Identifying SQL execution bottlenecks scientifically

Few days ago, a developer and I had an interesting conversation. Developer was trying to tune an expensive SQL statement, using following trial and error method:

```
loop until acceptable performance
        explain plan -> execute SQL with sql trace -> tkprof -> rewrite
end loop;
```

After looking at his trial and error method in amusement, I introduced him how to identify and tune SQL statements scientifically and decided to blog about it.

Let's look at a simple case and then proceed to slightly more complex versions. Following code fragment is creating test tables, indices and collecting statistics on those tables.

```
create table t1_vc as
select trunc(n/10000) n1, mod(n, 1000) n2 , lpad( n,255) c_filler
from (select level n from dual connect by level <= 100001);

create index t1_vc_i1 on t1_vc (n1);

create table t2_vc as
select trunc(n/ 100) n1, mod(n, 10000) n2 ,  lpad( n,255) c_filler
from (select level n from dual connect by level <= 100001);

create index t2_vc_i1 on t2_vc (n1);

exec dbms_stats.gather_table_stats(user, 't1_vc',estimate_percent => null, cascade => true);
exec dbms_stats.gather_table_stats(user, 't2_vc',estimate_percent => null, cascade => true);
```

Simple SQL, but I had to use hints to illustrate the point I am driving at. Let's do an explain plan on this SQL.

```
 explain plan for
   select /*+ use_nl (t1_vc, t2_vc ) */
        t1_vc.n1 , t2_vc.n2
   from  t1_vc, t2_vc where
   t1_vc.n1 = t2_vc.n1 and t1_vc.n2 between 101 and 105 and t1_vc.n1=1
/
select * from table(dbms_xplan.display)
/
```

```
plan hash value: 3808913109

-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |          |  5453 | 81795 |   643   (0)| 00:00:08 |
|   1 |  NESTED LOOPS                 |          |       |       |            |          |
|   2 |   NESTED LOOPS                |          |  5453 | 81795 |   643   (0)| 00:00:08 |
|*  3 |    TABLE ACCESS BY INDEX ROWID| T1_VC    |    55 |   385 |   368   (0)| 00:00:05 |
|*  4 |     INDEX RANGE SCAN          | T1_VC_I1 |  9091 |       |    18   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN           | T2_VC_I1 |   100 |       |     1   (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID | T2_VC    |   100 |   800 |     5   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

   3 - filter("T1_VC"."N2"<=105 AND "T1_VC"."N2">=101)
   4 - access("T1_VC"."N1"=1)
   5 - access("T2_VC"."N1"=1)

20 rows selected.
```

Execution plan looks favorably okay. But, this statement is executed millions of times, so we need to reduce time as much as possible. Can this SQL be tuned further?

Statistics_level

Enter statistics_level parameter available from Oracle version 9i onwards. Step level execution statistics are printed at each step if this parameter is set to all. Using this method, to tune an SQL, identify the step taking most time and reduce time in that step or completely eliminate it. statistics_level parameter is session modifiable and set to all to print more statistics in the trace file.

[My recommendation is not to modify this parameter at instance or database level, without extensive testing]

Let's enable trace and statistics_level parameter in our session, followed by tkprof. Event 10046 is used to enable sql trace. Other methods can be used to turn on sql trace as well.

```
alter session set events '10046 trace name context forever, level 12';
alter session set statistics_level=all;

select /*+ use_nl (t1_vc, t2_vc ) */ t1_vc.n1 , t2_vc.n2 from
t1_vc, t2_vc where
t1_vc.n1 = t2_vc.n1 and t1_vc.n2 between 101 and 105 and t1_vc.n1=1;
```

tkprof orcl11g_ora_2988.trc orcl11g_ora_2988.trc.out

Following lines are from tkprof output file generated above.

```
select /*+ use_nl (t1_vc, t2_vc ) */ t1_vc.n1 , t2_vc.n2 from
 t1_vc, t2_vc where
 t1_vc.n1 = t2_vc.n1 and t1_vc.n2 between 101 and 105 and t1_vc.n1=1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
```

```
Execute      1      0.00       0.00          0          0          0           0
Fetch      335      0.35       0.34          1       1438          0        5000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      337      0.35       0.34          1       1438          0        5000

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows     Row Source Operation
-------  ---------------------------------------------------
   5000  NESTED LOOPS  (cr=1438 pr=1 pw=1 time=273471 us).....................................................(5)
   5000   NESTED LOOPS  (cr=871 pr=1 pw=1 time=175734 us cost=643 size=81795 card=5453).....................(3)
     50    TABLE ACCESS BY INDEX ROWID T1_VC (cr=482 pr=0 pw=0 time=117421 us cost=368 size=385 card=55).....(2)
  10000     INDEX RANGE SCAN T1_VC_I1 (cr=48 pr=0 pw=0 time=21970 us cost=18 size=0 card=9091)(object id 71043)...(1)
   5000    INDEX RANGE SCAN T2_VC_I1 (cr=389 pr=1 pw=1 time=35440 us cost=1 size=0 card=100)(object id 71041).....(4)
   5000   TABLE ACCESS BY INDEX ROWID T2_VC (cr=567 pr=0 pw=0 time=0 us cost=5 size=800 card=100).............(6)
```

Let me explain this output. Execution plan printed above have a time component and shows how time is accumulated in each step. At step (#1) 21970 micro seconds consumed, followed by step (2) at which cumulative time consumption is 117,421 micro seconds. At step (3),a nested loops join between row sources at step (2) and step (4), consumed a cumulative time of 175,734 micro seconds.

Also, note that step (4) contributed to a time consumption of 35,440 micro seconds. In essence, cumulative time consumption is printed at parent nodes and time consumption at that step is printed in leaf nodes of execution tree.

To tune this SQL, we need to look for a step with a biggest jump in time consumption or the step consuming much time. Then reduce or eliminate time spent in that step.

Let's examine following few lines again.

```
     50    TABLE ACCESS BY INDEX ROWID T1_VC (cr=482 pr=0 pw=0 time=117421 us cost=368 size=385 card=55).....(2)
  10000     INDEX RANGE SCAN T1_VC_I1 (cr=48 pr=0 pw=0 time=21970 us cost=18 size=0 card=9091)(object id 71043)..(1)
```

At step (1) Index t1_vc_i1 is scanned for rows with n1=1 and 10,000 rows returned. It took 21,970 micro seconds in that step. Next step (2), accesses table block using rowids returned from the index. Cumulative time consumption jumped from 21,970 micro seconds to 117,421 micro seconds. This is a costlier step and to tune this SQL, we need to consider tuning these two steps first.

Now, we have scientifically identified which step needs to be tuned. Note that step (1) fetched 10,000 rows. Step (2) is to access t1_vc table and 50 rows were retrieved in step (2). In summary, 10000 rows were returned scanning the index and 9050 rows filtered out after accessing table block. There seems to be quite a waste here.

Is it possible to apply that filter in accessing index itself? We need to add an index so that filtering can be done more efficiently at index block itself. Rest is easy, we can add index on n2 and n1.

```
create index t1_vc_i2 on t1_vc (n2,n1);
exec dbms_stats.gather_table_stats(user, 't1_vc',estimate_percent => null, cascade => true);
```

Explain plan printed below and new index shows up in step 3 below.

```
Plan hash value: 3827863167

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

```
|   0 | SELECT STATEMENT             |           |  5998 | 89970 |   206   (1)| 00:00:03 |
|   1 |  NESTED LOOPS                |           |  5998 | 89970 |   206   (1)| 00:00:03 |
|   2 |   TABLE ACCESS BY INDEX ROWID| T2_VC     |   100 |   800 |     5   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | T2_VC_I1  |   100 |       |     1   (0)| 00:00:01 |
|*  4 |   INDEX RANGE SCAN           | T1_VC_I2  |    60 |   420 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   3 - access("T2_VC"."N1"=1)
   4 - access("T1_VC"."N2">=101 AND "T1_VC"."N1"=1 AND "T1_VC"."N2"<=105)
       filter("T1_VC"."N1"=1)
```

Tracing with statisics_level=all again, shows that we have reduced time spent in that step.

```
   select /*+ use_nl (t1_vc, t2_vc ) */ t1_vc.n1 , t2_vc.n2 from
        t1_vc, t2_vc where
    t1_vc.n1 = t2_vc.n1 and t1_vc.n2 between 101 and 105 and t1_vc.n1=1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      335      0.10       0.22          7        672          0        5000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      337      0.10       0.22          7        672          0        5000

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows     Row Source Operation
-------  ---------------------------------------------------
   5000  NESTED LOOPS  (cr=672 pr=7 pw=7 time=76861 us cost=206 size=89970 card=5998)
    100   TABLE ACCESS BY INDEX ROWID T2_VC (cr=133 pr=7 pw=7 time=8622 us cost=5 size=800 card=100)
    100    INDEX RANGE SCAN T2_VC_I1 (cr=33 pr=2 pw=2 time=827 us cost=1 size=0 card=100)(object id 71041)
   5000   INDEX RANGE SCAN T1_VC_I2 (cr=539 pr=0 pw=0 time=42568 us cost=2 size=420 card=60)(object id 71052)
```

In a nutshell, if you must tune an SQL, use statistics_level and understand where the bottleneck is. Remove or tune that bottleneck to tune the SQL.

More complex scenarios

As the complexity of SQL increases [as in real world] this method is very useful. Consider following query: Using time column printed in this explain plan, you could guess that step with id 7 is consuming much time. But that could be wrong, since explain plan is printing estimates from CBO, not actual execution statistics. Execution plans with numerous table joins have incorrect cardinality estimates and so, any knowledge gained from explain plan alone is not that useful. Even autotrace suffers from few such issues**.**

```
    select /*+ use_nl (t1_vc, t2_vc ) */ t1_vc.n1 , t2_vc.n2 from
      t1_vc, t2_vc where
    t1_vc.n1 = t2_vc.n1 and t1_vc.n2 between 101 and 105 and t1_vc.n1=1
    union
    select /*+ use_nl (t1_vc, t2_vc ) */ t1_vc.n1 , t2_vc.n2 from
      t1_vc, t2_vc where
    t1_vc.n1 = t2_vc.n1 and t1_vc.n1 between 1 and 10;

Plan hash value: 3076824877

-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name     | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |          |   153K|  1674K|       | 19777  (99)| 00:03:58 |
|   1 |  SORT UNIQUE                  |          |   153K|  1674K|  6113K| 19777  (99)| 00:03:58 |
|   2 |   UNION-ALL                   |          |       |       |       |            |          |
|   3 |    NESTED LOOPS               |          |  5998 | 89970 |       |   206   (1)| 00:00:03 |
|   4 |     TABLE ACCESS BY INDEX ROWID| T2_VC    |   100 |   800 |       |     5   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN         | T2_VC_I1 |   100 |       |       |     1   (0)| 00:00:01 |
|*  6 |     INDEX RANGE SCAN          | T1_VC_I2 |    60 |   420 |       |     2   (0)| 00:00:01 |
|   7 |    NESTED LOOPS               |          |   147K|  1587K|       | 18886   (1)| 00:03:47 |
|   8 |     TABLE ACCESS BY INDEX ROWID| T2_VC    |  1100 |  8800 |       |    47   (0)| 00:00:01 |
|*  9 |      INDEX RANGE SCAN         | T2_VC_I1 |  1100 |       |       |     4   (0)| 00:00:01 |
|* 10 |     INDEX RANGE SCAN          | T1_VC_I1 |   134 |   402 |       |    17   (0)| 00:00:01 |
```

```
-------------------------------------------------------------------------------------------------------

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

   5 - access("T2_VC"."N1"=1)
   6 - access("T1_VC"."N2">=101 AND "T1_VC"."N1"=1 AND "T1_VC"."N2"<=105)
       filter("T1_VC"."N1"=1)
   9 - access("T2_VC"."N1">=1 AND "T2_VC"."N1"<=10)
  10 - access("T1_VC"."N1"="T2_VC"."N1")
       filter("T1_VC"."N1">=1 AND "T1_VC"."N1"<=10)

27 rows selected.
```

But let's look at step level timing information printed below. Second nested loop branch consumed 53 seconds [as against 3 minutes and 47 seconds in the plan printed above] and 'UNION ALL' step consumed 58 seconds. So, to tune this SQL, we need to find ways to eliminate waste or improve efficiency of operation.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.20          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       68    130.03     134.76        185      53178          0        1000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       70    130.03     134.96        185      53178          0        1000

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows     Row Source Operation
-------  ---------------------------------------------------
   1000  SORT UNIQUE (cr=53178 pr=185 pw=185 time=2466 us cost=19777 size=1715088 card=153736)
9005200   UNION-ALL  (cr=53178 pr=185 pw=185 time=116316614 us)
   5000    NESTED LOOPS  (cr=117 pr=0 pw=0 time=41606 us cost=206 size=89970 card=5998)
    100     TABLE ACCESS BY INDEX ROWID T2_VC (cr=7 pr=0 pw=0 time=1041 us cost=5 size=800 card=100)
    100      INDEX RANGE SCAN T2_VC_I1 (cr=2 pr=0 pw=0 time=216 us cost=1 size=0 card=100)(object id 71041)
   5000     INDEX RANGE SCAN T1_VC_I2 (cr=110 pr=0 pw=0 time=21900 us cost=2 size=420 card=60)(object id 71052)
9000200    NESTED LOOPS  (cr=53061 pr=185 pw=185 time=53362636 us cost=18886 size=1625118 card=147738)
   1000     TABLE ACCESS BY INDEX ROWID T2_VC (cr=48 pr=29 pw=29 time=206052 us cost=47 size=8800 card=1100)
   1000      INDEX RANGE SCAN T2_VC_I1 (cr=8 pr=0 pw=0 time=3785 us cost=4 size=0 card=1100)(object id 71041)
9000200     INDEX RANGE SCAN T1_VC_I1 (cr=53013 pr=156 pw=156 time=21562840 us cost=17 size=402 card=134)(object id
71043)


Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                      68        0.00          0.00
  db file sequential read                       185        0.13          0.70
  SQL*Net message from client                    68        4.19          4.46
```

Issues

1. If time is spent in the column list, then these numbers are not accurate. Example below: This SQL consumed over 150 seconds, but not reflected correctly in the plan. Seemingly this happens if time is spent in function calls from select list.

<pre>
<font size="2" face="Lucida Console">

```
create or replace function func1
return number is
  n1 number;
begin
  select count(*)  into n1 from t1_vc where n1=1;
  return n1;
end;
/
```

```
select /*+ use_nl (t1_vc, t2_vc ) */ func1, t1_vc.n1 , t2_vc.n2 from
 t1_vc, t2_vc where
 t1_vc.n1 = t2_vc.n1 and t1_vc.n2 between 101 and 105 and t1_vc.n1=1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      320      1.51       1.62          0        709          0        4786
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      322      1.53       1.63          0        709          0        4786

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows     Row Source Operation
-------  ---------------------------------------------------
   4786  NESTED LOOPS  (cr=709 pr=0 pw=0 time=80756 us cost=206 size=89970 card=5998)
     96   TABLE ACCESS BY INDEX ROWID T2_VC (cr=193 pr=0 pw=0 time=3023 us cost=5 size=800 card=100)
     96    INDEX RANGE SCAN T2_VC_I1 (cr=97 pr=0 pw=0 time=1377 us cost=1 size=0 card=100)(object id 71041)
   4786   INDEX RANGE SCAN T1_VC_I2 (cr=516 pr=0 pw=0 time=53983 us cost=2 size=420 card=60)(object id 71052)
```

2. It is not possible to turn on this parameter on a session, which is executing already.

3. If the SQL execution time is very small, then this parameter is not printing step level information correctly.