problem. The 10046 trace says it returns in 0.2 sec. Where is the time spent? We need to look at all parts of the 10046 trace this time.

At the bottom of the tkprof report there is a section with the overall total for non-recursive statements and recursive statements executed by the query:

```
<table>
<thead>
<tr>
<th>Overall totals for all non-recursive statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>Parse</td>
</tr>
<tr>
<td>Fetch</td>
</tr>
<tr>
<td>total</td>
</tr>
</tbody>
</table>
```

The top part is from our query. It executed once, fetched 35 rows, spent 0.2 sec. However let’s look at the bottom part. This part sums up all the recursive calls by the query. The total time is about 2 min. What is this?

Let’s look through the total time spend of all the calls in 10046 trace file.
We found the top time consuming call. Line 627 in 10046 trace file shows its total time spent is almost the same as the total recursive calls in line 1137. From line 588 to line 627 in 10046 trace file is this call’s detail.

From this we can see that this call was executed 350 times, although consumed 143 seconds, but no rows returns. This is found later from one of the function call embedded in the query, and is determined the call is unnecessary. This function call was removed from the query, problem is resolved.
Summary:

- Take advantage of the available diagnostic and tuning tools
- Don’t forget the fundamental tool kit:
  - Top
  - Explain plan
  - 10046 trace and tkprof
  - Optimizer hint and SQL profile
- Prerequisite: Statistics is up to date