THE TOM KYTE BLOG

FRIDAY, APRIL 06, 2007

When the explanation doesn't sound quite right...

I was asked recently

> *Under what conditions, autotrace & explain plan can not give the correct execution plan of a sql?*

The question came in email - not via asktom. But I found it of enough general interest to write about it here. As an aside, I rarely, if ever, answer questions emailed to me directly. It just isn't a scalable solution. This is my attempt to make this answer "scale"...

To start with the answer to this - we need to understand that autotrace is just a feature of SQL Plus that automates an explain plan for us - so, autotrace and explain plan are sort of synonymous in this regard. I'll be using Oracle 10g Release 2 in these examples and will be using autotrace or sql_trace=true and TKPROF - you can get the same results in 9i and later using EXPLAIN PLAN and DBMS_XPLAN.DISPLAY to see the results.

Explain Plan is in the here and now...

This is the first problem with explain plan - it is in the "here and now". It uses the current optimizer environment, the current set of statistics and so on. That means the explain plan you see in a tkprof could differ from the REAL PLAN used 5 minutes ago (when performance was 'bad'). For example:

```
ops$tkyte%ORA10GR2> create table t
  2  as
  3  select a.*, 1 id
  4    from all_objects a
  5   where rownum = 1;
Table created.

ops$tkyte%ORA10GR2> create index t_idx on t(id);
Index created.

ops$tkyte%ORA10GR2> alter session set sql_trace=true;
Session altered.

ops$tkyte%ORA10GR2> select id, object_name from t where id = 1;
        ID OBJECT_NAME
---------- ------------------------------
         1 ICOL$

ops$tkyte%ORA10GR2> insert into t select a.*, 1 from all_objects a;
50338 rows created.

ops$tkyte%ORA10GR2> select id, object_name from t where id = 1;
        ID OBJECT_NAME
---------- ------------------------------
         1 ICOL$
...
         1 WRH$_TABLESPACE_STAT
50339 rows selected.
```

```
ops$tkyte%ORA10GR2> alter session set sql_trace=false;
Session altered.
```

Now, running TKPROF:

```
$ tkprof /home/ora10gr2/admin/ora10gr2/udump/ora10gr2_ora_12173.trc ./tk.prf
aggregate=no sys=no explain=/
```

We will discover (in 10g, where dynamic sampling will kick in!) this conundrum:

```
select id, object_name from t where id = 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     3357      0.33       0.28          0       7490          0       50339
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     3359      0.33       0.28          0       7490          0       50339

Rows     Row Source Operation
-------  ---------------------------------------------------
  50339  TABLE ACCESS BY INDEX ROWID T (cr=7490 pr=0 pw=0 time=402830 us)
  50339   INDEX RANGE SCAN T_IDX (cr=3478 pr=0 pw=0 time=151058 us)(object id 70390)


Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
  50339   TABLE ACCESS (FULL) OF 'T' (TABLE)
```

Note how the "**Row Source Operation**" (what I like to call 'reality') differs from the "**Execution Plan**" (I'll call that the 'guess'). What happened here was that the row source operation is captured in the trace file at the time of execution - it reflects what REALLY took place as that query executed. We followed this sequence of operations:

1. loaded a single row into the table T
2. ran a query against T - that did a hard parse. At the time of the hard parse, Oracle 10g dynamically sampled the table and found it to be very small - and id=1 to be rather selective. Based on that - it said "let's range scan the index"
3. loaded a lot more data into the table. All with the same ID=1 value however.
4. ran the same query from step 2 - this was a soft parse and just reused the plan generated back when only one row existed in the table. As you can see however - that lead to inefficient execution. We read every row from that table via the index.
5. Executing TKPROF with explain= shows an entirely different plan. That is because explain plan *always* does a hard parse, it evaluated the query plan "in the here and now, as of this moment in time". It dynamically sampled the table again - found it to be large and ID=1 to not be selective. Explain plan shows us that if we hard parsed that query right now - it would full scan. However, all executions of that query in "real life" will index range scan as long as that plan is cached in the shared pool...

An important note for this example - the placement of the ALTER SESSION SET SQL_TRACE=TRUE is important. I needed to set it before running the query the first time. As an exercise - move it to just be before the second execution of the query and you'll find (from the tkprof) that the query is hard parsed the second time - and the row source operation in the tkprof will be a full scan. That is because the first time a query is executed with sql_trace=true (as opposed to the default of false), it will be *hard parsed - as of right now*.

Explain plan is blind to the bind

Explain plan does not "bind peek". This is pretty easy to observe:

```
  2  as
  3  select a.*, 1 id
  4    from all_objects a
  5   where rownum <= 5000;
Table created.

ops$tkyte%ORA10GR2> update t
  2     set id = 99
  3   where rownum = 1;
1 row updated.

ops$tkyte%ORA10GR2> create index t_idx on t(id);
Index created.

ops$tkyte%ORA10GR2> exec dbms_stats.gather_table_stats
( user, 'T', method_opt=> 'for all indexed columns size 254' );
PL/SQL procedure successfully completed.
```

So we have created some skewed data. If we say "where id=1", we would expect a full scan (index would be inefficient). If we say "where id = 99", we would expect an index range scan - as id=99 returns a single row. Using two queries that differ only in bind names (which is sufficient to prevent cursor sharing - these are two DIFFERENT queries to Oracle!), we'll execute a query with a bind set to the value 99 and then another with a bind set to 1.

```
ops$tkyte%ORA10GR2> variable x_is_99_first number
ops$tkyte%ORA10GR2> variable x_is_1_first number
ops$tkyte%ORA10GR2> exec :x_is_99_first := 99; :x_is_1_first := 1;
PL/SQL procedure successfully completed.

ops$tkyte%ORA10GR2> alter session set sql_trace=true;
Session altered.

ops$tkyte%ORA10GR2> select id, object_name from t where id = :x_is_99_first;

        ID OBJECT_NAME
---------- ------------------------------
        99 ICOL$

ops$tkyte%ORA10GR2> select id, object_name from t where id = :x_is_1_first;

        ID OBJECT_NAME
---------- ------------------------------
         1 I_USER1
....
```

```
         1 USER_SCHEDULER_PROGRAM_ARGS
4999 rows selected.
```

Now we'll just flip flop the values and re-execute the queries. Note that they will *soft parse, just reuse the existing plans generated from above*.

```
ops$tkyte%ORA10GR2> exec :x_is_99_first := 1; :x_is_1_first := 99;
PL/SQL procedure successfully completed.

ops$tkyte%ORA10GR2> select id, object_name from t where id = :x_is_99_first;

        ID OBJECT_NAME
---------- ------------------------------
         1 I_USER1
...
         1 USER_SCHEDULER_PROGRAM_ARGS
4999 rows selected.

ops$tkyte%ORA10GR2> select id, object_name from t where id = :x_is_1_first;

        ID OBJECT_NAME
---------- ------------------------------
        99 ICOL$

ops$tkyte%ORA10GR2> alter session set sql_trace=false;
Session altered.
```

Reviewing the TKPROF report first for the "x is 99 first" query we see:

```
select id, object_name from t where id = :x_is_99_first

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        2      0.00       0.00          0          3          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          0          3          0           1

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 120  (OPS$TKYTE)

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS BY INDEX ROWID T (cr=3 pr=0 pw=0 time=59 us)
      1   INDEX RANGE SCAN T_IDX (cr=2 pr=0 pw=0 time=32 us)(object id 70394)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      1   TABLE ACCESS   MODE: ANALYZED (FULL) OF 'T' (TABLE)
```

```
********************************************************************************
select id, object_name from t where id = :x_is_99_first

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      0.02       0.02          0        739          0        4999
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      337      0.02       0.02          0        739          0        4999

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 120  (OPS$TKYTE)

Rows     Row Source Operation
-------  ---------------------------------------------------
   4999  TABLE ACCESS BY INDEX ROWID T (cr=739 pr=0 pw=0 time=50042 us)
   4999   INDEX RANGE SCAN T_IDX (cr=344 pr=0 pw=0 time=30018 us)(object id 70394)


Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
   4999   TABLE ACCESS   MODE: ANALYZED (FULL) OF 'T' (TABLE)
```

So, the "real plan" used is an index range scan - both times. But, explain plan - which cannot, does not bind peek - will say "full scan". The reason? explain plan is optimizing "select * from t where id = ?" - and it says "5,000 rows, 2 values of id, id is not selective, full scan". But the optimizer is optimizing the query "select * from t where id = 99" - because it peeked at the bind the first time! The soft parse won't peek (else it would be a hard parse!) and just reused the existing plan - the inefficient range scan to read every row out.

On the other hand, looking at the "x is 1 first" query:

```
select id, object_name from t where id = :x_is_1_first

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      0.01       0.01          0        398          0        4999
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      337      0.01       0.01          0        398          0        4999

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 120  (OPS$TKYTE)

Rows     Row Source Operation
```

```
-------  --------------------------------------------------
   4999  TABLE ACCESS FULL T (cr=398 pr=0 pw=0 time=15094 us)

Rows     Execution Plan
-------  --------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
   4999   TABLE ACCESS   MODE: ANALYZED (FULL) OF 'T' (TABLE)
********************************************************************************
select id, object_name from t where id = :x_is_1_first

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        2      0.00       0.00          0         67          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          0         67          0           1

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 120  (OPS$TKYTE)

Rows     Row Source Operation
-------  --------------------------------------------------
      1  TABLE ACCESS FULL T (cr=67 pr=0 pw=0 time=82 us)

Rows     Execution Plan
-------  --------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      1   TABLE ACCESS   MODE: ANALYZED (FULL) OF 'T' (TABLE)
```

Explain plan ***appears*** to have gotten it right - but only by accident. It is just a coincidence that the plans "match" - they were arrived at by very different thought processes. The optimizer optimized 'where id=1' and said "about 5,000 rows, about 4,999 will be returned, full scan". The explain plan optimized "where id=?" and said "about 5,000 rows in the table, two values for ID, about 50% of the table will be returned, full scan".

So, that example shows explain plan "getting it wrong" because it is blind to the bind - and shows the effect of bind variable peeking (which you can read more about on asktom using the link above...)

Explain plan doesn't see your datatype...

The last bit about explain plan I'll look at is the fact that explain plan doesn't see your bind datatype. It presumes ***all binds are varchar2's*** regardless of how the developer is binding. Consider:

```
ops$tkyte%ORA10GR2> create table t
  2  ( x varchar2(10) primary key,
  3    y date
  4  );
Table created.

ops$tkyte%ORA10GR2> insert into t values ( 1, sysdate );
1 row created.
```

```
ops$tkyte%ORA10GR2> variable x number
ops$tkyte%ORA10GR2> exec :x := 1
PL/SQL procedure successfully completed.

ops$tkyte%ORA10GR2> alter session set sql_trace=true;
Session altered.

ops$tkyte%ORA10GR2> select * from t where x = :x;

X          Y
---------- ---------
1          06-APR-07

ops$tkyte%ORA10GR2> alter session set sql_trace=false;
Session altered.
```

So, we have a table with a varchar2 datatype for the primary key - but we only stuff numbers in there. End users and developers know it is always a number and then presume the type is a number (makes sense) - but someone used the wrong datatype (just in case maybe....). When we look at the TKPROF we'll see the explain plan mismatch:

```
select * from t where x = :x

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          1          0           0
Fetch        2      0.00       0.00          0          7          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          0          8          0           1

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 120  (OPS$TKYTE)

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS FULL T (cr=7 pr=0 pw=0 time=76 us)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      1   TABLE ACCESS (BY INDEX ROWID) OF 'T' (TABLE)
      0    INDEX (UNIQUE SCAN) OF 'SYS_C0013586' (INDEX (UNIQUE))
```

Explain plan - the 'execution plan' shows an index unique scan, but reality (the row source operation) shows we full scanned. DBMS_XPLAN (autotrace in 10gr2 uses these new package introduced in 9ir2, you can use it directly if you like) shows us ***why*** we are full scanning:

```
ops$tkyte%ORA10GR2> select * from t where x = to_number(:x);

Execution Plan
----------------------------------------------------------
Plan hash value: 1601196873

--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    16 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T    |     1 |    16 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------

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

   1 - filter(TO_NUMBER("X")=TO_NUMBER(:X))

Note
-----
   - dynamic sampling used for this statement
```

So, when I told explain plan "hey, we have a NUMBER here" using to_number(), we can see what happened. In the predicate information, we see that when you compare a number to a string, Oracle will TO_NUMBER(the string). That of course makes using the index on the string not possible!

So, this example shows two things. Firstly, that explain plan assumes varchar2 (so use to_number or literals!! and to_date to get the right type conveyed to explain plan). Secondly, that implicit conversions are ***evil and should always be avoided***.

POSTED BY THOMAS KYTE AT 9:36 AM 

POST A COMMENT

23 COMMENTS:

Simon said....

Really interesting

FRI APR 06, 10:43:00 AM EDT

Alen Oblak said....

Sort of an "upgrade" of what you already written in your book "Effective Oracle By Design", nice.

Is it possible to force Oracle somehow to peek at the bind variables of a SQL without the cost of doing a hard parse? I guess not, since the peeking could lead to a different explain plan for each and every execution.

It would maybe be cool to do like this:
1. First time the SQL is executed, hard parse it and execute it. When peeking at the binds, store the predictions.
2. Second time the SQl is executed, Oracle finds it in the shared pool, peeks at the current binds, and comapre the predictions to the original predictions from step 1.
a) If they are the same as before, we dont need to do the steps "optimize" and "row source generate", kinda

like soft parse.
b) If the predictions are different, hard parse it, and store the predicions.
So in reality, we could end with a statement and a set of (predictions - execution plan) for it in the shared pool. Wouldnt that be great? Every time the same statement would be executed, we are sure the Oracle will use the best execution plan possible with that set of bind variables.

Sure, this could lead to more hard parses, but it is the tradeoff to efficient statement processing.

FRI APR 06, 12:05:00 PM EDT

Thomas Kyte said....

Is it possible to force Oracle somehow to peek at the bind variables of a SQL without the cost of doing a hard parse?

cursor_sharing=similar with literals will do that actually - at the huge expense of a soft parse.

FRI APR 06, 12:25:00 PM EDT

Anonymous said....

Once again, thanks for another detailed
explanation Tom.

Given the limitations of tkprof/explain plan, under what situations would you recommend using them instead of directly querying vsql, vsql_plan, v$sql_bind_capture when debugging performance issues?

FRI APR 06, 03:27:00 PM EDT

Thomas Kyte said....

tkprof doesn't have this limitation - just don't use explain=!!!

tkprof shows you reality (row source operation), v$ tables show you reality.

As long as you understand how explain plan works - it can show you reality too...

FRI APR 06, 04:47:00 PM EDT

Gudmundur Josepsson said....

Tom,

Despite the three examples that you have given where the explain plan command is “wrong” would you still agree that the explain plan command is useful because in all the examples you give it really was the designer/developer/DBA that was making a mistake?

1. The DBA makes the mistake of not recalculating statistics on a table that contains 50,000 times more rows than it did before.
2. The designer makes the mistake of using a bind variable where a literal variable might have been more appropriate.
3. The schema designer makes the mistake of using data types incorrectly.

Implementing proper database maintenance or making modifications to the code so it is “right” would have caused the explain plan command to be right every time. Might it be appropriate to say that the fact that there a difference between the explain plan output and the actual plan is indicative of an error on the designer’s/developer’s/DBA’s behalf?

FRI APR 06, 06:36:00 PM EDT

Alen Oblak said....

cursor_sharing=similar with literals will do that actually - at the huge expense of a soft parse.

But within PL/SQL, literals in SQL statements are transformed in bind variables, right? So this parameter doesn't help us at all.

SAT APR 07, 11:23:00 AM EDT

Anonymous said....

Alen, unless I am greatly mistaken :
PL/SQL *variables*, when used in static SQL, become bind variables;
on the other hand, *literals* (meaning stuff within single quotes) remain literals.

SAT APR 07, 04:00:00 PM EDT

Alen Oblak said....

PL/SQL variables, when used in static SQL, become bind variables.
Sure, my fault.

Well the point is, if we have a sql like this:
select name
from person
where name like :name

We have an index on "name" column
and we execute it with :name = "TOM", then the explain plan uses that index and gets one or two results,assumig we have a "normal" distribution of names in the table. Now we execute it again with :name = '%' and we get the same execution plan, reading the whole index and the whole table. The goal in this example is to use a different execution plan and to simply scan the entire table.

What I was asking for: is there a way to peek at that :name variable and choose a suitable execution plan each time the statement is executed? Like with cursor_sharing parameter, but on the bind variables, not only the literals.

MON APR 09, 06:42:00 AM EDT

Thomas Kyte said....

is there a way to peek at that :name variable and choose a suitable execution plan each time the statement is executed

No, the only way to achieve that is to use literals plus cursor_sharing=force and cause a soft parse to occur each and every execution.

MON APR 09, 08:26:00 AM EDT

Anonymous said....

Tom,

I understand that setting cursor_sharing=force will replace the literals with system bind variables and avoid hard parsing, at a cost of soft parsing.

How about applications where bind variables are already being used? Will setting cursor_sharing to force in such environments will cause additional soft parsing? In other words, can we use the cursor_sharing=force as a 'fall back just in case' without a downside?

MON APR 09, 11:49:00 AM EDT

Thomas Kyte said....

can we use the cursor_sharing=force as a 'fall back just in case' without a downside?

I would not suggest that. There are times when you **purposely do not bind -** and that would kill that.

cursor_sharing, if you set it, should be considered a "temporary bug fix, workaround for THIS APPLICATION ONLY until they fix it" sort of thing. (in my opinion)

MON APR 09, 12:05:00 PM EDT

Arun Mathur said....

LOVE using tkprof and sql_trace. Would love it if there was a way somehow to:

1) Identify a session.
2) Alter that session interactively via my current session ie:
alter session set timed_statistics=true;
alter session set events '10046 trace name context forever, level 12';
3) tail -f the trace file and let the fun begin

Regards,
Arun

WED APR 11, 04:10:00 PM EDT

Thomas Kyte said....

Would love it if there was a way somehow to:

well, timed stats should just always be on - regardless

and in 9i and before, dbms_system.set_ev can set the event (search asktom for that)

in 10g, dbms_monitor can do that

WED APR 11, 04:48:00 PM EDT

Arun Mathur said....

Excellent. Thanks!

WED APR 11, 04:51:00 PM EDT

Anonymous said....

Tom,

This may not be totally related to the topic in hand, but in the first part of your demo "Explain Plan is in the here and now..." I have a question:

You create a new table, index it, then insert values and when selecting 50339 rows, it does a consistent (query) read. Shouldnt that be current read because the SELECT statement right after the INSERT of the 50338 rows?

SAT APR 14, 08:54:00 PM EDT

Abhijit said....

Tom,
Thanks for the details about Explain Plan. I have soem doubts, please clarify.

When exactly the plan flushed from the Library Cache? After it flushed how the cache next time when the query called does it again do the parsing and generate the execution plan? What is the way to use the

efficient plan whenever a query is called for execution?

Thanks,
--Abhijit Mallick

THU MAY 10, 02:24:00 AM EDT

Moorthy said....

Tom,

Great post as usual. Thanks for sharing your insights.

Metalink Doc ID 462913.1 says:

The following situations are needed to ensure Row Source Operation details are reported in the raw trace and corresponding tkprof output;

1) Database parameter STATISTICS_LEVEL should be set to ALL. Verify using;

SQL> select value from v$parameter where name like 'statistics_level'

(an alternative parameter is setting _rowsource_execution_statistics = TRUE)

2) Row source operation will be shown only if the session from which performance is traced is closed properly. If the session remains open/active cursors can be left open, and hence the row source operation information cannot be fully recorded. It is therefore recommended for users to complete operations and exit properly from applications to ensure this is completed.

My question is:

While tuning in development and load test environments, I ask developers to close their sessions (reset the connection pool in weblogic environment) after the required work is traced to get the row source operation in traces to see what "really" happened.

If we experience a performance problem in production (I think every DBA must have faced this), it may not be feasible to close the connections or reset the connection pools.

Is there a way to get the row source operation in production environments without closing the sessions or resetting the connection pools?

When the sessions were not closed, sometimes I used to get row source operation and some times not. Then, came across that metalink article. Since then, if possible, I ask developers to close their sessions.

Thank you very much for your contributions to Oracle community.

Thanks,
Moorthy.

FRI MAR 06, 06:04:00 PM EST

Thomas Kyte said....

@Moorthy

in 11g, point #2 (about having to close the cursor) is no longer true

prior to that, it takes closing the cursor (for real, not a cached open cursor like jdbc and plsql can do).

FRI MAR 06, 06:24:00 PM EST

Anonymous said....

This may not be relevant to this section. In our environment, performance for couple of insert statements has deteriorated after upgrading to 11g. Statements are largely CPU bound which was not the case before. It is an IOT table with a nested data type. Do you see any reason for this behavior.

THU AUG 27, 10:35:00 AM EDT

Thomas Kyte said....

@anonymous

given the extremely high level of detail you have given

No one could say anything. Not at all.

THU AUG 27, 10:46:00 AM EDT

Anonymous said....

A pretty old article, but this still pops up close to the top when googling for the difference between rowsource ops and explain plan!

My question is: when we call DBMS_XPLAN.DISPLAY_AWR on an sql_id in the cache - rather than one in the plan_table - is this subject to the same limitations as 'explain plan for'? i.e. is it also showing an approximation rather than 'reality'? My question ir oriented at >=10g R2.

Thaks for the great books Tom.

THU MAR 21, 06:58:00 AM EDT

Thomas Kyte said....

when displaying from AWR/ASH or from v$sql tables - you are seeing what actually happened, those are not explain plans, those are the real plans used at that time.

THU MAR 21, 07:00:00 AM EDT

POST A COMMENT

<< Home