How One Should Help the Cost Based Optimizer

Jože Senegačnik
Member of OakTable and Oracle ACE Director
DbProf d.o.o.
Smrjene 153, 1291 Škofljica
Slovenia

Keywords:

optimizer cost of execution of PL/SQL function, cardinality feedback

Abstract:

Everybody likes automatics, but what one should do when the automatics fails? The Cost Based Optimizer (CBO) prepares performant execution plans most of the time, however from time to time it still fails to do a correct job. Usually this failure is a consequence of the fact that the CBO doesn't have enough knowledge about our data and therefore we get sub-optimal execution plan. In this presentation we will discuss the possible ways how one can help the CBO to get more knowledge about our data and how this influences the execution plan preparation.

Introduction:

Recently I was involved in one upgrade project upgrading one production database from 10.2.0.4 on IBM Aix to 11.2.0.2 on Linux. It was not a simple upgrade project which is usually finished in few days but was from the very beginning expected as a tough project because there were many problems with execution plan instability experienced during migration from 9i to 10g and the only way to retain good execution plans after upgrading from 9i to 10.2.0.3 was to set the parameter OPTIMIZER_FEATURES_ENABLE='9.2.0.8'.

So my plan was to create **stored outlines** on 10gR2 and migrate them on 11gR2 (11.2.0.2) and run the critical parts of the application, especially the batch process which were the most critical one, create SQL Plan Baselines and enable optimizer features for the current release (11.2.0.2). I also tried to convert outlines to baselines but the attempt failed as the conversion didn't go according to my expectations and the performance of critical SQL statements was not as it was in Oracle 10g.

The reason why I am explaining this lies in the fact that during the upgrade process there were quite limited options available to change the code so any kind of SQL statement tuning by inserting hint was out of the question.

Let me first start with the SQL Plan Management mechanism, a new feature in 11g.

SQL Plan Baselines

SQL Plan Baselines are a new mechanism introduced in 11.1 which guarantees plan stability and newly created execution plans are used only after they are accepted (evolved) and therefore proved to perform better than their ancestor. The mechanism behind is a successor of stored outlines introduced in Oracle 8i.
They are controlled by two init.ora parameter

- **optimizer_capture_sql_plan_baselines**
  - Controls auto-capture of SQL plan baselines for repeatable statements
  - Set to false by default in 11gR1 (system and session modifiable)

- **optimizer_use_sql_plan_baselines**
  - Controls the use of existing SQL plan baselines by the optimizer
  - Set to true by default in 11gR1 (system and session modifiable)

SQL Plan Baselines are visible in view **DBA_SQL_PLAN_BASELINE.** Package DBMS_SPM is used in background for managing SQL Plans

Of course there are, as always, some hidden parameters involved:

- **_evolve_plan_baseline_report_level** - Level of detail to show in plan verification/evolution report – default = "typical"
- **_plan_outline_data** - explain plan outline data enabled default=true
- **_plan_verify_improvement_margin** - Performance improvement criterion for evolving plan baselines default=150
- **_plan_verify_local_time_limit** - Local time limit to use for an individual plan verification default=0 (unlimited)
- **_sql_plan_baseline_capture_on_1_exec** - With auto-capture on, create new SQL plan baseline on first exec – default=false

- SQL statement can have:
  - SPM Baseline
  - SQL Profile
    - SQL Profile provides additional information to the CBO for scaling up/down the statistical data (base cardinality, join cardinality,...)
  - When SQL Profile is present CBO may choose different accepted plan.
  - If you have a fixed SQL Plan Baseline and you add a SQL Profile to the statement you have to manually change fixed attribute of the evolved SQL statement with SQL Profile because fixed attribute prevents generating new SQL Plan baselines.

While I was running critical parts of the applications after the database was already upgraded to 11.2.0.2, I was aware of the fact that if an outline is created and the parameter **USE_STORED_OUTLINES**=TRUE the automatic plan capture will not work for that SQL statement.

So potentially I had two options:

a.) Use the following process

1. Prepare the tuning set through **CAPTURE_CURSOR_CACHE_SQLSET** in DBMS_SQLTUNE package.
2. Create baseline from tuning set.
3. Disable using stored outlines (use_stored_outlines=FALSE)
4. SQL Plan management will now handle these statements as well.

b.) Create SQL Plan Baselines by manually loading critical SQL plans from cursor cache (V$SQL)

The second method was my preferred method as the most critical SQL statements were related to refreshing materialized views just prior and during the critical batch processes and those statements were easily identified.
Therefore I used the following code for manual load:

```
set serveroutput on
declare b binary_integer;
begin
  for a in
       (select sql_id,sql_text
        from v$sql
        where sql_text like '%BYPASS%') loop
     b := dbms_spm.LOAD_PLANS_FROM_CURSOR_CACHE(a.sql_id);
     dbms_output.put_line(to_char(b)||' sql_text: '||a.sql_text);
  end loop;
end;
/
```

When the materialized views are refreshed the SQL statement is hinted with the hint **BYPASS_RECURSIVE_CHECK** so this was the trick used to identify them.
I have deliberately decided not to create any other baselines based on the compatibility parameter OPTIMIZER_FEATURES_ENABLE set to 9.2.0.8 and give the latest version of the optimizer the chance to prepare optimal execution plans.

The alternative was to manually capture all the SQLs executed by the batch process and the following code was run several times during the run. However after short analysis it was found unnecessary and therefore only performance critical SQL Statements were loaded from cursor cache.

```
set serveroutput on
declare b binary_integer;
begin
  for a in
       (select sql_id,sql_text
       from v$sql
       where PARSING_SCHEMA_NAME='XXXXX') loop
     b := dbms_spm.LOAD_PLANS_FROM_CURSOR_CACHE(a.sql_id);
     dbms_output.put_line(to_char(b)||' sql_text: '||a.sql_text);
  end loop;
end;
/
```

For certain statements which were not performing according to the expectations SQL Profiles were created and accepted and later on the baselines were manually created:

```
set serveroutput on
declare b binary_integer;
begin
     b := dbms_spm.LOAD_PLANS_FROM_CURSOR_CACHE('&sql_id');
     dbms_output.put_line(to_char(b));
end;
/
```

The batch process was repeated several times and after each run the following code was used to evolve newly created SQL Plan Baselines:

```
SET SERVEROUTPUT ON
SET LONG 800000
DECLARE
  report clob;
BEGIN
  for a in
       (select plan_name
        from dba_sql_plan_baselines
        where enabled='YES'
          and accepted='NO'
          and sql_text not like '%BYPASS%') loop
    report := DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE(
                  plan_name=>a.plan_name);
    DBMS_OUTPUT.PUT_LINE(report);
  end loop;
END;
/
```

It was interesting that the DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE was not able to evolve the plans related to refreshing materialized views. Here is the output:

```
Enter value for sql_plan: SQL_PLAN_d3qcgtcrxfy3s7c6cc440

-------------------------------------------------------------------------------
Evolve SQL Plan Baseline Report
-------------------------------------------------------------------------------

Inputs:
-------
  SQL_HANDLE =
  PLAN_NAME  = SQL_PLAN_d3qcgtcrxfy3s7c6cc440
  TIME_LIMIT = DBMS_SPM.AUTO_LIMIT
  VERIFY     = YES
  COMMIT     = YES

Plan: SQL_PLAN_d3qcgtcrxfy3s7c6cc440
------------------------------------
  Plan was verified: Time used 9,85 seconds.
  Error encountered during plan verification (ORA-1732).
    ORA-01732: data manipulation operation not legal on this view
```

The explanation for this is simple. The SQL Performance Analyzer is involved in the evolving process. Although DML statements can be evolved normally and the results are never committed the limitation on materialized views was not bypassed. This limitation might change in future.

And finally let us look how the SQL Plan Baseline looks like. The actual baseline is stored in SYS.SQLOBJ$DATA

```
<outline_data>
   <hint><![CDATA[INDEX_RS_ASC(@"SEL$1" "T"@"SEL$1" ("T"."ID"))]]></hint>
   <hint><![CDATA[OUTLINE_LEAF(@"SEL$1")]]></hint>
   <hint><![CDATA[ALL_ROWS]]></hint>
   <hint><![CDATA[DB_VERSION('11.1.0.6')]]></hint>
   <hint><![CDATA[OPTIMIZER_FEATURES_ENABLE('11.1.0.6')]]></hint>
   <hint><![CDATA[IGNORE_OPTIM_EMBEDDED_HINTS]]></hint>
</outline_data>
```

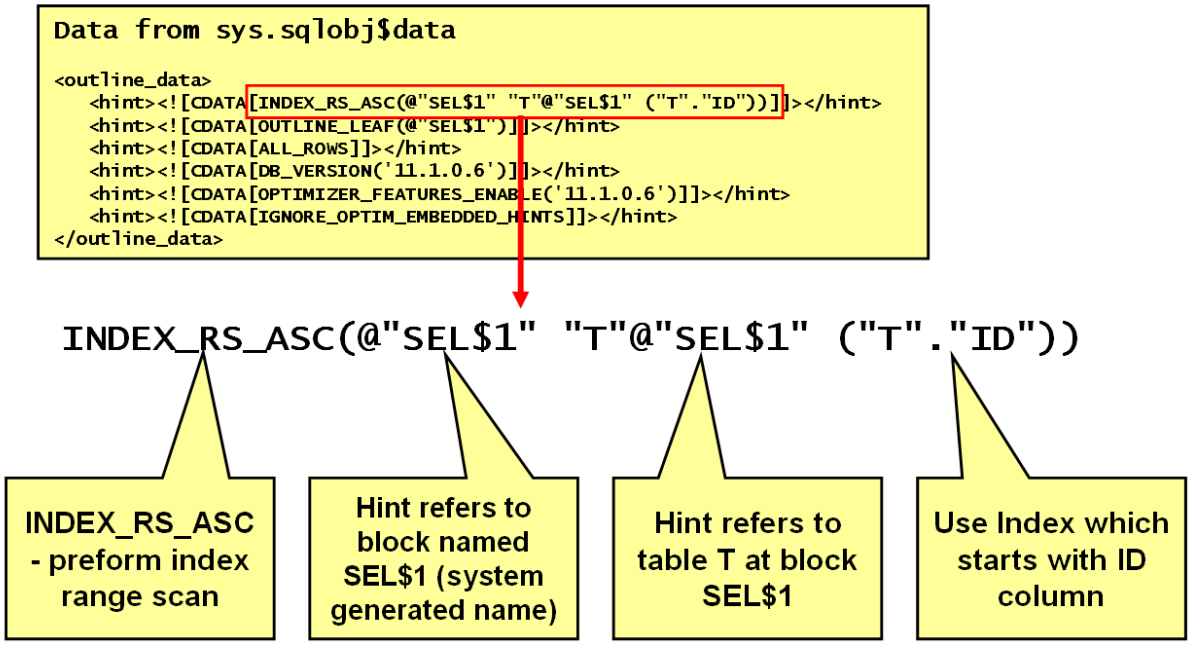
If we look in CBO Trace file, produced by setting event 10053, we can see that the baseline is identical as the outline reported in the CBO trace file for particular SQL statement. Here is the excerpt from CBO trace file for the same SQL statement for which we have retrieved the baseline above.

```
SPM: cost-based plan found in the plan baseline, planId = 284539257
SPM: cost-based plan was successfully matched, planId = 284539257
.....
============
Plan Table
============
--------------------------------------------+---------------------------------+
| Id  | Operation                   | Name    | Rows  | Bytes | Cost  | Time      |
--------------------------------------------+---------------------------------+
| 0   | SELECT STATEMENT            |         |       |       |     2 |           |
| 1   |  TABLE ACCESS BY INDEX ROWID | T       |     1 |   254 |     2 |  00:00:01 |
| 2   |   INDEX RANGE SCAN          | T_I1    |     1 |       |     1 |  00:00:01 |
--------------------------------------------+---------------------------------+
Predicate Information:
----------------------
2 - access("ID"=30)

Content of other_xml column
===========================
  db_version     : 11.1.0.6
  parse_schema   : JOC
  plan_hash      : 1977792910
  plan_hash_2    : 284539257
  Outline Data:
  /*+
    BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.1.0.6')
      DB_VERSION('11.1.0.6')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      INDEX_RS_ASC(@"SEL$1" "T"@"SEL$1" ("T"."ID"))
    END_OUTLINE_DATA
  */
```

It is obvious that the outlines/baselines use **extended global hint syntax** using the following rules:

- Every query block is uniquely named in >=10g
- Names are system-generated or hinted (using new QB_NAME hint)
- System-generated names contain two parts:
  - fixed prefix based on query block type: DEL$, INS$, MRG$, SEL$, UPD$, CRI$, SET$, MISC$
  - followed by alphanumeric string (up to 8 characters long) e.g. SEL$1, SEL$A5FF74C1, etc.
- Global hints can be specified in any query block, not just the one they target.



System Statistics

One of the features which is usually not used, but is by my opinion and experience a "must" is gathering workload system statistics. The system statistics was introduced in 9i when the "new" (at that time) cost model for execution plans was introduces which does not include only the I/O but also estimates the CPU time and in the final step converts the estimated CPU time to the I/O units.
Since 9i I am using this feature with the best possible results. Usually people don't know that unless you gather real workload system statistics the optimizer will use some kind of "defaults" which never good. Therefore the system statistics tells the optimizer the actual times that this particular database needs for a single or multi block I/O, how fast actually it the CPU and what are the system throughputs. The MBRC parameter tells the optimizer how many blocks in average can be read in one multi block operation what has a direct impact on the cost calculation for FULL TABLE SCAN operations.

Since 10g by default the CBO uses so called NOWORKLOAD statistics with some assumptions which are not too bad, however they are probably far away from the actual workload statistics for your system.

For gathering system statistics the most appropriate time is the typical workload and one can use the dbms_stats.gather_system_stats('start') and ('stop') procedure. To view the system stats one can run the following query:

```
SQL> select pname,pval1 from sys.aux_stats$;

PNAME                          PVAL1
------------------------------ ----------
SREADTIM                        26235,385
MREADTIM                       169522,145
CPUSPEED                             1848
MBRC                                   58
```

This query was run on 11.2.0.2 system after gathering the workload statistics during the batch run. It is interesting to see that the values for single block read time (SREATIM) and multi block read time (MREADTIM) are not in milliseconds as the documentation states. The average SREADTIM from v$system_wait was about 2 milliseconds. One can also see that the average block read in a multi block operation is 58 what lowers the cots for the full table scan operations. However, the MREADTIM is about 8 times bigger than SREADTIM what is quite logical as 58 blocks are read in one I/O.

The system statistics directly influences cost calculation and therefore helps the CBO to choose plans with the lowest cost which are most likely performing in the optimal way.

So my advice after so many years of experience with the system statistics is that it should always gathered as workload statistics. System statistics will help the CBO to use indexes in the right way and when it is gathered the "famous" OPTIMIZER_INDEX_COST_ADJ parameter should be set to its default value of 100. In fact, one can trick the CBO to heavily use index access paths by manually increasing the value of MREADTIM statistics and decreasing the value of MBRC. For me, this is the only correct way how to influence the CBO to "enforce" index access paths.

Object Statistics and Histograms

There are many new features in recent versions of Oracle database regarding gathering and using object statistics.
In past years I was involved in many optimization projects and one of the simple rules which I use always is that the system should automatically gather statistics. Of course there are always some exceptions from this rule when I apply the following remedies:

1.) If histogram is gathered on a column, where there should be no histogram, then don't let the system to gather it.
2.) If the number of buckets in histogram is not the appropriate one and the statistics should be gathered with a different number of histograms, then this rule should be applied for all future gatherings.
3.) Use extended statistics on several columns in all cases when the columns are dependent on each other.

Automatic Cardinality Feedback Tuning

Automatic tuning of SQL statements with the cardinality feedback from previous executions is a new feature of 11gR2. Every statement is monitored during the execution. Actually all statements which run longer than 5 seconds are monitored through SQL Monitor utility and one can see the results of ongoing execution in V$SQL_MONITOR and V$SQL_PLAN_MONITOR dynamic views.

If the CBO sees that the actual cardinality of row sources significantly differ from those estimated during the parse and optimization phase, the statement undergoes another hard parse during which the feedback cardinalities are used to achieve execution plan which performs better.

This feature was introduced in 11.2.0.1 and only in 11.2.0.2 in V$SQL_SHARED_CURSOR a ne column **USE_FEEDBACK_STATS** was introduced which contains value 'Y' if the cursor is not sharable due to significantly different cardinalities. The documentation for this column states: "A hard parse is forced so that the optimizer can re-optimize the query with improved cardinality estimates"
Otherwise the only trace that the automatic cardinality feedback was used to improve the performance is in V$SQL_PLAN.OTHER_XML column when it contains text "cardinality_feedback".
According to the optimizer group blog this feature should stop re-optimization after several attempts, however this might not be the case always. As all V$ tables are volatile the result of the following query can vary significantly, but I have seen values close to 40, what means that there very many re-optimization attempts.

```
  SELECT sql_id, COUNT (*)
    FROM v$sql_shared_cursor
   WHERE use_feedback_stats = 'Y'
GROUP BY sql_id
ORDER BY 2 DESC
```

Here is a part of sample output:

```
SQL_ID          COUNT(*) MAX(CHILD_NUMBER)
------------- ---------- -----------------
agfj9yqja74ka          10                10
91thqn3j4shfj           3                 4
gu7my5yzjm1ap           3                 2
68rvzaufbk6fr           3                 3
b7xhjjbmqdms4           3                 2
....
```

This means that there are several cases where this new feature somehow fails and generates many child cursors with the identical execution plan.

```
select plan_hash_value,count(*)
from v$sql
where  sql_id='agfj9yqja74ka'
group by plan_hash_value;


PLAN_HASH_VALUE   COUNT(*)
--------------- ----------
     3602887883         10
```

One can identify SQL statements which were tuned by applying feedback cardinality by running this statement:

```
select sql_id,child_number
from v$sql_plan
where other_xml is not null
and other_xml like '%cardinality_feedback%'
order by child_number desc;
```

```
SQL_ID        CHILD_NUMBER
------------- ------------
2bpp4r8ajsuz3           41
a9s5xz5v4qw95           36
2bpp4r8ajsuz3           35
97h27ay3zhhar           14
97h27ay3zhhar           12
91thqn3j4shfj            9
7ta3c5mugdj8d            4
231q3js3fbn0r            4
axm5005kdufpd            4
gc6k70xc7kfwj            4
193nyt3gxjgmm            3
7ta3c5mugdj8d            3
```

Here you can notice that the number of child cursors is pretty high. Of course this number constanly change as the memory used for the library cache is freed in order to make room for new SQL statements.

One can even disable automatic cardinality feedback tuning by setting „_optimizer_use_feedback“ parameter at system or session level. With opt_param hint (see next paragraph) one can disable it at SQL statement level and in the background by creating a SQL profile what is described in subsequent paragraph.

Extended statistics

One of the tough problems for the optimizer is when the rule of independency of columns is broken and columns depend on each other.

Let us create a table with two columns where both columns have the same values in all rows.

```
SQL> create table t2 as
     select trunc(rownum/300)-1 as c1, trunc(rownum/300)-1 as c2
     from dual connect by level <= 100000;

Table created.
```

Now let us see how accurate is CBO guess about cardinality – there are 300 rows for this combination.

```
SQL> explain plan for select * from t2 where c1=5 and c2=5;

Explained.

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

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1513984157

-----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |   876 | 22776 |    52   (2)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T2   |   876 | 22776 |    52   (2)| 00:00:01 |
-----------------------------------------------------------------------------
```

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

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------

   1 - filter("C1"=5 AND "C2"=5)

Note
   - dynamic sampling used for this statement (level=2)
```

So the guess is almost 3 times bigger than the actual number. Note that the dynamic sampling at level 2 was used to figure out the cardinality.
So if we gather statistics we should get better guess – but let us see what happens:

```
SQL> exec dbms_stats.gather_table_stats(
      ownname=>user,
      tabname=>'T2',
      method_opt=>'for all columns size skewonly');

PL/SQL procedure successfully completed.

SQL> select column_name,num_distinct,histogram
     from user_tab_col_statistics
     where table_name='T2' order by 1;
```

We have skew distribution and therefore we got height balanced histograms for both columns.

```
COLUMN_NAME                    NUM_DISTINCT HISTOGRAM
------------------------------ ------------ ---------------
C1                                      334 HEIGHT BALANCED
C2                                      334 HEIGHT BALANCED
```

Let us now check how this statistics helped the optimizer:

```
SQL> explain plan for select * from t2 where c1=5 and c2=5;

Explained.

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

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1513984157

-----------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |     8 |    52   (2)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T2   |     1 |     8 |    52   (2)| 00:00:01 |
-----------------------------------------------------------------------

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

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------

   1 - filter("C1"=5 AND "C2"=5)
```

Uuups, the cardinality estimate is 1 what is 300 times off. Now the optimizer follows the rule of column independency and that's why the cardinality is so misestimated.

Since 11g we have a new feature called extended statistics. Actually we build a histogram on a combination of several columns. Actually this is a virtual column.

First we have to create this extended statistics:

```
SQL> select dbms_stats.create_extended_stats(user,'T2','(c1,c2)') from dual;

DBMS_STATS.CREATE_EXTENDED_STATS(USER,'T2','(C1,C2)')
-----------------------------------------------------
SYS_STUF3GLKIOP5F4B0BTTCFTMX0W
```

Now we should re-gather the statistics and this extended statistics will be gathered as well:

```
SQL> exec dbms_stats.gather_table_stats(
          ownname=>user,tabname=>'T2',
          method_opt=>'for all columns size skewonly',
          estimate_percent=>null);

PL/SQL procedure successfully completed.

SQL> select column_name,num_distinct,histogram
     from user_tab_col_statistics
     where table_name='T2' order by 1;


COLUMN_NAME                    NUM_DISTINCT HISTOGRAM
------------------------------ ------------ ---------------
C1                                      334 HEIGHT BALANCED
C2                                      334 HEIGHT BALANCED
SYS_STUF3GLKIOP5F4B0BTTCFTMX0W          334 HEIGHT BALANCED


SQL> explain plan for select * from t2 where c1=5 and c2=5;

Explained.

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

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1513984157

-----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |   299 |  2392 |    52   (2)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T2   |   299 |  2392 |    52   (2)| 00:00:01 |
-----------------------------------------------------------------------------

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

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

   1 - filter("C1"=5 AND "C2"=5)
```

Now the cardinality estimate is almost 100% accurate.

This simple case is far from real life but a good example how the actual column dependency can mess up the cardinality estimates.

Tuning with SQL Profiles

SQL Profiles are normally prepared by Automatic Tuning Optimizer (ATO) by re-running the SQL statement full or by parts and preparing hints to improve the cardinality of row sources or even joins. However, not only hints for cardinality estimates **OPT_ESTIMATE** can be used in SQL Profiles but one can also change the optimizer environment. See the following case for enabling/disabling some parameters:

- opt_param('parallel_execution_enabled', 'false')
- opt_param('_b_tree_bitmap_plans', 'false')
- opt_param('optimizer_index_cost_adj',10)
- ….

Let us change the optimizer parameter OPTIMIZER_INDEX_COST_ADJ at the SQL statement level (for a particular statement) :

```
exec dbms_sqltune.import_sql_profile(-
                name => 'OICA',-
                category => 'DEFAULT',-
                sql_text => 'select * from t1 where n1 between 100 and 1000',-
                profile =>sqlprof_attr('opt_param(''optimizer_index_cost_adj'',10)'))
```

OPT_PARAM hint is used to set the optimizer environment. The parameters which can be set are defined in:

V$SYS_OPTIMIZER_ENV - Instance level
V$SES_OPTIMIZER_ENV - Session level
V$SQL_OPTIMIZER_ENV - Statement level

There are also some hidden parameters that can be set. Changing the optimizer environment forces optimizer to create a new child in the library cache (a hard parse is performed).

Sometimes one would like to disable all embedded optimizer hints which cause sub-optimal execution plan. This can be achieved either by SQL plan baseline or SQL Profile. Let's look the latter case where we simply use the hint IGNORE_OPTIM_EMBEDDED_HINTS:

```
exec dbms_sqltune.import_sql_profile(-
       name => 'ignore_all_hints',-
       category => 'DEFAULT',-
       sql_text => 'select /*+ full(joc1) */ count(*) from joc1',-
       profile => sqlprof_attr('IGNORE_OPTIM_EMBEDDED_HINTS'));
```

Unfortunately other optimizer hints like INDEX, USE_NL, etc.. are neglected by the optimzer if present in SQL profile.

Execution Cost of PL/SQL Functions/Packages Used In WHERE Clause

Back in 2005 I had my first presentation about so called “Extensible Optimizer” which also includes one very nice feature used to define the cost of execution of PL/SQL functions, packages, User defined index access and selectivity. Unfortunately this feature is overlooked although the SQL command ‘ASSOCIATE STATISTICS WITH’ which is used for defining the selectivity and cost is there since Oracle 9i.

Due to limited time I will show here how one can use this feature to define the cost of execution of a user defined function. If you define nothing oracle by default assigns a default cost and selectivity to a function which is used in the WHERE clause of the SQL statement.

Consider the following statement:

```
select * from t
where id between 1 and 1000
and my_func(t.id) = 0;
```

The CBO should as cost based optimizer know the execution cost of the function `my_func` and also the selectivity of this function – how many rows will pass the condition `my_func(t.id) = 0.` If this is not know then the CBO neither can estimate the cardinality of this statement result nor the cost of execution.

Here is the source of function my_func:

```
create or replace function my_func (p1 number)
return number is
begin
     return 0;
end;
/
```

It is obvious that this function really does nothing; it just returns 0. However, this is quite enough for our demonstration purpose. Let us run a simple query against table T and observe the execution plan.

```
SQL> explain plan for
select * from t
where c2 between 20 and 49
and my_func(c2)=0;


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

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 1601196873

------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |  2996 |   447K|  3068   (2)| 00:00:37 |
|*  1 |  TABLE ACCESS FULL| T    |  2996 |   447K|  3068   (2)| 00:00:37 |
------------------------------------------------------------------------------
```

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

   1 - filter("C2">=20 AND "MY_FUNC"("C2")=0 AND "C2"<=49)
```

There are some indexes present on this table but it is obvious that the FULL TABLE SCAN is the cheapest access method.

Now let us change the selectivity and cost of execution of function my_func. Oracle default values are: selectivity 1%, and cost is defaulted as 3000 cpu cycles and no I/O cost.
So making this function „heavier“ for execution should change the execution plan. Let us define selectivity 10% (this function will pass 10% of rows) and cost as 100000000 CPU cycles per execution and I/O cost as 1000 I/O operations per execution. The network cost is 0. Here is the command:

```
SQL> associate statistics with functions my_func
     default selectivity 10, default cost (100000000,1000,0);

Statistics associated.

SQL> explain plan for
     select * from t
     where c2 between 20 and 49
     and my_func(c2)=0;

Explained.

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

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 729576041

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             | 29961 |  4476K|    30M  (1)|100:22:27 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T           | 29961 |  4476K|    30M  (1)|100:22:27 |
|*  2 |   INDEX SKIP SCAN            | T_I_COMBINE | 29961 |       |    30M  (1)|100:22:19 |
-----------------------------------------------------------------------------------------------

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

   2 - access("C2">=20 AND "C2"<=49)
       filter("C2">=20 AND "MY_FUNC"("C2")=0 AND "C2"<=49)

15 rows selected.
```

It is obvious that the plan has change and INDEX SKIP SCAN method is used. Due toe heavily increased cost of single execution of my_func the overall cost for this statement has increased from 3068 to 30M.

It is important to remember the following **three rules**:

1.) The execution of functions which are very costly in terms of CPU usage or perform a lot of I/O should be postponed as much as possible in order to be executed on the smallest possible set of rows.
2.) More selective functions – thus which fill filter out more rows – will be executed first because they will filter out many rows in the very first steps of execution.
3.) If neither default selectivity nor default cost is defined then the functions will be executed in the order as they appear in the text of SQL statement.

One can disassociate statistics from with the command DISASSOCIATE STATISTICS FROM.

Finally how one can determine the cost of execution which should be used? In order to correctly determine CPU and I/O cost we can trace the execution of a function or package. Then we can use TKPROF output figures to determine the CPU time and number of I/O operations. When we figure out the required CPU time per execution, we can use the DBMS_ODCI.ESTIMATE_CPU_UNITS function to convert the CPU time per execution to the approximate number of CPU instructions. The returned number is defined as 1000s of instructions. In the following case we will calculate the CPU cost for a function that always executes in 0.002 seconds.

```
SQL> variable a number
SQL> begin :a := 1000*DBMS_ODCI.ESTIMATE_CPU_UNITS(0.002);end;
  2  /

PL/SQL procedure successfully completed.

SQL> print a;

         A
----------
312722,323
```

Contact address:

Jože Senegačnik
Dbprof d.o.o.
Smrjene 153
SI-1291 Skofljica, Slovenia

Phone: +386 41 72 44 61
Fax: +386 59 92 56 25
Email joze.senegacnik@dbprof.com
Internet: www.dbprof.com; joze-senegacnik.blogspot.com