<u>Using V$SQL_PLAN to get accurate execution plans</u>

The cost-based optimiser is good (and continues to get better with each Oracle release) but is not yet foolproof. Every system the author has ever worked on has had a small number of key SQL statements that the cost-based optimiser somehow fails to optimise as expected. Much can be done with collecting specific statistics or tweaking specific session settings to improve the cost-based optimiser's choice of execution. It is, however, inevitable that at some time the database administrator is going to have to work out the access path being used. With this knowledge some way of speeding the query up (or reducing resource usage) can be determined. This article focuses on using the dynamic view V$SQL_PLAN (which first became available in Oracle9i) and its derivatives to get accurate access path information as an aid in the task of query tuning.

There are various tuning tools and utilities that the Oracle database administrator can employ to work out the execution plan being used as the first step to tuning the query. Some common examples are:

- The `EXPLAIN PLAN` command
- `AUTOTRACE` from within SQL*Plus (which itself automatically invokes an `EXPLAIN PLAN` behind the scenes)
- Turning on session trace to capture statistics for later analysis via the TKPROF utility
- Various Oracle supplied or 3rd party tools such as Oracle Enterprise Manager, Grid Control, SQL Developer and TOAD

`EXPLAIN PLAN`, and also SQL*Plus `AUTOTRACE` as it uses `EXPLAIN PLAN` to generate part of its output, are not necessarily accurate. It is important to realise that `EXPLAIN PLAN` works out the execution path by evaluating the query supplied to it in the context of the session/environment in place at the time the `EXPLAIN PLAN` command itself is issued. There are a number of important factors that can affect whether or not `EXPLAIN PLAN` comes up with the correct plan for the query being tuned. Some crucial ones are:

- system and object statistics as currently held are used
- the currently available indexes are used
- database and session settings as at the time the `EXPLAIN PLAN` is run are used
- the optimiser mode (`ALL_ROWS, FIRST_ROWS, CHOOSE` etc)
- `EXPLAIN PLAN` does not do bind variable peeking

Even the slightest change in any of these can result in a different execution path being shown as opposed to the one that was actually used. `EXPLAIN PLAN` and `AUTOTRACE` are very useful when writing or re-writing queries to give an indication of the execution path that can be achieved, but should not be relied upon when trying to determine what actually went on sometime in the past, not matter how recently.

In Oracle9i the `EXPLAIN PLAN` from `AUTOTRACE` lacks certain elements of detail such as partition elimination. This has been corrected in Oracle10g.

Getting a TKPROF from a session that has SQLTrace turned on is the ultimate tuning tool, as it produces a lot more information than just the execution path. The information it generates is also going to be an accurate statement of what actually happened, as it is recorded immediately after each significant event ends. However, you can only trace a running session; that is to say; if trace was not turned on for your problem query session (or database wide) you have no way of retrospectively getting the trace information you need. It is by far the most comprehensive and accurate way to tune a system (at which point I highly recommend the excellent book "Optimizing Oracle Performance" by Cary Millsap and Jeff Holt, which covers system-wide tuning in great detail), but this article is really focussed on getting some help when tuning errant SQL statements. If you have the luxury of being able to request the re-running of the troublesome query, then setting SQLTrace for the relevant session will be most beneficial. In reality, that luxury is not often allowed us. To compound things there are situations where the explain plan does not get written to the trace file. The most common reason is that the cursor does not get closed within the session – the explain plan is written to the only when the cursor is closed. So if you terminate the session, you will not get the complete trace details. What I offer here is a quicker and easier method to see a SQL statement's execution path.

Oracle tools and 3rd party tools cost money. Check very carefully the terms of usage of those wonderful GUI tools Oracle supply such as Grid Control, as not every feature comes for free. In addition, some tools merely provide a pretty front end and nicely formatted output to standard `EXPLAIN PLAN,` so will suffer the same drawbacks as described earlier. Of the tools I had access to at the time of writing this article, it seems that TOAD 8.6.1 still used the `EXPLAIN PLAN` method, as did SQL Developer (formerly Raptor) but Grid Control 10gR2 utilised V$SQL_PLAN.

It's about time V$SQL_PLAN was formally introduced to you. Oracle9i and above make the execution plans of statements still in the statement cache available, via this single V$ view. Presumably the information was available in prior versions as the optimiser would have to have it available for re-use. Some exceedingly clever readers may know of how to extract this information for Oracle8i from X$ or memory structures, but that data was never made common knowledge (if, indeed, it is possible for your standard database administrator to get at it).

What makes V$SQL_PLAN so useful is the fact that it combines both the execution path with the SQL still in the statement cache. So you have a chance of seeing the exact execution plan used on a statement after it has run (so long as it is still in the cache). The data is in the same format as you get from an `EXPLAIN PLAN` (but you can be sure it is accurate, which you cannot with `EXPLAIN PLAN`s for the reasons stated above) and so there are no new syntax or concepts to learn. The data is also available via simple SQL, with no need to start a SQLTrace session, locate the .TRC file, format with TKPROF and read through etc. For these reasons I have found it a very useful addition to my DBA tool-kit. With it you can very quickly see the execution path of a troublesome statement.

To make things even better, even if the statement is still running you can see the optimiser plan it is using. So if you are a production DBA and get a call saying a certain part of your business critical application is running slow, you could be able to locate a slow running piece of code and see, within a few seconds (if it is a single SQL statement that is the cause), the execution plan it is using. From there you can immediately start your tuning process.

The V$SQL_PLAN dynamic view has the following columns available for versions 9i and 10g, for comparison the corresponding PLAN_TABLE columns are shown.

V$SQL_PLAN (9i)	PLAN_TABLE (9i)	V$SQL_PLAN (10g)	PLAN_TABLE (10g)
-	STATEMENT_ID	-	STATEMENT_ID
-	-	-	PLAN_ID
-	-	SQL_ID	-
-	TIMESTAMP	TIMESTAMP	TIMESTAMP
-	REMARKS	REMARKS	REMARKS
OPERATION	OPERATION	OPERATION	OPERATION
ADDRESS	-	ADDRESS	-
HASH_VALUE	-	HASH_VALUE	-
-	-	PLAN_HASH_VALUE	-
-	-	CHILD_ADDRESS	-
CHILD_NUMBER	-	CHILD_NUMBER	-
OPTIONS	OPTIONS	OPTIONS	OPTIONS
OBJECT_NODE	OBJECT_NODE	OBJECT_NODE	OBJECT_NODE
OBJECT#	OBJECT_INSTANCE[1]	OBJECT#	OBJECT_INSTANCE[1]
-	OBJECT_TYPE	OBJECT_TYPE	OBJECT_TYPE
OBJECT_OWNER	OBJECT_OWNER	OBJECT_OWNER	OBJECT_OWNER
OBJECT_NAME	OBJECT_NAME	OBJECT_NAME	OBJECT_NAME
-	-	OBJECT_ALIAS	OBJECT_ALIAS
OPTIMIZER	OPTIMIZER	OPTIMIZER	OPTIMIZER
ID	ID	ID	ID
PARENT_ID	PARENT_ID	PARENT_ID	PARENT_ID
DEPTH	-	DEPTH	DEPTH
POSITION	POSITION	POSITION	POSITION
SEARCH_COLUMNS	SEARCH_COLUMNS	SEARCH_COLUMNS	SEARCH_COLUMNS
COST	COST	COST	COST
CARDINALITY	CARDINALITY	CARDINALITY	CARDINALITY
BYTES	BYTES	BYTES	BYTES
OTHER_TAG	OTHER_TAG	OTHER_TAG	OTHER_TAG
PARTITION_START	PARTITION_START	PARTITION_START	PARTITION_START
PARTITION_STOP	PARTITION_STOP	PARTITION_STOP	PARTITION_STOP
PARTITION_ID	PARTITION_ID	PARTITION_ID	PARTITION_ID
OTHER	OTHER	OTHER	OTHER
DISTRIBUTION	DISTRIBUTION	DISTRIBUTION	DISTRIBUTION
CPU_COST	CPU_COST	CPU_COST	CPU_COST
IO_COST	IO_COST	IO_COST	IO_COST
TEMP_SPACE	TEMP_SPACE	TEMP_SPACE	TEMP_SPACE
ACCESS_PREDICATES	ACCESS_PREDICATES	ACCESS_PREDICATES	ACCESS_PREDICATES
FILTER_PREDICATES	FILTER_PREDICATES	FILTER_PREDICATES	FILTER_PREDICATES
-	-	PROJECTION	PROJECTION
-	-	TIME	TIME

[1] I make the assumption that V$SQL_PLAN.OBJECT# maps to PLAN_TABLE.OBJECT_INSTANCE but it is not actually needed to use DBMS_XPLAN.

```
       -                    -          QBLOCK_NAME        QBLOCK_NAME
       -                    -          OTJER_XML          OTJER_XML
```

If you compare this with the layout of the PLAN_TABLE table (as used by `EXPLAIN_PLAN`), you can see that the data collected is almost identical, with the exception of different identifier columns (naturally), the addition of columns for remarks and timestamp on PLAN_TABLE. The only difference on Oracle10g is that V$SQL_PLAN has depth columns whereas PLAN_TABLE does not.

Figures 1 through 6 show the main elements of the Oracle9i version of a SQL*Plus script I use to make quick and easy use of V$SQL_PLAN. At this point I must state that the script was developed over time, with each incremental development inspired by something I read, heard about or learnt of from sources such as the UK OUG, on-line discussion forums (eg Oracle-l) and web sites (eg askTom, Oracle FAQ). Mine was the perspiration; the inspiration comes from a number of other sources.

The first SQL statement in the script (figure 1) is used to track down the statement to be tuned. It does a basic wild-card search on some user entered search string to locate SQL still in the statement cache (using V$SQL). It joins this to V$SQL_PLAN using the statement address and hash to ensure there is a plan available and also to get the child number (the same SQL statement can have different execution plans for a number of reasons, each different plan is given a unique child number). From that, it displays the SQL statement (note that theV$SQL column SQL_TEXT only holds the first 1000 characters of the statement, so not only do you only see that many characters but the search is limited to that portion of the SQL statement – I've been caught out by that a number of times). It also displays the statement address, hash_value and child_number which are required by subsequent queries in the script.

Figure 1
```
SELECT /*+IGNOREME*/
       DISTINCT ses.address||'/'||ses.hash_value
      ,sql.sql_text stmt
      ,ses.child_number
FROM    v$sql_plan ses
       ,v$sql sql
WHERE  ses.address=sl.address
AND    ses.hash_value=sql.hash_value
AND    UPPER(sql.sql_text)
         LIKE UPPER('%&&sqlstm%')
AND    sql.sql_text
         NOT LIKE '%/*+IGNOREME*/%'
ORDER BY 1;
```

The DBA has then to decide which SQL statement she/he wants to see the execution plan for, I usually just cut and past the address/hash_plan fields and make a mental note of the child. From these inputs the statement is uniquely identified and the script then shows the complete SQL (NB In Oracle10g the complete SQL is made available directly on V$SQL via the SQL_FULLTEXT column so there is no need to mess about with V$SQLTEXT_WITH_NEWLINES as is needed with Oracle9i). This is shown in figure 2.

Figure 2

```
SELECT  sql.sql_text stmt
FROM    v$sqltext_with_newlines sql
WHERE   address='&&addr'
AND     hash_value='&&hash'
ORDER BY sql.piece;
```

There are now 2 choices in how to get the execution plan displayed in a format that the DBA can read. The first, and possibly more traditional, route is to use a query very similar to the query Oracle supply with the XPLAN (`EXPLAIN PLAN`) scripts as shown in figure 3.

Figure 3

```
SELECT '| Operation                          | PHV/Object Name      |  Rows | Bytes|   Cost |'
as "Optimizer Plan:"
FROM dual
UNION ALL
SELECT * FROM (SELECT
       rpad('|'||substr(lpad(' ',1*(depth-1))||operation||
            decode(options, null,'',' '||options), 1, 32), 33, ' ')||'|'||
       rpad(decode(id, 0, '----- '||to_char(hash_value)||' ['||to_char(child_number)||']
-----'
                     , substr(decode(substr(object_name, 1, 7), 'SYS_LE_', null,
object_name)
                       ||' ',1, 20)), 21, ' ')||'|'||
       lpad(decode(cardinality,null,'  ',
               decode(sign(cardinality-1000), -1, cardinality||' ',
               decode(sign(cardinality-1000000), -1, trunc(cardinality/1000)||'K',
               decode(sign(cardinality-1000000000), -1, trunc(cardinality/1000000)||'M',
                      trunc(cardinality/1000000000)||'G')))), 7, ' ') || '|' ||
       lpad(decode(bytes,null,' ',
               decode(sign(bytes-1024), -1, bytes||' ',
               decode(sign(bytes-1048576), -1, trunc(bytes/1024)||'K',
               decode(sign(bytes-1073741824), -1, trunc(bytes/1048576)||'M',
                      trunc(bytes/1073741824)||'G')))), 6, ' ') || '|' ||
       lpad(decode(cost,null,' ',
               decode(sign(cost-10000000), -1, cost||' ',
               decode(sign(cost-1000000000), -1, trunc(cost/1000000)||'M',
                      trunc(cost/1000000000)||'G'))), 8, ' ') || '|' as "Explain plan"
        FROM    v$sql_plan
        WHERE   address='&&addr'
        AND     hash_value='&&hash'
        AND     child_number=&&child
        ORDER BY hash_value,child_number,id);
```

NB: The source for this statement originates from Oracle metalink note 260942.1.

The second method uses the DBMS_XPLAN package coupled with the fact that you can manually insert into the PLAN_TABLE used by `EXPLAIN PLAN`. DBMS_XPLAN contains a function that will return a neatly formatted report for the last statement in PLAN_TABLE. So the data from V$SQL_PLAN is fed into PLAN_TABLE and DBMS_XPLAN.DISPLAY used to output a fully formatted execution plan as shown in figures 4 and 5.

Figure 4

```
INSERT INTO plan_table
SELECT   DISTINCT address,sysdate,'REMARKS',operation,options,object_node
        ,object_owner,object_name, 0,'object_type',optimizer,search_columns
        ,id,parent_id,position,cost,cardinality,bytes,other_tag
        ,partition_start,partition_stop,partition_id,other,distribution
        ,cpu_cost,io_cost,temp_space
FROM    v$sql_plan
WHERE   address='&&addr'
AND     hash_value='&&hash'
AND     child_number=&&child;
```

Figure 5

```
SELECT plan_table_output
FROM TABLE(dbms_xplan.display
 ('plan_table',null,'serial'));
```

NB: At this point it is worth adding that it makes sense to create the PLAN_TABLE as a global temporary table rather than permanent table (unless, of course, you do wish to keep your `EXPLAIN PLAN` results for longer than the life of the session you are using for your tuning).

One final thing I like to do is to get some indication of how much work the statement has actually done – ie: to get a feel for how much impact it is having on the database itself (as one bad statement does not necessarily have a significant impact). In figure 6, V$SQL is again used for this to show how many times the statement has been called, how much CPU has been used, how much IO was spent and how much data was fetched. All the values are averaged to give an indication of the cost of each per statement execution (and it is only an indication, no more). Rather than show details for the child number used previously, it shows the summary details for each child statement – just as a final check to make sure the child being tuned is the one having an impact.

Figure 6

```
SELECT executions e
      ,cpu_time c
      ,DECODE(executions,0,0,ROUND(cpu_time/executions,2)) pe
      ,buffer_gets   bg
      ,DECODE(executions,0,0,ROUND(buffer_gets/executions)) peb
      ,rows_processed rp
      ,DECODE(executions,0,0,ROUND(rows_processed/executions)) per
      ,disk_reads     dr
      ,DECODE(executions,0,0,ROUND(disk_reads/executions)) ped
      ,address||'/'||hash_value||'['||child_number||']' d
FROM   v$sql
WHERE  address='&&addr'
AND    hash_value='&&hash';
```

You can use V$SQL_PLAN in other ways too. My favourite way is to use it to locate what statements use a given index (recognising that this will only be for statements still in the statement cache). This is useful to quickly check that the statement is actually using an index that has been added for a specific performance reason (note that it only shows the index usage in SELECT statements, not the usage that corresponds to index maintenance when rows are added, updated or deleted from the owning table). Index monitoring could also have been used, but using V$SQL_PLAN is much easier and less intrusive (no DDL needs to be run), so long as you are aware of the limitations.

So what if your statement is no longer in the cache, is V$SQL_PLAN of no use to you? Well, not exactly. Various Oracle-supplied utilities take snapshots of the dynamic view and preserve the data into permanent tables. Both statspack and AWR do this. My personal preference is statspack as it is free (i.e.: no additional licensing worries) and tried and trusted. Even though rumour had it that statspack was to be deprecated in 10g, a lot of work must have gone into it between 9i and 10g as it appears to support the majority of the new 10g features. Provided you snap at level 6 or above and frequently enough to capture most of you SQL plans, statspack will populate a table called

STATS$SQLPLAN with data out of V$SQL_PLAN. Then it is just a case of hunting down the plan's hash_value and using one of the techniques described earlier. Figure 7 shows how to insert into PLAN_TABLE from statspack recorded data.

Figure 7

```
INSERT INTO plan_table
SELECT DISTINCT 1,SYSDATE,'REMARKS',operation,options
       ,object_node,object_owner,object_name,0,'object_type'
       ,optimizer,search_columns,id,parent_id,position, cost
       ,cardinality,bytes,other_tag,partition_start
       ,partition_stop,partition_id,other,distribution,cpu_cost
       ,io_cost,temp_space,access_predicates,filter_predicates
FROM  stats$sql_plan
WHERE plan_hash_value=&&hash_value;
```

Statspack, however, does has some limitations over its newer rival AWR as it does not capture the most resource intensive statements for the snap interval – instead it captures the current top N statements as seen in V$SQL at snap time based on your statspack thresholds. The difference is subtle but important. Statspack is reporting on what is in the library cache at the time the snap is taken – the statements in the library cache did not necessarily run since the last snap was taken. The SQL recorded could have been in cache for some time, Oracle has not had any cause to flush them out even though they may not have been run for a while. The problem can hit you from the other direction too; there could have been statements that executed within the snapshot interval that you need to know but statspack did not record them as either they did not meet the threshold settings or they were flushed out of memory just before the snap was taken.

By comparison the top SQL captured by AWR does accurately reflect SQL for the interval.

Statspack also fails to record data for access predicates and filter_predicates, at least in the Oracle9i version of the code.

If the statement is no longer in the cache and you do not have trustworthy statspack or AWR data recording it, then the only route open to you is to get the statement re-run. For V$SQL_PLAN all you need do is to have the SQL started and then once it is actually executing you can kill the session it is running in – you do not have to wait for the query to return data or suffer the prolonged affects of the query running on your system. As soon as the optimisation phase is completed, the data you require will be in V$SQL_PLAN and you can see the execution path via any of the means I detailed earlier in this article.

There is one final trick you can use which I recently read about on the wonderful ORACLE-L email list. This does not require the SQL statement to even run! However, it is not guaranteed to always work and you must make sure the statement you use with it is exactly the same as the one you are tuning. The trick uses a PL/SQL cursor and simply open it then closes it (see figure 8), which is enough in most cases to require Oracle to generate an execution plan. For more information on this trick go to the “Oracle (tm) Users' Co-Operative FAQ” at http://www.jlcomp.demon.co.uk/faq/how_to_explain.html.

Figure 8

```
DECLARE
  b NUMBER:=0;
  CURSOR c1
  IS SELECT /*+unique string*/ *
     FROM   t
     WHERE col1=b1;
BEGIN
  OPEN c1;
  CLOSE c1;
END;
```

Finally, it is worth noting that things change a little again with Oracle10g. Additional data has been added to V$SQL and V$SQL_PLAN and new functions been added to DBMS_XPLAN. This greatly simplifies the use of V$SQL_PLAN when used in SQL statement tuning. A 10g version of the script is available from my website. If you download this you will see the 10g version is altered to take account of the following new features:

1. V$SQL includes the SQL_FULLTEXT column (as a CLOB) so searching is no longer limited to the first 1000 lines and the full SQL statement can be displayed directly without having to use V$SQLTEXT_WITH_NEWLINES
2. V$SQL and V$SQL_PLAN both contain a new column, SQL_ID, which makes identifing and keeping track of the errant SQL statement easier
3. DBMS_XPLAN has a new function, DISPLAY_CURSOR, which will return a formatted `EXPLAIN PLAN` output directly from V$SQL_PLAN, simply by passing in the SQL_ID and child number (note that there is also a new function, DISPLAY_AWR, which can be used with data stored in the AWR repository)
4. The new view V$SQL_PLAN_STATISTICS_ALL can be used to show useful metrics on the performance of the SQL statement

I feel I must reiterate the warning I gave previously about licensing options. When using Oracle10g with features such as AWR you must be careful that you are not breaking the terms of your license agreement; some features can only be validly used if you have a specific option pack (for AWR that is the diagnostic pack). Much confusion exists over what is available as standard and what is an extra cost option. AWR is listed as an extra cost option, yet is installed by default and even creates itself a regular scheduled task to collect data as part of the default database install. In particular, use of the DBMS_PLAN.DISPLAY_AWR function may require the Database Diagnostic Pack licensing, as Oracle state that this additional license is required when ANY access to DBA_HIST_* package is used. The author understands that in order for Oracle to use DISPLAY_AWR it would need to use the DBA_HIST_SQLPLAN view.

<u>Disclaimer</u>: You must always check the hints, tips and scripts presented in this paper before using them and always try them out on a test database before running against a live system. Whilst every care has been taken to ensure the scripts function properly and are totally unobtrusive and benign (when used

Tim Onions is an independent database consultant with over 15 years of experience with Oracle databases. He specializes in the architecture, application design and database design of high performance systems, as well as tuning and optimisation techniques together with technical management. Tim can be contacted via Tim.Onions@TOdC.co.uk. The scripts described in this article are available for download from the website http://www.TOdC.co.uk.