A Monograph on ASH

Practical Use of Oracle Active Session History

Prepared By David Kurtz, Go-Faster Consultancy Ltd.

A Monograph on ASH
Version 1.1
Monday 3 November 2014

(E-mail: david.kurtz@go-faster.co.uk, telephone +44-7771-760660)

File: Practical_ASH.docx, 3 November 2014

Contents

Introduction

This document started as preparation for a presentation

Agenda

- Briefly, what is ASH and what does it collect (see page 4)
  - Recent/Historical Activity
- OEM and ASH Report (see page 5)
- Compare and Contrast with SQL Trace (see page 10)`.
- Application Instrumentation (see page 12).
  - PeopleSoft specific example of adding your own instrumentation.
- Using SQL to Analyse
  - Top SQL
  - Monitoring progress of process in read time (see page 23).
  - Lock Analysis (see page 40)
    - Blocking Session Not Active.
  - Changing Exection Plans (see page 58)
  - Source of I/O (see page 46)
  - Temporary Tablespace Usage (see page 66)
  - Limitations (see page 67)
    - Cannot Obtain SQL (space 67)
    - Error Messages (see page 75)

A Very Brief Overiew of Active Session History

Active Session History (ASH) was introduced in Oracle 10g. It samples the activity of each active[1] database session every second. The data is held in a buffer in memory in the database. The design goal is to keep about an hour (your mileage will vary). If a session is not active it will not be sampled. The in-memory buffer is exposed via a view called *v$active_session_history*.

You could sort of simulate some of ASH by taking a snapshot of *v$session* for every session, but the overhead would be prohibitive. ASH is built into the Oracle kernel, so its overhead is minimal.

When an AWR snapshot is taken, 1 row in 10 from the ASH buffer is copied down into the AWR repository. It can also be flushed to disk between snapshots when the buffer reaches 66% full, so there is no missed data.The data is stored in WRH$_ACTIVE_SESSION_HISTORY and it is exposed via DBA_HIST_ACTIVE_SESS_HISTORY.

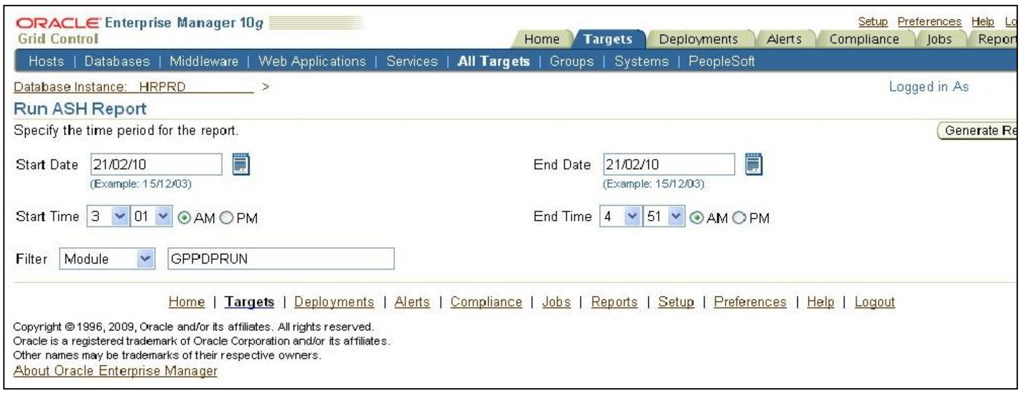
ASH is enabled by default, but before you rush off to use it, be aware that it is a licenced feature. It is part of the Diagnostic Pack, so you have to pay for it. I don’t like that either, but that’s how it is.

[1] I want to emphasise that if the session is not active it will not be sampled. You can actually set a parameter *_ash_enable_all* = TRUE to force all sessions, including idle sessions, to be sampled.

But as Doug Burns points out in his blog posting (http://oracledoug.com/serendipity/index.php?/archives/1395-ASH-and-the-psychology-of-Hidden-Parameters.html), these are undocumented, unsupported parameters, and they are set this way for a reason – you have been warned.

ASH in Oracle Enterprise Manager

Of course, OEM provides a way to run ASH reports, and here you see I have picked a particular time window, and I have specified a module name – in this case the main payroll calculation process.


ORACLE Enterprise Manager 10g
Grid Control
Setup Preferences Help Lo
Home Targets Deployments Alerts Compliance Jobs Repor
Hosts | Databases | Middleware | Web Applications | Services | All Targets | Groups | Systems | PeopleSoft
Logged in As
Generate Re

Database Instance: HRPRD >

Run ASH Report

Specify the time period for the report.

Start Date 21/02/10 (Example: 15/12/03)

End Date 21/02/10 (Example: 15/12/03)

Start Time 3 01 ● AM ○ PM

End Time 4 51 ● AM ○ PM

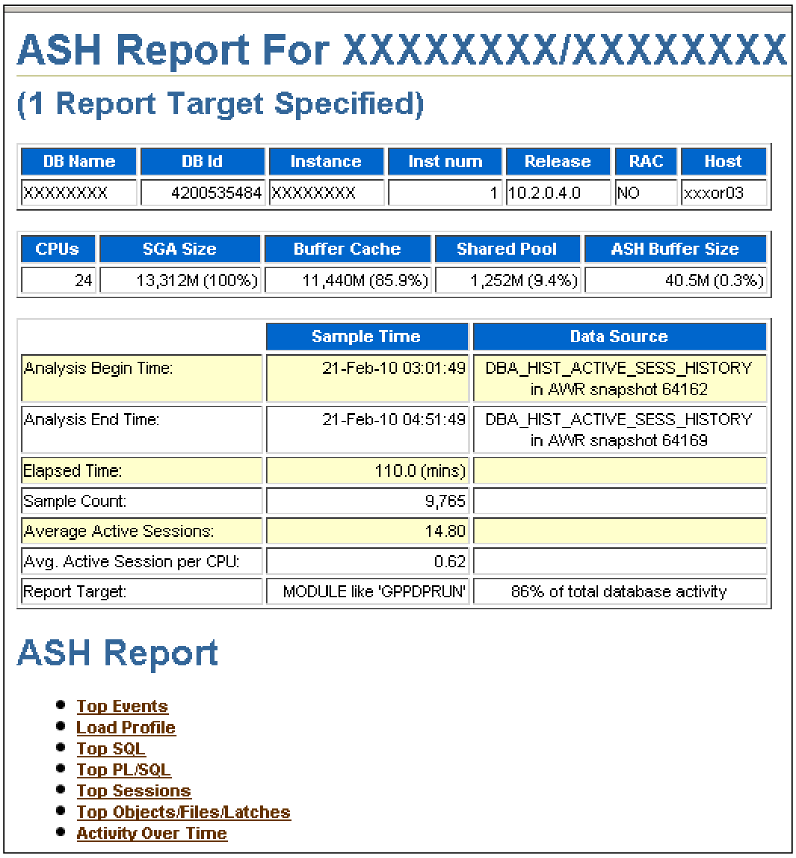
Filter Module GPPDPRUN

Home | Targets | Deployments | Alerts | Compliance | Jobs | Reports | Setup | Preferences | Help | Logout

Copyright © 1996, 2009, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.
About Oracle Enterprise Manager

And this is great. The report is easy to produce, and it tells you lots of things. Which SQL statements are consuming the most time, which objects have the most I

You can see in this example I picked a module that was responsible for 86% of the total, and there were an average of 14.8 active sessions (I know there were 32 concurrent processes).



ASH Report For XXXXXXXX/XXXXXXXX

(1 Report Target Specified)

DB Name	DB Id	Instance	Inst num	Release	RAC	Host
XXXXXXXX	4200535484	XXXXXXXX	1	10.2.0.4.0	NO	xxxor03

CPUs	SGA Size	Buffer Cache	Shared Pool	ASH Buffer Size
24	13,312M (100%)	11,440M (85.9%)	1,252M (9.4%)	40.5M (0.3%)

	Sample Time	Data Source
Analysis Begin Time:	21-Feb-10 03:01:49	DBA_HIST_ACTIVE_SESS_HISTORY in AWR snapshot 64162
Analysis End Time:	21-Feb-10 04:51:49	DBA_HIST_ACTIVE_SESS_HISTORY in AWR snapshot 64169
Elapsed Time:	110.0 (mins)	
Sample Count:	9,765	
Average Active Sessions:	14.80	
Avg. Active Session per CPU:	0.62	
Report Target:	MODULE like 'GPPDPRUN'	86% of total database activity

ASH Report

- Top Events
- Load Profile
- Top SQL
- Top PL/SQL
- Top Sessions
- Top Objects/Files/Latches
- Activity Over Time

But, you don't get execution plans, and for that you will need to dig deeper yourself, and learn to use the DBMS_XPLAN package.

What data does ASH retain?

Most of the columns on v$active_session_history are taken directly from column of the same name on v$session, some have different name, and there is some additional information that is not available elsewhere.

Column on v$active_session_history	**Correspondence to v$session**
SAMPLE_ID	ID of ASH Sample
SAMPLE_TIME	Time of ASH Sample
IS_AWR_SAMPLE	*New in 11gR2*
SESSION_ID	V$SESSION.SID
SESSION_SERIAL#	V$SESSION.SERIAL#
USER_ID	V$SESSION.USER#
SQL_ID	√
IS_SQL_ID_CURRENT	*New in 11gR2*
SQL_CHILD_NUMBER	√
FORCE_MATCHING_SIGNATURE	not on V$SESSION
SQL_OPCODE	√
TOP_LEVEL_SQL_ID	*New in 11gR1*
TOP_LEVEL_SQL_OPCODE	*New in 11gR1*
SQL_PLAN_HASH_VALUE	not on V$SESSION
SQL_PLAN_LINE_ID	*New in 11gR1*
SQL_PLAN_OPERATION	*New in 11gR1*
SQL_PLAN_OPTIONS	*New in 11gR1*
SQL_EXEC_ID	√ *New in 11gR1*
SQL_EXEC_START	√ *New in 11gR1*
PLSQL_ENTRY_OBJECT_ID	√
PLSQL_ENTRY_SUBPROGRAM_ID	√
PLSQL_OBJECT_ID	√
PLSQL_SUBPROGRAM_ID	√
SERVICE_HASH	V$ACTIVE_SERVICES.NAME_HASH

SESSION_TYPE	V$SESSION.TYPE
SESSION_STATE	Waiting/On-CPU
QC_SESSION_ID	Parallel query co-ordinator
QC_INSTANCE_ID	√
QC_SESSION_SERIAL#	*New in 11gR1*
BLOCKING_SESSION	√
BLOCKING_SESSION_STATUS	VALID – blocking session within the same instance GLOBAL – blocking session in another instance.
BLOCKING_SESSION_SERIAL#	V$SESSION.SERIAL# of blocking session
EVENT	√
EVENT_ID	From V$EVENT_NAME
EVENT#	√
SEQ#	√
P1TEXT	√
P1	√
P2TEXT	√
P2	√
P3TEXT	√
P3	√
WAIT_CLASS	√
WAIT_CLASS_ID	√
WAIT_TIME	√
TIME_WAITED	√
XID	Not on V$SESSION
REMOTE_INSTANCE#	*New in 11gR1*
CURRENT_OBJ#	V$SESSION.ROW_WAIT_OBJ#
CURRENT_FILE#	V$SESSION.ROW_WAIT_FILE#

CURRENT_BLOCK#	V$SESSION.ROW_WAIT_BLOCK#
CURRENT_ROW#	√ *New in 11gR1*
CONSUMER_GROUP_ID	*New in 11gR1*
PROGRAM	√
MODULE	√
ACTION	√
CLIENT_ID	V$SESSION.CLIENT_IDENTIFIER
FLAGS	Undocumented
IN_CONNECTION_MGMT	*New in 11gR1*
IN_PARSE	*New in 11gR1*
IN_HARD_PARSE	*New in 11gR1*
IN_SQL_EXECUTION	*New in 11gR1*
IN_PLSQL_EXECUTION	*New in 11gR1*
IN_PLSQL_RPC	*New in 11gR1*
IN_PLSQL_COMPILATION	*New in 11gR1*
IN_JAVA_EXECUTION	*New in 11gR1*
IN_BIND	*New in 11gR1*
IN_CLOSE_CURSOR	*New in 11gR1*
IN_SEQUENCE_LOAD	*New in 11gR2*
CAPTURE_OVERHEAD	*New in 11gR2*
REPLAY_OVERHEAD	*New in 11gR2*
IS_CAPTURED	*New in 11gR2*
IS_REPLAYED	*New in 11gR2*
MACHINE	√ *New in 11gR2*
PORT	√ *New in 11gR2*
ECID	√ *New in 11gR2*
TM_DELTA_TIME	*New in 11gR2*
TM_DELTA_CPU_TIME	*New in 11gR2*

TM_DELTA_DB_TIME	*New in 11gR2*
DELTA_TIME	*New in 11gR2*
DELTA_READ_IO_REQUESTS	*New in 11gR2*
DELTA_WRITE_IO_REQUESTS	*New in 11gR2*
DELTA_READ_IO_BYTES	*New in 11gR2*
DELTA_WRITE_IO_BYTES	*New in 11gR2*
DELTA_INTERCONNECT_BYTES	*New in 11gR2*
PGA_ALLOCATED	*New in 11gR2*
TEMP_SPACE_ALLOCATED	*New in 11gR2*

Comparison with SQL Trace

ASH and SQL*Trace are not the same thing, but both are valuable tools for finding out about where processes spend time.

SQL*Trace (or event 10046 as we used to call it) has been my weapon of choice for solving performance issues for a very long time, and it is extremely effective, and there is still a place for it.

There are difficulties with using SQL trace, especially in a production environment.

- Firstly, it does have a run time overhead. You could afford to trace a single process, but you certainly couldn't trace the entire database.

- You have to work with trace in a reactive way. You will probably not already be tracing a process when you experience a performance problem, so you need to run the process again and reproduce the poor performance with trace.

- Trace will tell you if a session is blocked waiting on a lock. However, it will not tell you who is blocking you. ASH will do this (although there are limitations).

- A trace file records everything that happens in a session, whereas ASH data samples the session every seconds. Short-lived events will be missed, so the data has to be handled statistically (see page 14).

- There are problems with both approaches if you have the kind of application where you have lots of different SQL statements because the application uses literal values rather than bind variables (and cursor sharing is EXACT).

- Oracle's TKPROF trace file profiler cannot aggregate these statements, but I have found another called ORASRP (www.oracledba.ru/orasrp) that can. With ASH, you will see different SQL_IDs, but it can be effective to group statements with the same execution plan.

- You may have trouble finding the SQL text in the SGA (or via the DBMS_XPLAN package) because it has already been aged out of the library cache. You may have similar problems with historical ASH data because the statement had been aged out when the AWR snapshot was taken.

- A trace file, with STATISTICS_LEVEL set to ALL, will give you timings for each operation in the execution plan. So, you can see where in the execution plan the time was spent. ASH will only tell you how long the whole statement takes to execute, and how long was spent on which wait event.

> Through the rest of this document you will see SQL_IDs. However, in a SQL trace the statements are identified by hash_value. Those hash values do not show up if you profile your trace file with tkprof, but they do if you use OraSRP. SQL_ID is just a fancy representation of hash value, so you can convert from a SQL_ID to a hash_value. Oracle supply function DBMS_UTILITY.SQLID_TO_SQLHASH(), but as the comment on the blog says Tanel's script is much cooler[2].

You can't get the whole of the SQL_ID back from the hash values (because it is trimmed off), but you can get the last 5 or 6 characters it help you find or match SQL statements[3]

[2] See Tanel Poder's blog: http://blog.tanelpoder.com/2009/02/22/sql_id-is-just-a-fancy-representation-of-hash-value/

[3] And I could never have written this without seeing Tanel's code!

```
CREATE OR REPLACE FUNCTION h2i (p_hash_value NUMBER) RETURN VARCHAR2 IS
 l_output VARCHAR2(10) := '';
BEGIN
 FOR i IN (
  SELECT substr('0123456789abcdfghjkmnpqrstuvwxyz',1+floor(mod(p_hash_value/(POWER(32,LEVEL-1)),32)),1) sqlidchar
  FROM dual CONNECT BY LEVEL <= LN(p_hash_value)/LN(32) ORDER BY LEVEL DESC
 ) LOOP
  l_output := l_output || i.sqlidchar;
 END LOOP;
 RETURN l_output;
END;
/
```

Application Instrumentation

Oracle has provided a package called DBMS_APPLICATION_INFO since at least Oracle 8. This allows you to set two attributes; MODULE and ACTION for a session. That value then appears in *v$session*, and can be very useful to help you identify what database sessions relate to what part of an application. These values are then also captured by ASH.

I cannot over-emphasise the importantance of this instrumentation when analysing performance issues. Without sensible values in these columns all you have is the program name. You will probably struggle to identify ASH data for the sessions which are of interest.

These values are not set by default. Instead DBAs are dependent on developers to include them in their code. For example, Oracle E-Business Suite has built this into the application.

PeopleSoft Specific Instrumentation

However, other application vendors have not. For example, PeopleSoft (prior to PeopleTools 8.50) only write the name of the executable into the module[4]. This is really no help at all because the executable name is held in another column.

For batch processes, I have developed a trigger which is fired by batch processes as they start and which sets a meaningful process name, and puts the unique process instance number into the action.

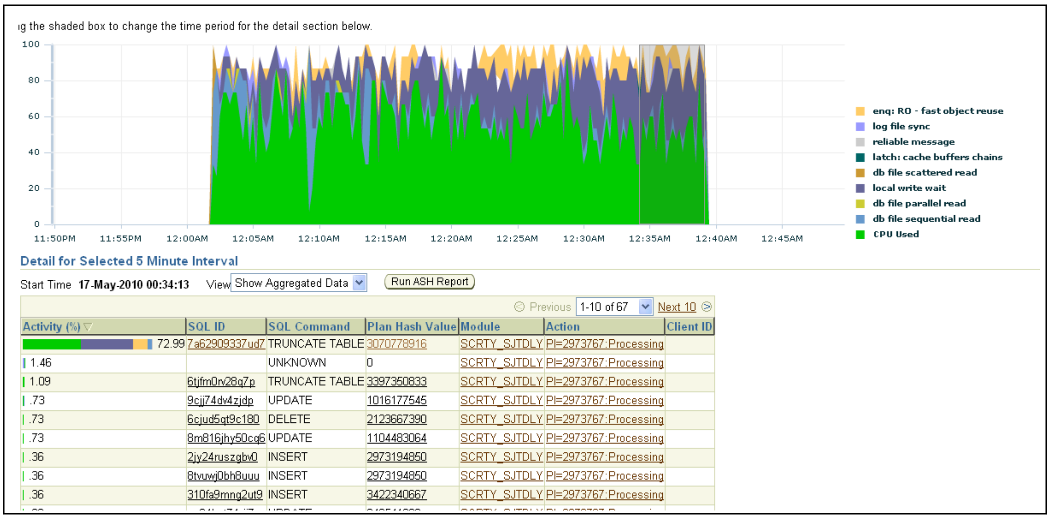
```
CREATE OR REPLACE TRIGGER sysadm.psftapi_store_prcsinstance
BEFORE UPDATE OF runstatus ON sysadm.psprcsrqst FOR EACH ROW
WHEN (      (new.runstatus IN('3','7','8','9','10') OR old.runstatus IN('7','8'))
AND new.prcstype != 'PSJob')
BEGIN
…
  psftapi.set_action(p_prcsinstance=>:new.prcsinstance
                    ,p_runstatus=>:new.runstatus
                    ,p_prcsname=>:new.prcsname);
…
EXCEPTION WHEN OTHERS THEN NULL; --exception deliberately coded to suppress all exceptions
END;
/
```

From PeopleTools 8.50, Oracle added instrumentation for the on-line part of the application.

In PeopleTools 8.52, further instrumentation was added for Application Engine. The Application Engine program name, section name, step name and step type are written to the ACTION. The PeopleSoft Operator ID is stored in CLIENT_ID

[4]

The results of this instrumentation are visible in Enterprise Manager



Detail for Selected 5 Minute Interval

Start Time **17-May-2010 00:34:13**

Activity (%) ▽	SQL ID	SQL Command	Plan Hash Value	Module	Action	Client ID
72.99	7a62909337ud7	TRUNCATE TABLE	3070778916	SCRTY_SJTDLY	PI=2973767:Processing	
1.46		UNKNOWN	0	SCRTY_SJTDLY	PI=2973767:Processing	
1.09	6tjfm0rv28q7p	TRUNCATE TABLE	3397350833	SCRTY_SJTDLY	PI=2973767:Processing	
.73	9cjj74dv4zjdp	UPDATE	1016177545	SCRTY_SJTDLY	PI=2973767:Processing	
.73	6cjud5qt9c180	DELETE	2123667390	SCRTY_SJTDLY	PI=2973767:Processing	
.73	8m816jhy50cq6	UPDATE	1104483064	SCRTY_SJTDLY	PI=2973767:Processing	
.36	2jy24ruszgbv0	INSERT	2973194850	SCRTY_SJTDLY	PI=2973767:Processing	
.36	8tvuwj0bh8uuu	INSERT	2973194850	SCRTY_SJTDLY	PI=2973767:Processing	
.36	310fa9mng2ut9	INSERT	3422340667	SCRTY_SJTDLY	PI=2973767:Processing	

Later, you will see the value of this instrumentation as I use it to join a combination of data in the application about batch processes with the ASH repository to identify where a given process spent time.

Using SQL to Analyse ASH Data

Statistical Analysis Approach

ASH data is a sample and so must be handled statistically. If something happens that lasts 10 seconds, then it will be sampled about 10 times.

However, not everything that happens is captured. If something happens that last less than a second, but it happens very frequently, some of them will be captured. For example, if something happens which lasts for 1/10th of a second, but happens 100 times then you would expect to capture it about 10 times. In all, the 100 occurences lasted 10 times. So by counting each ASH row as worth 1 seconds of wait time you come out at the right answer. This is what I mean by taking a statistical approach.

So, if you are looking at a current or recent process you the raw ASH data, and the query that you have to construct when working with is something along these lines

```
SELECT      …
,           SUM(1) ash_secs
FROM        v$active_session_history
WHERE       …
GROUP BY …
```

And if you are going further back in time then you have to work with the historical data, only 1 in 10 rows are kept, so now each row is worth 10 seconds

```
SELECT      …
,           SUM(10) ash_secs
FROM        dba_hist_active_sess_history
WHERE       …
GROUP BY …
```

And of course, you won't see recent data in this view until there is an AWR snapshot for the ASH buffer fills to 2/3 and flushes.

ASH History is exposed by the view DBA_HIST_ACTIVE_SESSION_HISTORY. It is stored in the table SYS. WRH$_ACTIVE_SESSION_HISTORY which is range partitioned on DBID and SNAP_ID. To make the SQL work efficiently you need to specify the snap ID, for that I use dba_hist_snapshotS to identify the range of snapshots that you want to use, and the partitions first so that you eliminate unwanted partitions. You may need the LEADING hint to force Oracle to start with the snapshot view, and then the USE_NL hint to force it to work through each snapshot, which will guarantee a single partition access. Otherwise your queries could run for ever!

```
SELECT     /*+LEADING(x) USE_NL(h)*/ …
,          SUM(10) ash_secs
FROM       dba_hist_active_sess_history h
,          dba_hist_snapshot x
WHERE      x.snap_id = h.snap_id
AND        x.dbid = h.dbid
AND        x.instance_number = h.instance_number
AND        x.end_interval_time >= …
AND        x.begin_interval_time <= …
AND        …
GROUP BY …
```

Objectives

Ask yourself what you are trying to find out.

- Are you interested in a single database session, or a group of sessions, or the whole database?
- All ASH Data –v- One Wait Event
- Time Window

PeopleSoft Specific ASH Queries

To get the most out of ASH you need to know how to relate database session to processes. That starts with using DBMS_APPLICAITON_INFO to register the process name and process instance of batch processes on the session (see page 12). But there is more.

Batch Processes

The start and end time of a batch process is recorded on the process request table, and you can use that to identify the snapshots, and thence the active session history.

```
SELECT /*+LEADING(r x h) USE_NL(h)[5]*/
            r.prcsinstance
,           h.sql_id
--,         h.sql_child_number
,           h.sql_plan_hash_value
,           (CAST(r.enddttm AS DATE)-CAST(r.begindttm AS DATE))*86400 exec_secs
,           SUM(10) ash_secs
FROM        dba_hist_snapshot x
```

[5] Specify a hint to ensure good performance. Start with the process request table, then go to the snapshots, finally go to the ASH data and look it up with a nested loop join.

```
,           dba_hist_active_sess_history h
,           sysadm.psprcsrqst r[6]
WHERE       x.end_interval_time >= r.begindttm[7]
AND         X.begin_interval_time <= r.enddttm
AND         h.sample_time BETWEEN r.begindttm AND r.enddttm[8]
AND         h.snap_id = x.snap_id
AND         h.dbid = x.dbid
AND         h.instance_number = x.instance_number
AND         h.module = r.prcsname[9]
AND         h.action LIKE 'PI='||r.prcsinstance||'%'[10]
AND         r.prcsinstance = 1956338[11]
GROUP BY r.prcsinstance, r.prcsname, r.begindttm, r.enddttm, h.sql_id, h.sql_plan_hash_value
ORDER BY 1
/
```

Application Engine from PeopleTools 8.52

From PeopleTools 8.52 there is additional instrumentation of the session in Application Engine processes.

- Module is now set to string composed of PSAE.<name of scheduled Application Engine program>.<session ID number>. The Application Engine name is as it appears in Process Monitor. The session ID number is the operation system process ID of the client process. It is recorded in PSPRCSQUE.SESSIONIDNUM.

- Action is set to the concatenation of the Application Engine program name, section name, step name and step type. The string can be truncated if it is too long.

Consequently a slightly different SQL query is required to analyse ASH data for these processes[12]. This construction is only applicable to Application Engine from PeopleTools 8.52, and will not work on Application Engine in earlier versions of PeopleTools, the construction in the previous section is still applicable to other process types in PeopleTools 8.52.

```
From (
 select /*+leading(r q x h)[13] use_nl(h)*/
  r.prcsinstance
```

[6] This table described the process

[7] Identify the AWR snapshots that coincide with the period that the process was running

[8] Filter ASH data to exactly the period that the process was running.

[9] Filter ASH data by Module which is the name of the process on the process request table

[10] Filter ASH data by Action which includes the process instance number

[11] Uniquely identify process

[12] However, most of the examples in this document were written against PeopleTools 8.49.

[13] Note that the LEADING hint has been changed to include PSPRCSQUE as the second table visited.

```
, h.action, h.sql_id
, h.sql_plan_hash_value
, (CAST(enddttm AS DATE)-CAST(begindttm AS DATE))*86400 exec_secs
, sum(10) ash_secs
from DBA_HIST_SNAPSHOT x
, DBA_HIST_ACTIVE_SESS_HISTORY h
, sysadm.psprcsrqst r
, sysadm.psprcsque q[14]
WHERE r.prcsinstance = q.prcsinstance
and r.prcsinstance = 10622259
and r.prcsname = 'TL_TIMEADMIN'
AND X.END_INTERVAL_TIME >= r.begindttm
And x.begin_interval_time <= r.enddttm
and  h.SNAP_id = X.SNAP_id
and h.dbid = x.dbid
and h.instance_number = x.instance_number
and h.module like 'PSAE.'||r.prcsname||'.'||q.sessionidnum[15]
and h.sample_time BETWEEN r.begindttm AND r.enddttm
group by r.prcsinstance, r.prcsname, r.begindttm, r.enddttm
, h.action
, h.sql_id
, h.sql_plan_hash_value
) where ash_secs>exec_secs/100
order by ash_secs desc
/
```

Now it is possible to include the step reference from the Action in the ASH profile. Of course it is likely, as in this example, that one step produces different SQL IDs on different executions either due to dynamically generated SQL, or different bind variables values in different executions being resolved to different litteral values by Application Engine.

```
                                                              SQL Plan       Exec        ASH
PRCSINSTANCE ACTION                           SQL_ID        Hash Value       Secs       Secs
------------ -------------------------------- ------------- ---------- ---------- ----------
    10622259 TL_TIMEADMIN.END.STATS2.S        636f1jtg06rjk 2915643330       5901        320
    10622259 TL_TIMEADMIN.END.STATS2.S        cbrj18vrfb2qj  821036523       5901        320
    10622259 FO_TL_OVR_RT.MAIN.Step03.S       4rgvvjm5jt1gn 2867360147       5901        300
    10622259 TL_TRPROFILE.TRPROFIL.End_Effd.  gbwayc9ac1jxu 3317352158       5901        300
    10622259 FO_TL_OVR_RT.MAIN.Step05.S       2zyz4zr0js2j8 1281985392       5901        250
    10622259 TL_TA001100.TA001120.Step09A.S   bcrxp3xps3466  537875261       5901        120
    10622259 TL_TA000900.TA000960.Step130.S   9j67wxxk6gut5  334959449       5901         90
    10622259 TL_TA001000.TA001000.Step02.S    anmqwa0sn18yh 2593881656       5901         80
    10622259 FO_TL_OVR_RT.MAIN.Step01.S       f4ybwvc0pzkvj 2562206473       5901         70
                                                                                  ----------
```

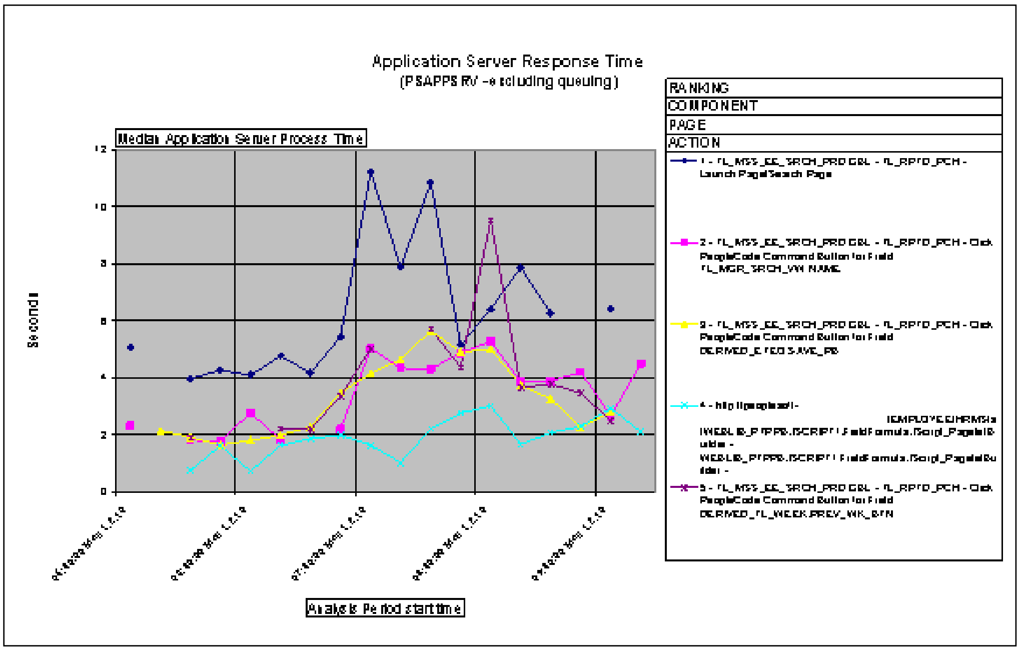
[14] PSPRCSQUE is also needed to obtain the session ID number and this can be joined to PSPRCSRQST by PRCSINSTANCE.

[15] The combination of process name, session ID number and sample time is not guaranteed to be unique. It is possible that two instances of the same program with the same session ID number could run on different Process Schedulers on different servers concurrently, although this is not likely.

```
sum                                                       1850
```

On-Line Activity

I have used the PeopleSoft Performance Monitor (PPM) to find a period in time when the system exhibits degraded performance.



With on-line activity it is not possible to add module and action instrumentation. At the moment the program name is copied to module, and that is no advantage at all because I already have program in the ASH data

Enhancement Request: *PeopleSoft added instrumentation for Performance Monitor, the context information they there use there for a PIA transaction could also be set in DBMS_APPLICATION_INFO. Combine Component and Page to Module, and set Action as Action*

So, all I can do is query ASH data relating to PSAPPSRV programs. If you have separte PSQRYSRV processes, you can analyse that separately too.

```
SELECT /*+LEADING(x h) USE_NL(h)*/
            h.sql_id
,           h.sql_plan_hash_value
,           SUM(10) ash_secs
FROM        dba_hist_snapshot x
,           dba_hist_active_sess_history h
WHERE       x.end_interval_time   >= TO_DATE('201002010730','yyyymmddhh24mi')
AND         x.begin_interval_time <= TO_DATE('201002010830','yyyymmddhh24mi')
AND         h.sample_time BETWEEN    TO_DATE('201002010730','yyyymmddhh24mi')
                      AND              TO_DATE('201002010830','yyyymmddhh24mi')
AND         h.snap_id = x.snap_id
AND         h.dbid = x.dbid
AND         h.instance_number = x.instance_number
AND         h.module like 'PSAPPSRV%'
GROUP BY h.sql_id, h.sql_plan_hash_value
ORDER BY ash_secs DESC
/
```

At least most of the SQL in the on-line application uses bind variables (except for certain bits of dynamically generated code), so it does aggregate properly in the ASH data.

```
                SQL Plan
SQL_ID        Hash Value   ASH_SECS
------------- ---------- ----------
7hvaxp65s70qw 1051046890       1360
fdukyw87n6prc  313261966        760
8d56bz2qxwy6j 2399544943        720
876mfmryd8yv7  156976114        710
bphpwrud1q83t 3575267335        690
…
```

XML Report

If you make use of XML reporting, usually to deliverer PeopleSoft Queries then you find that they are all run through an Application Engine program called PSXPQRYRPT. You can use the PS_CDM_FILE_LIST table to work out the Report ID that was requested, and you can look at the report definition (PSXPRPTDEFN) to find the underlying query.

This query just reports run time for a report called XGF_WK_LATE. We haven't added any ASH data yet.

```
SELECT r.prcsinstance, r.begindttm, d.report_defn_id, d.ds_type, d.ds_id
,      (CAST(r.enddttm AS DATE)-CAST(r.begindttm AS DATE))*86400 secs
FROM   sysadm.psprcsrqst r
,      sysadm.ps_cdm_file_list f
,      sysadm.psxprptdefn d
,      sysadm.psxpdatasrc s
WHERE  r.prcsname = 'PSXPQRYRPT'
and    r.prcsinstance = f.prcsinstance
and NOT f.cdm_file_type IN('AET','TRC','LOG')
and    d.report_defn_id = SUBSTR(f.filename,1,instr(f.filename,'.')-1)
and d.report_defn_id = 'XGF_WK_LATE'
and s.ds_type = d.ds_type
and s.ds_id = d.ds_id
and s.oprid = d.oprid
and begindttm BETWEEN TO_DATE('201001200000','yyyymmddhh24mi')
                  AND SYSDATE -- TO_DATE('201001211600','yyyymmddhh24mi')
ORDER BY r.begindttm
/
```

```
    P.I. BEGINDTTM           Report ID    Type Data Source ID           SECS
-------- ------------------- ------------ ---- ---------------------- -------
…
 1953197 19:56:56 20/01/2010 XGF_WK_LATE  QRY  XGF_WKLY_LATENESS_RPT      753
 1956338 09:01:56 21/01/2010 XGF_WK_LATE  QRY  XGF_WKLY_LATENESS_RPT   19,283
 1956805 09:50:08 21/01/2010 XGF_WK_LATE  QRY  XGF_WKLY_LATENESS_RPT   16,350
 1956925 10:01:28 21/01/2010 XGF_WK_LATE  QRY  XGF_WKLY_LATENESS_RPT   15,654
…
```

Now I want to see what SQL Statements that were executed by those processes, and what were their execution plans.

```
SELECT /*+LEADING(r f d x h) USE_NL(h)*/
            r.prcsinstance
,           h.sql_id
--,         h.sql_child_number
,           h.sql_plan_hash_value
,           (CAST(r.enddttm AS DATE)-CAST(r.begindttm AS DATE))*86400 exec_secs
,           SUM(10) ash_secs
FROM        dba_hist_snapshot x
,           dba_hist_active_sess_history h
,           sysadm.psprcsrqst r
,      sysadm.ps_cdm_file_list f
,      sysadm.psxprptdefn d
WHERE       x.end_interval_time between r.begindttm AND r.enddttm
```

```
AND        h.sample_time BETWEEN r.begindttm AND r.enddttm
AND        h.snap_id = x.snap_id
AND        h.dbid = x.dbid
AND        h.instance_number = x.instance_number
AND        h.module = r.prcsname
AND        h.action LIKE 'PI='||r.prcsinstance||'%'
AND    r.prcsinstance = f.prcsinstance
AND    NOT f.cdm_file_type IN('AET','TRC','LOG')
AND    d.report_defn_id = SUBSTR(f.filename,1,instr(f.filename,'.')-1)
AND    d.report_defn_id = 'XGF_WK_LATE'
AND        r.prcsname = 'PSXPQRYRPT'
AND    r.begindttm BETWEEN TO_DATE('201001200000','yyyymmddhh24mi')
                       AND TO_DATE('201001211600','yyyymmddhh24mi')
GROUP BY r.prcsinstance, r.prcsname, r.begindttm, r.enddttm, h.sql_id, h.sql_plan_hash_value
ORDER BY 1
/
```

One of the challenges of PeopleSoft Queries with Operator related row-level security is that a precate on the operator ID as added to the query, and the operator ID is a litteral value not a bind variable. That means that if two different operators run the same query, they will generate different SQL_IDs.

```
SQL_ID djqf1zcypm5fm
--------------------
SELECT ...
FROM PS_TL_EXCEPTION A, PS_PERSONAL_DATA B, PS_PERALL_SEC_QRY B1,
…
WHERE B.EMPLID = B1.EMPLID      AND B1.OPRID = '12345678'
…
```

This is rather perverse considering all the other parameters in a query are proper bind variables, so if a use runs the same query with different paramters that will usually have the same SQL_ID!

Most the SQL_IDs in this report are essentially the same query with different Operator IDs, and you can see that there are 4 different execution plans.

```
    P.I. SQL_ID        SQL_PLAN_HASH_VALUE  EXEC_SECS   ASH_SECS
-------- ------------- ------------------- ---------- ----------
 1949129 0uj7k70z1s76y          2239378934        619        210
 1949819 0sd03jvun7us6          2239378934        336         20
 1953197 22kn2sb7vttnp          2239378934        753        150
 1956338 0xkjtywub2861          2602481067      19283      18550
 1956338 998wf4g84dk8z          1041940423      19283         10
 1956805 7c7dzavm70yku          2602481067      16350      15690
 1956925 1knvx57dnrz29          2602481067      15654      15010
 1956925 a9mw8hjxfwczm           338220129      15654         10
 1957008 9s2jct0jfmwgy          2602481067      15077      14430
 1957008 9s2jct0jfmwgy          3265949623      15077         10
 1957087 cwarnq7kv4d84          2602481067      14638      14000
 1957691 9nv93p134xjb0          2602481067      13477      12980
 1958659 9s2jct0jfmwgy          2602481067       9354       9140
 1958697 1bd0fg0fvsfyp          2602481067       9176       8950
 1958742 1knvx57dnrz29          2602481067       8903       8680
 1958873 6uzhyw11wxwqn          2602481067       8025       7810
 1958963 3ydv1rbx5yut1          2602481067       7294       7100
 1958963 bct3ytxuby0wm           481148914       7294         10
 1959099 0yf3nx1tm4f18          2602481067       6084       5690
 1959525 7gu27skrd5uvu          2602481067       5621       5230
 1959645 6wxbk0rkgm08a          2602481067       5148       4550
 1959716 c7btm765fcrjy          2602481067       4706       4100
 1959763 ffjj75qcv9a3a          2602481067       4342       3740
 1959773 5c2x8b7ur4hzj          2602481067       6361       5810
 1960066 46smbgcfcrb8d          2602481067       5766       5210
```

This is one of those situations where it can be effective to just GROUP BY SQL_PLAN_HASH_VALUE and work out which execution plan has the most execution plan. That is might be an undesirable plan and you might want to work out why Oracle is choosing it, and consider what you are going to do about it.

Other Techniques

Monitoring Progress of Processes in Real Time

```
SELECT     /*+LEADING(r)*/
           r.prcsinstance
,          h.sql_id
,          h.sql_child_number
,          h.sql_plan_hash_value
,          (NVL(r.enddttm,SYSDATE)-r.begindttm)*86400 exec_secs
,          SUM(1) ash_secs
,          max(sample_time) max_sample_time
FROM       v$active_Session_history h
,          sysadm.psprcsrqst r
WHERE      h.sample_time BETWEEN r.begindttm AND NVL(r.enddttm,SYSDATE)
AND        h.module = r.prcsname
AND        h.action LIKE 'PI='||r.prcsinstance||'%'
AND        r.prcsinstance = 1561519
GROUP BY   r.prcsinstance, r.prcsname, r.begindttm, r.enddttm, h.sql_id,
h.sql_plan_hash_value, h.sql_child_number
ORDER BY   max_sample_time desc
```

This was run on a fairly quiet database and the ASH buffer has held 5 hours of data.

Note that Statement 9yj020x2762a9 has clocked 17688 seconds at 4.24pm.

```
   Process                Child   SQL Plan   Exec    ASH
  Instance SQL_ID           No. Hash Value   Secs   Secs Last Running
---------- ------------- ------ ---------- ------ ------ -------------------------
   1561509 9yj020x2762a9      0 3972644945  18366  17688 19-FEB-10 04.24.41.392 PM
   1561509 9yj020x2762a9      0  799518913  18366      1 19-FEB-10 11.26.29.096 AM
   1561509 b5r9c04ck29zb      1  149088295  18366      1 19-FEB-10 11.26.28.085 AM
   1561509 5vdhh2m8skh86      1          0  18366      1 19-FEB-10 11.26.27.075 AM
   1561509 gyuq5arbj7ykx      0 3708596767  18366      1 19-FEB-10 11.26.26.065 AM
   1561509                    0          0  18366      1 19-FEB-10 11.26.25.055 AM
   1561509 5jkh8knvxw7k2      0 1549543019  18366      1 19-FEB-10 11.26.24.043 AM
   1561509 9pz262n5gbhmk      0 1935542594  18366      1 19-FEB-10 11.26.23.033 AM
   1561509 6qg99cfg26kwb      1 3610545376  18366      1 19-FEB-10 11.26.22.035 AM
   1561509 gpdwr389mg61h      0  672996088  18366    422 19-FEB-10 11.26.21.014 AM
   1561509 gpdwr389mg61h      0 3588911518  18366      1 19-FEB-10 11.19.13.931 AM
   1561509 fmbbqm351p05q      0 2548875690  18366      1 19-FEB-10 11.19.12.916 AM
   1561509 dwfwa9bsgsnv3      0 2495151791  18366     14 19-FEB-10 11.19.11.912 AM
   1561509 d0wu61901pbx4      0 3123499903  18366      9 19-FEB-10 11.18.57.771 AM
   1561509 g7psub9favw54      0 2314801731  18366     10 19-FEB-10 11.18.48.679 AM
   1561509 cbppam9ph5bu8      0          0  18366      1 19-FEB-10 11.18.38.571 AM
   1561509 cbppam9ph5bu8      0 3488560417  18366      1 19-FEB-10 11.18.37.551 AM
   1561509 3cswz2x9ubjm3      0  504495601  18366      1 19-FEB-10 11.18.36.541 AM
```

But later not that the timings for statement 9yj020x2762a9, the timing has gone down. So part of the ASH data has been purged.

```
                             SQL
   Process                Child   SQL Plan   Exec    ASH
  Instance SQL_ID           No. Hash Value   Secs   Secs Last Running
---------- ------------- ------ ---------- ------ ------ -------------------------
```

```
   1561509 gdcva48t01v3m      1  915452742  38153      1 19-FEB-10 09.54.27.827 PM
   1561509 3snbjfz6zqcus      1          0  38153      1 19-FEB-10 09.54.26.817 PM
   1561509 d4v0gbxwdkgju      1  557995251  38153      1 19-FEB-10 09.54.25.807 PM
   1561509 apn21px6qggpk      0 1655174710  38153   1077 19-FEB-10 09.54.24.798 PM
   1561509 9md3rncjkx42h      0 2227914321  38153    188 19-FEB-10 09.36.15.070 PM
   1561509 62ct90nt8wu8v      0 3123499903  38153     49 19-FEB-10 09.33.04.612 PM
   1561509 1gpsnf5s10r9m      0 1906339927  38153      1 19-FEB-10 09.32.15.018 PM
   1561509 7ca17q7c99dgq      0 3827753996  38153    100 19-FEB-10 09.32.13.994 PM
   1561509 64a4yfs60t9rf      0 1488496785  38153     98 19-FEB-10 09.30.32.216 PM
   1561509 5zq8mtxp0nfn8      0 1505304026  38153      1 19-FEB-10 09.28.52.628 PM
   1561509 b023ph16myv5d      0 1416307094  38153     30 19-FEB-10 09.28.51.618 PM
   1561509 b023ph16myv5d      0   51594791  38153      1 19-FEB-10 09.28.21.300 PM
   1561509 14k7bqan2vfh8      0 1620828024  38153      1 19-FEB-10 09.28.20.280 PM
   1561509 d2498j5x025rq      0 3746253366  38153     82 19-FEB-10 09.28.19.270 PM
   1561509 fsywq5xqn66nf      0 3232283327  38153     43 19-FEB-10 09.26.54.280 PM
   1561509 4z29htzn27cct      0  763665386  38153     14 19-FEB-10 09.24.54.853 PM
   1561509 4z29htzn27cct      0 3569720797  38153      1 19-FEB-10 09.24.27.533 PM
   1561509 a4zg5sgfc23kt      0 1936785589  38153     78 19-FEB-10 09.24.26.523 PM
   1561509 8x1u4hd6jq6pg      0 2692129132  38153     42 19-FEB-10 09.23.07.685 PM
   1561509 amakpc5aqxvh4      0 3033962754  38153      3 19-FEB-10 09.22.25.207 PM
   1561509 8za7232u5pnrf      0 3717166321  38153  13296 19-FEB-10 09.22.21.167 PM
   1561509 8za7232u5pnrf      0 2937741215  38153      1 19-FEB-10 05.38.13.085 PM
   1561509 8msvfudz3bc1w      0 1444355751  38153     24 19-FEB-10 05.38.11.939 PM
   1561509 5fvtbncfpkbuu      0 1444355751  38153     32 19-FEB-10 05.37.47.615 PM
   1561509 59sdxn718fs8w      0 1746491243  38153     11 19-FEB-10 05.37.13.236 PM
   1561509 g0by0mj1d6dy2      0 2128929267  38153      1 19-FEB-10 05.37.02.049 PM
   1561509 7sx5p1ug5ag12      1 2873308018  38153      1 19-FEB-10 05.37.01.033 PM
   1561509 9yj020x2762a9      0 3972644945  38153  13295 19-FEB-10 05.36.59.620 PM
```

And if I want to look at an execution plan

```
SELECT DISTINCT 'SELECT * FROM table(dbms_xplan.display_cursor('''||sql_id||''','||sql_child_number||',''ADVANCED''));'
FROM (
…
)
```

To generate this command

```
SELECT * FROM table(dbms_xplan.display_cursor('9yj020x2762a9',0,'ADVANCED'));
```

Developers not Using Bind Variables

This is what happens when developers do not use Bind Variables. It happens in PeopleSoft Application Engine programs if developers do not use the ReUse statement option, which is not enabled by default. It can also happen when a process uses dynamically generated SQL.

I started with my standard query for analysing a named process.

```
SELECT /*+LEADING(r x h) USE_NL(h)*/
           r.prcsinstance
,          h.sql_id
,          h.sql_plan_hash_value
,          (CAST(r.enddttm AS DATE)-CAST(r.begindttm AS DATE))*86400
exec_secs
,          SUM(10) ash_secs
FROM       dba_hist_snapshot x
,          dba_hist_active_sess_history h
,          sysadm.psprcsrqst r
WHERE      x.end_interval_time >= r.enddttm
And        x.begin_interval_time <= r.enddttm
AND        h.sample_time BETWEEN r.begindttm AND r.enddttm
and        h.snap_id = x.snap_id
AND        h.dbid = x.dbid
AND        h.instance_number = x.instance_number
AND        h.module = r.prcsname
AND        h.action LIKE 'PI='||r.prcsinstance||'%'
AND        r.prcsname = 'XXES036'
GROUP BY   r.prcsinstance, r.prcsname, r.begindttm, r.enddttm
, h.sql_id, h.sql_plan_hash_value
ORDER BY ash_secs DESC
```

I got lots of SQL statements with the same execution plan. That is going to happen when the statements are very similar, and/or when the only differences are the values of literals in the SQL.

SQL*Trace profiled TKPROF has the same problem. This is a challenge that I face very frequently, and ORASRP is a better profiling tool.

```
PRCSINSTANCE SQL_ID        SQL_PLAN_HASH_VALUE  EXEC_SECS   ASH_SECS
------------ ------------- ------------------- ---------- ----------
    50002824                                 0      10306         50
    50002824 2ybtak62vmx58          2262951047      10306         20
    50002824 ck3av6cnquwfc          2262951047      10306         20
    50002824 gvys6kd9fqn7u          2262951047      10306         20
    50002824 7ymcbn6q8utj8          2262951047      10306         10
    50002824 9qud2n3qq7nzr          2262951047      10306         10
    50002824 6pxvns97m1fua          2262951047      10306         10
    50002824 5ngqj5zg8vbz8          2262951047      10306         10
    50002824 9zp6nndfvn66b          2262951047      10306         10
    50002824 15kfs3c3005xm          2262951047      10306         10
    50002824 4qvhpygc7cq2t          2262951047      10306         10
    50002824 23yc8dcz9z4yj          2262951047      10306         10
    50002824 bn8xczrvs2hpr          2262951047      10306         10
    50002824 9g6k9dnrjap08          2262951047      10306         10
    50002824 1art8dhzbvpwt          2262951047      10306         10
    50002824 6gqj337xnr5y4          2262951047      10306         10
    50002824 77rx2ctnzwcgf          2262951047      10306         10
    50002824 5p5tvh4wfp1ur          2262951047      10306         10
…
```

So now, I will remove SQL ID FROM my query, and just GROUP BY SQL Plan Hash Value

```
SELECT /*+LEADING(r x h) USE_NL(h)*/
           r.prcsinstance
,          h.sql_plan_hash_value
,          (CAST(r.enddttm AS DATE)-CAST(r.begindttm AS DATE))*86400
exec_secs
,          SUM(10) ash_secs
FROM       dba_hist_snapshot x
,          dba_hist_active_sess_history h
,          sysadm.psprcsrqst r
WHERE      x.end_interval_time >= r.enddttm
And        x.begin_interval_time <= r.enddttm
AND        h.sample_time BETWEEN r.begindttm AND r.enddttm
and        h.snap_id = x.snap_id
AND        h.dbid = x.dbid
AND        h.instance_number = x.instance_number
AND        h.module = r.prcsname
AND        h.action LIKE 'PI='||r.prcsinstance||'%'
AND        r.prcsname = 'XXES036'
GROUP BY   r.prcsinstance, r.prcsname, r.begindttm, r.enddttm
, h.sql_plan_hash_value
ORDER BY ash_secs DESC
```

Now, most of my time is in one execution plan.

```
PRCSINSTANCE SQL_PLAN_HASH_VALUE  EXEC_SECS   ASH_SECS
------------ ------------------- ---------- ----------
    50002824          2262951047      10306       2300
    50002824                   0      10306         60
    50002824          3085938243      10306         20
    50002824           563410926      10306         10
    50002824          1068931976      10306         10
```

Now, I need to look at at least one of those SQL statements with that plan

```
SELECT * FROM table(dbms_xplan.display_awr('9vnan5kqsh1aq', 2262951047,NULL,'ADVANCED'));
```

This query groups the SQL by SQL_ID and SQL PLAN hash plan, but reports the total amount of time for each plan in ASH, it ranks the statements within each plan by the amount of time recorded against statements captured by AWR.

```
SELECT 'SELECT * FROM
table(dbms_xplan.display_awr('''||sql_id||''','||sql_plan_hash_value||',NULL,''ADVANCED''))/*'||tot_ash_secs||','||
tot_awr_secs||'*/;'
from     (
         SELECT   ROW_NUMBER()over (PARTITION BY x.sql_plan_hash_value order by x.awr_secs desc) as ranking
         ,        x.sql_id, x.sql_plan_hash_value
         ,        SUM(x.ash_secs) over (PARTITION BY x.sql_plan_hash_value) tot_ash_secs
         ,        SUM(x.awr_secs) over (PARTITION BY x.sql_plan_hash_value) tot_awr_secs
         ,        COUNT(distinct sql_id) over (PARTITION BY x.sql_plan_hash_value) sql_ids
         FROM     (
                  SELECT   h.sql_id
                  ,        h.sql_plan_hash_value
                  ,        SUM(10) ash_secs
                  ,        10*count(t.sql_id) awr_secs
                  from     DBA_HIST_SNAPSHOT x
         ,        DBA_HIST_ACTIVE_SESS_HISTORY h
                           LEFT OUTER JOIN dba_hist_sqltext t[16]
                           ON t.sql_id = h.sql_id
         WHERE    x.end_interval_time >= TRUNC(SYSDATE,'mm')
         AND      x.begin_interval_time <= TRUNC(SYSDATE,'mm')+7
         AND      h.sample_time BETWEEN TRUNC(SYSDATE,'mm') AND TRUNC(SYSDATE,'mm')+7
         and      h.snap_id = x.snap_id
         and      h.dbid = x.dbid
         and      h.instance_number = x.instance_number
         and      h.module = h.program
         group by h.sql_id, h.sql_plan_hash_value
         ) x
) y
where    y.ranking = 1
and tot_ash_secs > 900
order by tot_ash_secs desc, ranking
/
```

```
   RANKING SQL_ID        SQL_PLAN_HASH_VALUE TOT_ASH_SECS TOT_AWR_SECS    SQL_IDS
---------- ------------- ------------------- ------------ ------------ ----------
         1 8mkvraydrxycn                   0        38270          480         74[17]
         1 027qsfj7n71cy          1499159071         4230         4230          1[18]
         1 cxwz9m3auk4y7          1898065720         4190         4190        198[19]
         1 9513hhu1vucxz          2044891559         3590         3590          1
```

[16] By outer joining the ASH data to DBA_HIST_SQLTEXT we can check whether the statement was captures by AWR

[17] The first statement is a special case. There is no plan – probably because it’s a PL/SQL function. There were 74 statements, but in reality they will all be totally different..

[18] One SQL, one plan, this is a shareable SQL_ID, or it did just execute once.

[19] This is many statements with the same plan, at least 198.

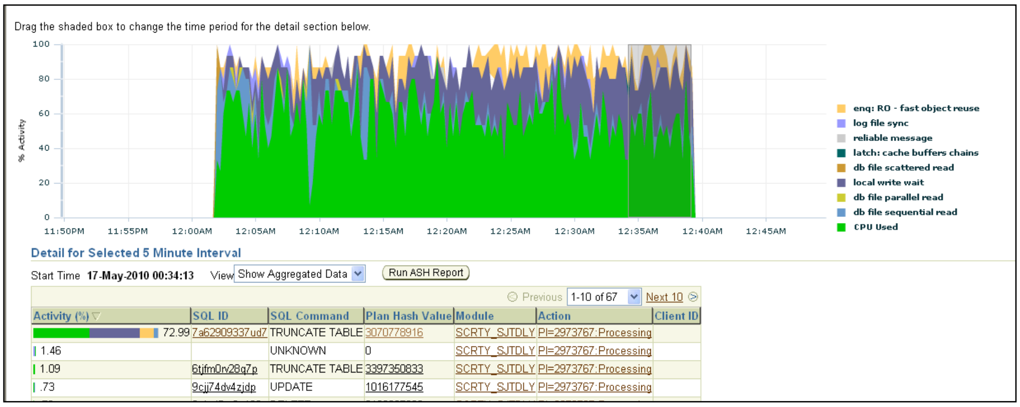
```
        1 95dx0mkjq38v5          1043916244         3450          3450          23
…
```

```
SELECT * FROM table(dbms_xplan.display_awr('8mkvraydrxycn',0,NULL,'ADVANCED'))/*38270,480*/;
SELECT * FROM table(dbms_xplan.display_awr('027qsfj7n71cy',1499159071,NULL,'ADVANCED'))/*4230,4230*/;
SELECT * FROM table(dbms_xplan.display_awr('cxwz9m3auk4y7',1898065720,NULL,'ADVANCED'))/*4190,4190*/;
SELECT * FROM table(dbms_xplan.display_awr('9513hhu1vucxz',2044891559,NULL,'ADVANCED'))/*3590,3590*/;
SELECT * FROM table(dbms_xplan.display_awr('95dx0mkjq38v5',1043916244,NULL,'ADVANCED'))/*3450,3450*/;
…
```

How Many Executions?

Oracle 10g

In 10g you cannot directly determine the number of executions from ASH data. Here is an example from OEM. This truncate statement is consuming a lot of time. But it isn't a single execution. It is a huge number of small executions.



Detail for Selected 5 Minute Interval

Start Time **17-May-2010 00:34:13** View Show Aggregated Data

1-10 of 67

Activity (%)	SQL ID	SQL Command	Plan Hash Value	Module	Action	Client ID
72.99	7a62909337ud7	TRUNCATE TABLE	3070778916	SCRTY_SJTDLY	PI=2973767:Processing	
1.46		UNKNOWN	0	SCRTY_SJTDLY	PI=2973767:Processing	
1.09	6tjfm0rv28q7p	TRUNCATE TABLE	3397350833	SCRTY_SJTDLY	PI=2973767:Processing	
.73	9cjj74dv4zjdp	UPDATE	1016177545	SCRTY_SJTDLY	PI=2973767:Processing	

Oracle 11g

However, in 11g there is a new column *sql_exec_id* in the *v$active_session_history* and *dba_hist_active_sess_history*. Each execution of a statement gets a unique execution ID. Counting the number of distinct execution IDs determines the number of executions.

```
select /*+leading(x h) use_nl(h)*/
            h.program
,           h.sql_id
,           h.sql_plan_hash_value
,           sum(10) ash_secs
,           COUNT(distinct xid) XIDs
,           COUNT(distinct h.sql_exec_id) Execs
,           count(distinct h.session_id) users
,           min(h.sample_time)+0 min_sample_time
,           max(h.sample_time)+0 max_sample_time
From        DBA_HIST_SNAPSHOT x
,           DBA_HIST_ACTIVE_SESS_HISTORY h
WHERE       X.END_INTERVAL_TIME   >= TO_DATE('201102211540', 'yyyymmddhh24mi')
AND         x.begin_interval_time <= TO_DATE('201102211510', 'yyyymmddhh24mi')
and         h.sample_TIME         >= TO_DATE('201102211510', 'yyyymmddhh24mi')
AND         h.sample_time         <= TO_DATE('201102211540', 'yyyymmddhh24mi')
and         h.SNAP_id = X.SNAP_id
and         h.dbid = x.dbid
and         h.instance_number = x.instance_number
and         h.user_id != 0 /*omit oracle shadow processes*/
group by h.program, h.sql_id, h.sql_plan_hash_value
order by ash_secs desc
/
```

So I can see that these statements burnt about 3020 and 320 seconds. This query has counted 297 and 32 executions respectively.

```
                            SQL Plan    ASH
PROGRAM     SQL_ID        Hash Value   Secs   XIDS  EXECS  USERS First Running       Last Running
----------- ------------- ---------- ------ ------ ------ ------ ------------------- ---------------
t_async.exe 7q90ra0vmd9xx 2723153562   3020      0    297     20 15:10:21 21/02/2011 15:37:21 21/02/2011
t_async.exe 6mw25bgbh1stj 1229059401    320      0     32     17 15:19:49 21/02/2011 15:37:31 21/02/2011
…
```

However, remember that because this query was based on *dba_hist_active_sess_history* there is one sample per 10 seconds, so each row is counted as 10 seconds. The number of executions can never be calculated as being greater than the number of ASH records. So when the number of executions is close to or the same as the number of ASH records it is likely that there are actually many more executions that are recorded here.

How Many Transactions?

You cannot tell how many times a statement has executed in 10g. This becomes possible in 11g. However, you do have the transaction ID is recorded in the ASH data, but only if the statement is a part of a transaction.

```
Column last_sample_time format a25
Column first_sample_time format a25
select /*+leading(r h) use_nl(h)*/
       r.prcsinstance
--,    h.sql_id
--,    h.sql_child_number
,      h.xid
,      h.sql_plan_hash_value
,      (NVL(r.enddttm,SYSDATE)-r.begindttm)*86400 exec_secs
,      sum(1) ash_secs
,      min(sample_Time) first_sample_time
,      max(sample_Time) last_sample_time
FROM   gv$active_session_history h
,      sysadm.psprcsrqst r
WHERE  h.sample_time BETWEEN r.begindttm AND NVL(r.enddttm,SYSDATE)
AND    h.module = r.prcsname
AND    h.action LIKE 'PI='||r.prcsinstance||'%'
AND    r.prcsinstance = 10026580
AND    h.sql_id = 'dungu07axr0z5'
group by r.prcsinstance, r.prcsname, r.begindttm, r.enddttm
, h.sql_id, h.sql_plan_hash_value
, h.sql_child_number
, h.xid
--, h.program
--having sum(1) > (NVL(r.enddttm,SYSDATE)-r.begindttm)*86400/1000
order by last_sample_time, ash_secs desc
/
```

One statement executed 4 at least times in the same process, with the same process, but as a part of 3 different transactions. Note that the last entry is not part of any transaction.

```
PRCSINSTANCE XID              SQL_PLAN_HASH_VALUE  EXEC_SECS   ASH_SECS FIRST_SAMPLE_TIME         LAST_SAMPLE_TIME
------------ ---------------- ------------------- ---------- ---------- ------------------------- ----------------------
    10026580 00080026000185A7           461068291       4774        943 23-APR-10 11.13.50.548    23-APR-10 11.29.33.546
    10026580 000100250001861A           461068291       4774        906 23-APR-10 11.30.16.590    23-APR-10 11.45.22.487
    10026580 000700280001CC47           461068291       4774        783 23-APR-10 11.46.06.543    23-APR-10 11.59.09.286
    10026580                            461068291       4774        775 23-APR-10 11.59.51.325    23-APR-10 12.12.46.056
```

When Did the Transaction Start

Here is the output for a very similar query at a different time. On these occasions the SQL starts without a transaction ID, and acquires one later.

```
                SQL Plan                     ASH   Exec
SQL_ID        Hash Value XID                Secs   Secs First Running            Last Running
------------- ---------- ---------------- ------ ------ ------------------------ ------------------
7uj72ad03k13k 3087414546                      82   1124 28-APR-10 04.42.48.662 PM 28-APR-10 04.44.10.662 PM
7uj72ad03k13k 3087414546 000A001400044C6D      1   1124 28-APR-10 04.44.11.672 PM 28-APR-10 04.44.11
1ng9qkc0zspkh 3423396304                     104   1124 28-APR-10 04.44.12.682 PM 28-APR-10 04.45.56.961 PM
1ng9qkc0zspkh 3423396304 0007002D0004116E      5   1124 28-APR-10 04.45.57.971 PM 28-APR-10 04.46.02
```

The statements involved are monolithic deletes. My interpretation is that it takes a while for these queries to identify rows to be deleted, and it is not until the first row is deleted that a transaction is initiated. It is entirely plausible that, depending upon data, statements could run for a while before finding some data to delete.

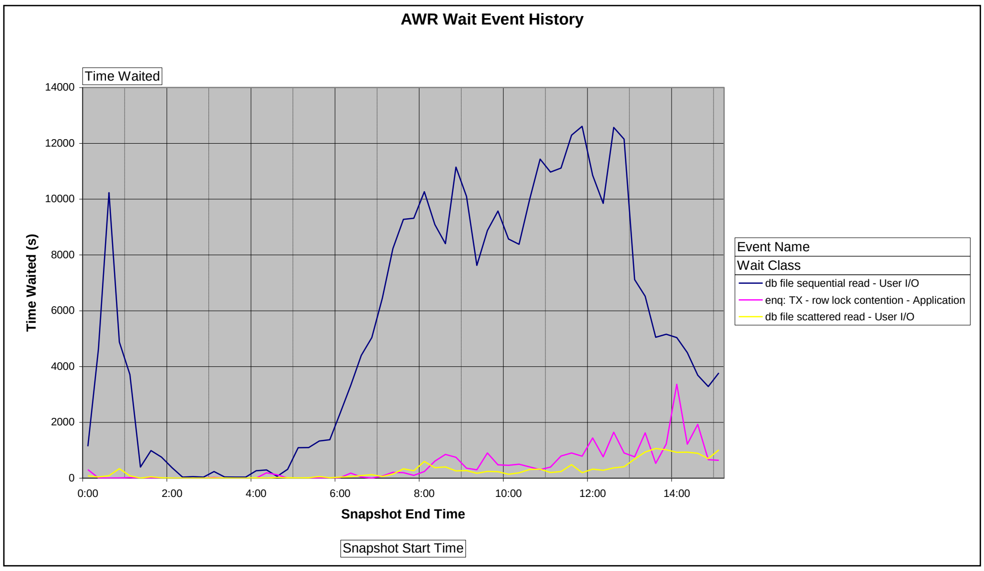
```
SQL_ID  7uj72ad03k13k, child number 0
-------------------------------------
DELETE /*GPPCANCL_D_ERNDGRP*/ FROM PS_GP_RSLT_ERN_DED WHERE EMPLID BETWEEN :1 AND :2 AND CAL_RUN_ID=
EMPLID IN (SELECT EMPLID FROM PS_GP_GRP_LIST_RUN WHERE RUN_CNTL_ID=:4 AND OPRID=:5) AND EXISTS (SELE
FROM PS_GP_PYE_RCLC_WRK RW WHERE RW.CAL_ID = PS_GP_RSLT_ERN_DED.CAL_ID AND RW.CAL_RUN_ID =
PS_GP_RSLT_ERN_DED.CAL_RUN_ID AND RW.GP_PAYGROUP = PS_GP_RSLT_ERN_DED.GP_PAYGROUP AND RW.EMPLID BETW
AND :7 AND RW.CAL_RUN_ID = :8 AND RW.EMPLID = PS_GP_RSLT_ERN_DED.EMPLID AND RW.EMPL_RCD =
PS_GP_RSLT_ERN_DED.EMPL_RCD)

Plan hash value: 3087414546
----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name               | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
----------------------------------------------------------------------------------------------------
|   0 | DELETE STATEMENT              |                    |       |       |     5 (100)|          |       |       |
|   1 |  DELETE                       | PS_GP_RSLT_ERN_DED |       |       |            |          |       |       |
|*  2 |   FILTER                      |                    |       |       |            |          |       |       |
|   3 |    NESTED LOOPS SEMI          |                    |     1 |   172 |     5  (20)| 00:00:01 |       |       |
|*  4 |     HASH JOIN SEMI            |                    |     1 |   131 |     5  (20)| 00:00:01 |       |       |
|   5 |      PARTITION RANGE ITERATOR|                    |     2 |   164 |     2   (0)| 00:00:01 |   KEY |   KEY |
|*  6 |       INDEX RANGE SCAN        | PS_GP_RSLT_ERN_DED |     2 |   164 |     2   (0)| 00:00:01 |   KEY |
|*  7 |      TABLE ACCESS FULL        | PS_GP_PYE_RCLC_WRK |    15 |   735 |     2   (0)| 00:00:01 |       |
|   8 |     PARTITION RANGE ITERATOR  |                    |     1 |    41 |     0   (0)|          |   KEY |   KEY |
|*  9 |      INDEX RANGE SCAN         | PS_GP_GRP_LIST_RUN |     1 |    41 |     0   (0)|          |   KEY |   KEY |
----------------------------------------------------------------------------------------------------
```

```
PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------------
SQL_ID  1ng9qkc0zspkh, child number 0
-------------------------------------
DELETE /*GPPCANCL_D_PINGRP*/ FROM PS_GP_RSLT_PIN WHERE EMPLID BETWEEN :1 AND :2 AND CAL_RUN_ID=:3 AN
EMPLID IN (SELECT EMPLID FROM PS_GP_GRP_LIST_RUN WHERE RUN_CNTL_ID=:4 AND OPRID=:5) AND EXISTS (SELE
FROM PS_GP_PYE_RCLC_WRK RW WHERE RW.CAL_ID = PS_GP_RSLT_PIN.CAL_ID AND RW.CAL_RUN_ID =
PS_GP_RSLT_PIN.CAL_RUN_ID AND RW.GP_PAYGROUP = PS_GP_RSLT_PIN.GP_PAYGROUP AND RW.EMPLID BETWEEN :6 A
AND RW.CAL_RUN_ID = :8 AND RW.EMPLID = PS_GP_RSLT_PIN.EMPLID AND RW.EMPL_RCD = PS_GP_RSLT_PIN.EMPL_R

Plan hash value: 3423396304


----------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name               | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
----------------------------------------------------------------------------------------------------
|   0 | DELETE STATEMENT             |                    |       |       |     5 (100)|          |       |       |
|   1 |  DELETE                      | PS_GP_RSLT_PIN     |       |       |            |          |       |       |
|*  2 |   FILTER                     |                    |       |       |            |          |       |       |
|   3 |    NESTED LOOPS SEMI         |                    |     1 |   170 |     5  (20)| 00:00:01 |       |       |
|*  4 |     HASH JOIN SEMI           |                    |     1 |   129 |     5  (20)| 00:00:01 |       |       |
|   5 |      PARTITION RANGE ITERATOR|                    |    31 |  2480 |     2   (0)| 00:00:01 |   KEY |   KEY |
|   6 |       PARTITION LIST SINGLE  |                    |    31 |  2480 |     2   (0)| 00:00:01 |   KEY |   KEY |
|*  7 |        INDEX RANGE SCAN      | PS_GP_RSLT_PIN     |    31 |  2480 |     2   (0)| 00:00:01 |   KEY
|*  8 |      TABLE ACCESS FULL       | PS_GP_PYE_RCLC_WRK |    15 |   735 |     2   (0)| 00:00:01 |       |
|   9 |     PARTITION RANGE ITERATOR |                    |     1 |    41 |     0   (0)|          |   KEY |   KEY |
|* 10 |      INDEX RANGE SCAN        | PS_GP_GRP_LIST_RUN |     1 |    41 |     0   (0)|          |   KEY |   KEY |
----------------------------------------------------------------------------------------------------


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


   2 - filter((:7>=:1 AND :6<=:2 AND :6<=:7 AND :1<=:2 AND :8=:3))
   4 - access("RW"."CAL_ID"="PS_GP_RSLT_PIN"."CAL_ID" AND "RW"."CAL_RUN_ID"="PS_GP_RSLT_PIN"."CAL_RU
              AND "RW"."GP_PAYGROUP"="PS_GP_RSLT_PIN"."GP_PAYGROUP" AND "RW"."EMPLID"="PS_GP_RSLT_PIN"."EMP
              "RW"."EMPL_RCD"="PS_GP_RSLT_PIN"."EMPL_RCD")
   7 - access("EMPLID">=:1 AND "PS_GP_RSLT_PIN"."CAL_RUN_ID"=:8 AND "EMPLID"<=:2)
       filter(("CAL_RUN_ID"=:3 AND "PS_GP_RSLT_PIN"."CAL_RUN_ID"=:8 AND "PS_GP_RSLT_PIN"."EMPLID">=:
              "PS_GP_RSLT_PIN"."EMPLID"<=:7))
   8 - filter(("RW"."CAL_RUN_ID"=:8 AND "RW"."CAL_RUN_ID"=:3 AND "RW"."EMPLID">=:6 AND "RW"."EMPLID"
              AND "RW"."EMPLID">=:1 AND "RW"."EMPLID"<=:2))
  10 - access("RUN_CNTL_ID"=:4 AND "OPRID"=:5 AND "EMPLID"="EMPLID")
       filter(("EMPLID">=:1 AND "EMPLID"<=:2 AND "EMPLID">=:6 AND "EMPLID"<=:7 AND "EMPLID"="EMPLID"

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

Single Wait Event

Earlier we looked at an example of on-line activity, and I used the PeopleSoft Performance Monitor to identify a period when degradation in performance was noticed (see Application Engine from PeopleTools 8.52 on page 16). I want to look at the behaviour of the database in the same period.

Oracle Enterprise Manager will give you a graphical representation of the ASH data. I often graph wait event data collected by AWR in excel[20].



According to AWR, we have as many of 12 concurrent sessions waiting on this event.

Time Waited	**Event Name**	Wait Class
	db file sequential read	enq: TX - row lock contention
Snapshot Start Time	User I/O	Application
Mon 1.2.10 06:00	2,329.153	16.822
Mon 1.2.10 06:15	3,323.358	174.772
Mon 1.2.10 06:30	4,397.850	41.172
Mon 1.2.10 06:45	5,037.319	1.595
Mon 1.2.10 07:00	6,451.124	72.692
Mon 1.2.10 07:15	8,226.684	205.765
Mon 1.2.10 07:30	9,274.853	196.430
Mon 1.2.10 07:45	9,315.794	99.286
Mon 1.2.10 08:00	10,267.237	233.664
Mon 1.2.10 08:15	9,084.140	607.859
Mon 1.2.10 08:30	8,404.167	845.342
Mon 1.2.10 08:45	11,145.149	746.139
Mon 1.2.10 09:00	10,097.621	352.595
Mon 1.2.10 09:15	7,625.934	298.300
Mon 1.2.10 09:30	8,876.006	896.529
Grand Total	113,856.388	4,788.961

[20] There are various advantanges to this approach, see http://blog.go-faster.co.uk/2008/12/graphing-awr-data-in-excel.html

A simple variant on the usual query, and we can look for the statement with the highest I/O overhead.

```
SELECT /*+LEADING(x h) USE_NL(h)*/
       h.sql_id
,      h.sql_plan_hash_value
,      SUM(10) ash_secs
FROM   dba_hist_snapshot x
,      dba_hist_active_sess_history h
WHERE  x.end_interval_time <= TO_DATE('201002010830','yyyymmddhh24mi')
AND    x.begin_interval_time >= TO_DATE('201002010730','yyyymmddhh24mi')
AND    h.sample_time BETWEEN TO_DATE('201001261100','yyyymmddhh24mi')
                     AND     TO_DATE('201001261300','yyyymmddhh24mi')
AND    h.snap_id = x.snap_id
AND    h.dbid = x.dbid
AND    h.instance_number = x.instance_number
AND    h.event = 'db file sequential read'
GROUP BY h.sql_id, h.sql_plan_hash_value
ORDER BY ash_secs DESC
/
```

So, here at the top statements

```
                SQL Plan
SQL_ID        Hash Value   ASH_SECS
------------- ---------- ----------
90pp7bcnmz68r 2961772154       2490
81gz2rtabaa8n 1919624473       2450
7hvaxp65s70qw 1051046890       1320
7fk8raq16ch0u 3950826368        890
9dzpwkff7zycg 2020614776        840
…
```

And just for a laugh, this is the query

```
SQL_ID 90pp7bcnmz68r
--------------------
SELECT DISTINCT A.GP_PAYGROUP, M.XGF_REGION_NAME, M.XGF_AREA_NAME, A.LOCATION, B.DESCR, D.DESCR, A.EMPLID,
C.LAST_NAME, C.FIRST_NAME, TO_CHAR(A.TERMINATION_DT,'YYYY-MM-DD'), TO_CHAR(A.LAST_DATE_WORKED,'YYYY-MM-DD'),
G.PIN_NET_VAL,B.SETID,B.LOCATION,TO_CHAR(B.EFFDT,'YYYY-MM-DD'),D.SETID,D.DEPTID,TO_CHAR(D.EFFDT,'YYYY-MM-DD')
FROM PS_JOB A, PS_XGF_JOB_QRY A1, PS_LOCATION_TBL B, PS_PERSONAL_DATA C, PS_PERALL_SEC_QRY C1, PS_DEPT_TBL D,
PS_XGF_TREE_RP1_VW M, PS_GP_PYE_SEG_STAT G, PS_EMPLMT_SRCH_QRY G1, PS_GP_CAL_RUN_DTL F
```

```
WHERE A.EMPLID = A1.EMPLID AND A.EMPL_RCD = A1.EMPL_RCD AND A1.OPRID = 'batchuser' AND C.EMPLID = C1.EMPLID AND
C1.OPRID = 'batchuser' AND G.EMPLID = G1.EMPLID AND G.EMPL_RCD = G1.EMPL_RCD AND G1.OPRID = 'batchuser' AND (
A.EFFDT =  (SELECT MAX(A_ED.EFFDT) FROM PS_JOB A_ED  WHERE A.EMPLID = A_ED.EMPLID  AND A.EMPL_RCD = A_ED.EMPL_RCD
AND A_ED.EFFDT <= ( F.PRD_END_DT+1)) AND A.EFFSEQ =  (SELECT MAX(A_ES.EFFSEQ) FROM PS_JOB A_ES  WHERE A.EMPLID =
A_ES.EMPLID  AND A.EMPL_RCD = A_ES.EMPL_RCD  AND A.EFFDT = A_ES.EFFDT) AND A.ACTION = 'DEA' AND A.PER_ORG = 'EMP'
AND F.GP_PAYGROUP = A.GP_PAYGROUP AND F.CALC_TYPE = 'P' AND F.RUN_TYPE <> 'RT MIG' AND F.CAL_IDNT_TS IS NOT NULL
AND F.CAL_IDNT_TS = (SELECT MAX( N.CAL_IDNT_TS) FROM PS_GP_CAL_RUN_DTL N WHERE N.GP_PAYGROUP = F.GP_PAYGROUP AND
N.CALC_TYPE = F.CALC_TYPE) AND ((A.TERMINATION_DT >= F.PRD_BGN_DT AND A.TERMINATION_DT <= F.PRD_END_DT) OR (
A.TERMINATION_DT < F.PRD_BGN_DT AND A.ACTION_DT >= (SELECT TO_DATE(MAX( O.CAL_FINAL_TS)) FROM PS_GP_CAL_RUN_DTL O
WHERE O.GP_PAYGROUP = A.GP_PAYGROUP AND O.CALC_TYPE = 'P' AND O.CAL_FINAL_TS < (SELECT MAX( P.CAL_IDNT_TS) FROM
PS_GP_CAL_RUN_DTL P WHERE P.GP_PAYGROUP = O.GP_PAYGROUP AND P.CALC_TYPE = O.CALC_TYPE)) AND A.ACTION_DT <=
F.PRD_END_DT)) AND B.SETID = A.SETID_LOCATION AND B.LOCATION = A.LOCATION AND B.EFFDT =  (SELECT MAX(B_ED.EFFDT)
FROM PS_LOCATION_TBL B_ED  WHERE B.SETID = B_ED.SETID  AND B.LOCATION = B_ED.LOCATION  AND B_ED.EFFDT <=
F.PRD_END_DT) AND C.EMPLID = A.EMPLID AND D.SETID = A.SETID_DEPT AND D.DEPTID = A.DEPTID AND D.EFFDT =  (SELECT
MAX(D_ED.EFFDT) FROM PS_DEPT_TBL D_ED  WHERE D.SETID = D_ED.SETID  AND D.DEPTID = D_ED.DEPTID  AND D_ED.EFFDT <=
F.PRD_END_DT) AND M.SETID = A.SETID_DEPT AND M.TREE_NAME = 'DEPT_SECURITY' AND M.DEPTID = A.DEPTID AND G.EMPLID =
A.EMPLID AND G.CAL_RUN_ID = F.CAL_RUN_ID AND G.EMPL_RCD = A.EMPL_RCD AND G.GP_PAYGROUP = A.GP_PAYGROUP AND G.CAL_ID
= F.CAL_ID ) ORDER BY 1, 4, 5, 6, 7, 9, 8

Plan hash value: 2961772154

------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name               | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                    |       |       |  2139 (100)|          |       |       |
|   1 |  SORT UNIQUE                             |                    |     1 |   578 |  2138   (2)| 00:00:03 |       |       |
|   2 |   FILTER                                 |                    |       |       |            |          |       |       |
|   3 |    TABLE ACCESS BY INDEX ROWID           | PS_SJT_PERSON      |     2 |    72 |     4   (0)| 00:00:01 |       |       |
|   4 |     NESTED LOOPS                         |                    |     1 |   578 |  2044   (1)| 00:00:03 |       |       |
|   5 |      NESTED LOOPS                        |                    |     1 |   542 |  2040   (1)| 00:00:03 |       |       |
|   6 |       NESTED LOOPS                       |                    |     1 |   509 |  2035   (1)| 00:00:03 |       |       |
|   7 |        NESTED LOOPS                      |                    |     1 |   485 |  2034   (1)| 00:00:03 |       |       |
|   8 |         NESTED LOOPS                     |                    |     1 |   429 |  2003   (1)| 00:00:03 |       |       |
|   9 |          NESTED LOOPS                    |                    |     1 |   395 |  2001   (1)| 00:00:03 |       |       |
|  10 |           NESTED LOOPS                   |                    |     1 |   365 |  1999   (1)| 00:00:03 |       |       |
|  11 |            HASH JOIN                     |                    |    65 | 19045 |  1868   (1)| 00:00:03 |       |       |
|  12 |             TABLE ACCESS FULL            | PS_GP_CAL_RUN_DTL  |    48 |  3168 |     7   (0)| 00:00:01 |       |       |
|  13 |             TABLE ACCESS BY LOCAL INDEX ROWID | PS_GP_PYE_SEG_STAT |    18 |   900 |     2   (0)| 00:00:01 |       |       |
|  14 |              NESTED LOOPS                |                    |  8376 |  1856K|  1859   (1)| 00:00:03 |       |       |
|  15 |               NESTED LOOPS               |                    |   474 | 83898 |  1107   (1)| 00:00:02 |       |       |
|  16 |                NESTED LOOPS              |                    |   479 | 67539 |    35   (0)| 00:00:01 |       |       |
|  17 |                 NESTED LOOPS             |                    |     6 |   588 |    11   (0)| 00:00:01 |       |       |
|  18 |                  NESTED LOOPS            |                    |     1 |    72 |     4   (0)| 00:00:01 |       |       |
|  19 |                   NESTED LOOPS           |                    |     1 |    48 |     3   (0)| 00:00:01 |       |       |
|  20 |                    TABLE ACCESS BY INDEX ROWID| PSOPRDEFN          |     1 |    24 |     2   (0)| 00:00:01 |       |       |
|  21 |                     INDEX UNIQUE SCAN    | PS_PSOPRDEFN       |     1 |       |     1   (0)| 00:00:01 |       |       |
|  22 |                    TABLE ACCESS BY INDEX ROWID| PSOPRDEFN          |     1 |    24 |     1   (0)| 00:00:01 |       |       |
|  23 |                     INDEX UNIQUE SCAN    | PS_PSOPRDEFN       |     1 |       |     0   (0)|          |       |       |
|  24 |                   TABLE ACCESS BY INDEX ROWID | PSOPRDEFN          |     1 |    24 |     1   (0)| 00:00:01 |       |       |
|  25 |                    INDEX UNIQUE SCAN     | PS_PSOPRDEFN       |     1 |       |     0   (0)|          |       |       |
|  26 |                  TABLE ACCESS BY INDEX ROWID  | PS_SJT_OPR_CLS     |     6 |   156 |     7   (0)| 00:00:01 |       |       |
|  27 |                   INDEX RANGE SCAN       | PS_SJT_OPR_CLS     |     6 |       |     1   (0)| 00:00:01 |       |       |
|  28 |                 PARTITION LIST SINGLE    |                    |    83 |  3569 |     4   (0)| 00:00:01 |   KEY |   KEY |
|  29 |                  INDEX RANGE SCAN        | PSCSJT_CLASS_ALL   |    83 |  3569 |     4   (0)| 00:00:01 |     1 |     1 |
|  30 |                TABLE ACCESS BY INDEX ROWID    | PS_SJT_PERSON      |     1 |    36 |     3   (0)| 00:00:01 |       |       |
|  31 |                 INDEX RANGE SCAN         | PS_SJT_PERSON      |     1 |       |     2   (0)| 00:00:01 |       |       |
|  32 |               PARTITION RANGE ITERATOR        |                    |    31 |       |     1   (0)| 00:00:01 |   KEY |   KEY |
|  33 |                INDEX RANGE SCAN               | PS_GP_PYE_SEG_STAT |    31 |       |     1   (0)| 00:00:01 |   KEY |   KEY |
|  34 |            PARTITION RANGE ITERATOR           |                    |     1 |    72 |     2   (0)| 00:00:01 |   KEY |   KEY |
|  35 |             TABLE ACCESS BY LOCAL INDEX ROWID | PS_JOB             |     1 |    72 |     2   (0)| 00:00:01 |   KEY |   KEY |
|  36 |              INDEX RANGE SCAN                 | PSAJOB             |     1 |       |     1   (0)| 00:00:01 |   KEY |   KEY |
|  37 |               SORT AGGREGATE                  |                    |     1 |    20 |            |          |       |       |
|  38 |                PARTITION RANGE SINGLE         |                    |     1 |    20 |     2   (0)| 00:00:01 |   KEY |   KEY |
|  39 |                 INDEX RANGE SCAN              | PSAJOB             |     1 |    20 |     2   (0)| 00:00:01 |   KEY |   KEY |
|  40 |                  SORT AGGREGATE               |                    |     1 |    23 |            |          |       |       |
|  41 |                   PARTITION RANGE SINGLE      |                    |     1 |    23 |     2   (0)| 00:00:01 |   KEY |   KEY |
|  42 |                    INDEX RANGE SCAN           | PSAJOB             |     1 |    23 |     2   (0)| 00:00:01 |   KEY |   KEY |
|  43 |           TABLE ACCESS BY INDEX ROWID         | PS_LOCATION_TBL    |     1 |    30 |     2   (0)| 00:00:01 |       |       |
|  44 |            INDEX RANGE SCAN                   | PS_LOCATION_TBL    |     1 |       |     1   (0)| 00:00:01 |       |       |
|  45 |             SORT AGGREGATE                    |                    |     1 |    19 |            |          |       |       |
|  46 |              INDEX RANGE SCAN                 | PS_LOCATION_TBL    |     1 |    19 |     2   (0)| 00:00:01 |       |       |
|  47 |          TABLE ACCESS BY INDEX ROWID          | PS_DEPT_TBL        |     1 |    34 |     2   (0)| 00:00:01 |       |       |
|  48 |           INDEX RANGE SCAN                    | PS_DEPT_TBL        |     1 |       |     1   (0)| 00:00:01 |       |       |
|  49 |            SORT AGGREGATE                     |                    |     1 |    21 |            |          |       |       |
|  50 |             INDEX RANGE SCAN                  | PS_DEPT_TBL        |     1 |    21 |     2   (0)| 00:00:01 |       |       |
|  51 |         TABLE ACCESS BY INDEX ROWID           | PS_XGF_TREE        |     1 |    56 |    31   (4)| 00:00:01 |       |       |
|  52 |          INDEX RANGE SCAN                     | PS_XGF_TREE        |     1 |       |    30   (4)| 00:00:01 |       |       |
|  53 |           SORT AGGREGATE                      |                    |     1 |    28 |            |          |       |       |
|  54 |            INDEX RANGE SCAN                   | PS_XGF_TREE        |  4150 |   113K|    33  (10)| 00:00:01 |       |       |
|  55 |        TABLE ACCESS BY INDEX ROWID            | PS_PERSONAL_DATA   |     1 |    24 |     1   (0)| 00:00:01 |       |       |
|  56 |         INDEX UNIQUE SCAN                     | PS_PERSONAL_DATA   |     1 |       |     0   (0)|          |       |       |
|  57 |       TABLE ACCESS BY INDEX ROWID             | PS_SJT_PERSON      |     5 |   165 |     5   (0)| 00:00:01 |       |       |
|  58 |        INDEX RANGE SCAN                       | PSASJT_PERSON      |     5 |       |     2   (0)| 00:00:01 |       |       |
|  59 |      INDEX RANGE SCAN                         | PSASJT_PERSON      |     3 |       |     2   (0)| 00:00:01 |       |       |
|  60 |    SORT AGGREGATE                             |                    |     1 |    20 |            |          |       |       |
|  61 |     TABLE ACCESS FULL                         | PS_GP_CAL_RUN_DTL  |    14 |   280 |     7   (0)| 00:00:01 |       |       |
|  62 |      SORT AGGREGATE                           |                    |     1 |    19 |            |          |       |       |
|  63 |       FILTER                                  |                    |       |       |            |          |       |       |
|  64 |        TABLE ACCESS FULL                      | PS_GP_CAL_RUN_DTL  |    16 |   304 |     7   (0)| 00:00:01 |       |       |
|  65 |        SORT AGGREGATE                         |                    |     1 |    20 |            |          |       |       |
|  66 |         TABLE ACCESS FULL                     | PS_GP_CAL_RUN_DTL  |    14 |   280 |     7   (0)| 00:00:01 |       |       |
|  67 |          NESTED LOOPS                         |                    |     1 |    69 |     4   (0)| 00:00:01 |       |       |
|  68 |           PARTITION LIST SINGLE               |                    |     1 |    43 |     3   (0)| 00:00:01 |   KEY |   KEY |
|  69 |            INDEX RANGE SCAN                   | PSASJT_CLASS_ALL   |     1 |    43 |     3   (0)| 00:00:01 |     1 |     1 |
|  70 |           INDEX RANGE SCAN                    | PSASJT_OPR_CLS     |     1 |    26 |     1   (0)| 00:00:01 |       |       |
|  71 |            NESTED LOOPS                       |                    |     1 |    60 |     2   (0)| 00:00:01 |       |       |
|  72 |             PARTITION LIST SINGLE             |                    |     1 |    34 |     1   (0)| 00:00:01 |   KEY |   KEY |
|  73 |              INDEX RANGE SCAN                 | PSASJT_CLASS_ALL   |     1 |    34 |     1   (0)| 00:00:01 |     2 |     2 |
|  74 |             INDEX RANGE SCAN                  | PSASJT_OPR_CLS     |     1 |    26 |     1   (0)| 00:00:01 |       |       |
|  75 |              COUNT STOPKEY                    |                    |       |       |            |          |       |       |
|  76 |               FILTER                          |                    |       |       |            |          |       |       |
|  77 |                NESTED LOOPS                   |                    |     1 |    69 |     4   (0)| 00:00:01 |       |       |
|  78 |                 PARTITION LIST SINGLE         |                    |     1 |    43 |     3   (0)| 00:00:01 |   KEY |   KEY |
|  79 |                  INDEX RANGE SCAN             | PSASJT_CLASS_ALL   |     1 |    43 |     3   (0)| 00:00:01 |     1 |     1 |
|  80 |                 INDEX RANGE SCAN              | PSASJT_OPR_CLS     |     1 |    26 |     1   (0)| 00:00:01 |       |       |
------------------------------------------------------------------------------------------------------------------------------------
```

What Kind of Single Block Read

I created a temporary working storage table with a classification for each tablespace. Here my classification is by object type in the tablespace. This is relatively easy if you have a reasonable tablespace naming convention.

```
drop table dmk_data_files
/
create table dmk_data_files as
SELECT tablespace_name
, file_id
, CASE
 WHEN f.tablespace_name LIKE 'SYS%' THEN 'SYSTEM'
 WHEN f.tablespace_name LIKE 'UNDO%' THEN 'UNDO'
 WHEN f.tablespace_name LIKE '%IDX%' THEN 'INDEX'
 WHEN f.tablespace_name LIKE '%INDEX%' THEN 'INDEX'
ELSE 'TABLE'
 END as tablespace_type
FROM dba_data_files f
ORDER BY tablespace_name
/
create unique index dmk_data_files on dmk_data_files(file_id)
/
```

I recommend that you do not work directly with DBA_DATA_FILES, because the resulting query will be slow. Instead, build a working storage table.

When ASH reports a wait on file I/O it also logs the object, file and block numbers. Although, beware, because the values may not have been cleared out FROM the previous sample.

So you know which database, and hence which tablespaces was accessed.

It’s a simple matter work out how much time was spent writing to which type of tablespace

```
SELECT /*+LEADING(x h) USE_NL(h f)*/
           f.tablespace_type
,          SUM(10) ash_secs
FROM       dba_hist_snapshot x
,          dba_hist_active_sess_history h
,          dmk_data_files f
WHERE      x.end_interval_time <= TO_DATE('201002161300','yyyymmddhh24mi')
AND       x.begin_interval_time >= TO_DATE('201002161100','yyyymmddhh24mi')
AND        h.sample_time BETWEEN TO_DATE('201001261100','yyyymmddhh24mi')
                          AND     TO_DATE('201001261300','yyyymmddhh24mi')
and        h.snap_id = x.snap_id
AND        h.dbid = x.dbid
AND        h.instance_number = x.instance_number
AND        h.event LIKE 'db file%'
AND        h.p1text = 'file#'
and        h.p2text = 'block#'
AND        h.event IS NOT NULL
AND        f.file_id = h.p1
GROUP BY f.tablespace_type
ORDER BY ash_secs DESC
/
```

Here, we can see we are spending more time on index reads that table reads, and very little on the undo tablespace, so there is not too much work to maintain read consistency occurring.

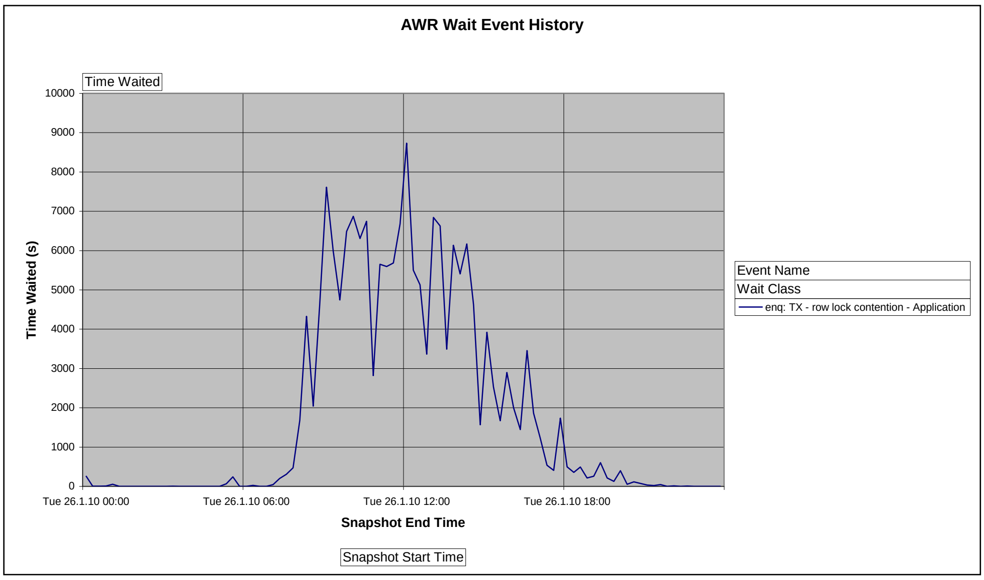
```
TABLES   ASH_SECS
------ ----------
INDEX       30860
TABLE       26970
UNDO         1370
SYSTEM        490
```

Of course, you could classify your tablespaces differently. You might have different applications all in one database. You might want to know how much of the load comes FROM which application.

I suppose you could look go down to each individual object being accessed, but that will be more involved, and I haven’t tried that.

Blocking Lock Analysis

This graph is derived from AWR data[21], and it shows a period of time when a system exhibited a lot of time lost to row level wait. We lost 13 hours of user time in the two-hour period from 11am to 1pm.



Lets take a look at the historical ASH data in the AWR snapshots, and see where we lost time to row level locking in that period across the whole database.

```
SELECT /*+LEADING(x h) USE_NL(h)*/
            h.sql_id
,           h.sql_plan_hash_value
,           SUM(10) ash_secs
FROM        dba_hist_snapshot x
,           dba_hist_active_sess_history h
WHERE       x.end_interval_time   <= TO_DATE('201001261300','yyyymmddhh24mi')
AND         x.begin_interval_time >= TO_DATE('201001261100','yyyymmddhh24mi')
AND         h.sample_time BETWEEN TO_DATE('201001261100','yyyymmddhh24mi')
              AND     TO_DATE('201001261300','yyyymmddhh24mi')
AND         h.snap_id = x.snap_id
AND         h.dbid = x.dbid
AND         h.instance_number = x.instance_number
AND         h.event = 'enq: TX - row lock contention'
GROUP BY h.sql_id, h.sql_plan_hash_value
ORDER BY ash_secs DESC
/
```

[21] This blog extra explains how to produce such a graph: http://blog.go-faster.co.uk/2008/12/graphing-awr-data-in-excel.html

And rather reassuringly the ASH total agrees quite well with AWR. The top statement alone is costing us nearly 5 hours.

```
                SQL Plan
SQL_ID        Hash Value   ASH_SECS
------------- ---------- ----------
7qxdrwcn4yzhh 3723363341      26030
652mx4tffq415 1888029394      11230
c9jjtvk0qf649 3605988889       6090
artqgxug4z0f1    8450529        240
gtj7zuzy2b4g6 2565837323        100
```

Let's look at the statements involved. They all come FROM the PeopleSoft Publish and Subcribe Servers.

The first statement shows a homemade sequence. PeopleSoft is a platform agnostic development, so it doesn't use Oracle sequence objects. The other two statements show an update to a queue management table.

```
SQL_ID 7qxdrwcn4yzhh
--------------------
UPDATE PSIBQUEUEINST SET QUEUESEQID=QUEUESEQID+:1 WHERE QUEUENAME=:2
```

```
SQL_ID 652mx4tffq415
--------------------
UPDATE PSAPMSGPUBSYNC SET LASTUPDDTTM=SYSDATE WHERE QUEUENAME=:1
```

```
SQL_ID c9jjtvk0qf649
--------------------
UPDATE PSAPMSGSUBCSYNC SET LASTUPDDTTM=SYSDATE WHERE QUEUENAME=:1
```

There is nothing I can do about any of these because the code is deep inside PeopleTools and cannot be changed. This is the way that the Integration Broker works.

I cannot find the statement that is blocking these statements. Oracle doesn't hold that information. It is probably another instance of the same statement, but that it isn't the question. The real question is 'what is the session that is holding the lock doing while it is holding the lock, and can I do something about that?'

The ASH data has three columns that help me to identify the blocking session.

- BLOCKING_SESSION_STATUS – this column has the value VALID if the blocking session is within the same instance, but GLOBAL if is in another instance.
- BLOCKING_SESSION – this is the session ID of the blocking session if the session is within the same instance, otherwise it is null.
- BLOCKING_SESSION_SERIAL# - this is the serial number of the blocking session if the session is within the same instance, otherwise it is null.

For cross-instance locking I cannot use ASH in 10g to find the exact session that is holding the lock. All I know is that I am locked by a session connected to another instance. The 11g ASH data does contain this information. So this technique only works for locking within a single instance on 10g.

The queries that I need to write don't perform well on the ASH views, so I am going to extract them to a temporary working storage table.

```
DROP TABLE my_ash
/

CREATE TABLE my_ash AS
SELECT /*+LEADING(x) USE_NL(h)*/  h.*
FROM       dba_hist_snapshot x
,          dba_hist_active_sess_history h
WHERE      x.end_interval_time >= TO_DATE('201001261100','yyyymmddhh24mi')
AND      x.begin_interval_time <= TO_DATE('201001261300','yyyymmddhh24mi')
AND        h.sample_time BETWEEN TO_DATE('201001261100','yyyymmddhh24mi')
                          AND     TO_DATE('201001261300','yyyymmddhh24mi')
AND        h.snap_id = x.snap_id
AND        h.dbid = x.dbid
AND        h.instance_number = x.instance_number
/

CREATE INDEX my_ash ON my_ash (dbid, instance_number, snap_id, sample_id,
sample_time) COMPRESS 3
/
CREATE INDEX my_ash2 ON my_ash (event, dbid, instance_number, snap_id)
COMPRESS 3
/
```

I now want to look for statements running in the sessions that are blocking the sessions that are waiting on TX enqueue.

```
SELECT     /*+LEADING(x w) USE_NL(h w)*/
           h.sql_id
,          h.sql_plan_hash_value
,          SUM(10) ash_secs
FROM       my_ash w
    left outer join my_ash h
    on     h.snap_id = w.snap_id
    AND    h.dbid = w.dbid
    AND    h.instance_number = w.instance_number
    AND    h.sample_id = w.sample_id
    AND    h.sample_time = w.sample_time
    AND    h.session_id = w.blocking_session
    AND    h.session_serial# = w.blocking_session_serial#
WHERE      w.event = 'enq: TX - row lock contention'
GROUP BY h.sql_id, h.sql_plan_hash_value
ORDER BY ash_secs DESC
```

This is the top of list of statements.

Note that two of the statements that appear in this list were the original SQL_IDs that we started with. I'll come back to this below.

```
SQL_ID        SQL_PLAN_HASH_VALUE   ASH_SECS
------------- ------------------- ----------
                                       29210
5st32un4a2y92          2494504609      10670
652mx4tffq415          1888029394       7030
artqgxug4z0f1             8450529        580
7qxdrwcn4yzhh          3723363341        270
```

The first line in the report is blank because there is no ASH data for the session holding the lock because it is not active on the database. This indicates that the client process is busy, or waiting on something else outside the database. This is where the majority of the time is spent, and there is nothing that can be done within the database to address this. It is a matter of looking at the client process.

However the line in the report says that a statement blocks other sessions for 10670 seconds. We can look at that.

```
SELECT * FROM table(dbms_xplan.display_awr('5st32un4a2y92',2494504609,NULL,'ADVANCED'));
```

Note also that this is the execution plan when the query was first seen. The cost is the cost then, not now. The value of the bind variable was the value then not now!

```
SQL_ID 5st32un4a2y92
--------------------
SELECT 'X' FROM PS_CDM_LIST  WHERE CONTENTID = :1

Plan hash value: 2494504609

-------------------------------------------------------------------------------------
| Id  | Operation            | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |              |       |       |    22 (100)|          |
|   1 |  INDEX FAST FULL SCAN| PS_CDM_LIST  |     1 |     5 |    22  (10)| 00:00:01 |
-------------------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1 / PS_CDM_LIST@SEL$1

Peeked Binds (identified by position):
--------------------------------------

   1 - :1 (NUMBER): 17776
```

If I run a fresh execution plan on this statement, the cost is now 3178. This reflects how the table has grown over time.

```
explain plan for SELECT 'X' FROM PS_CDM_LIST WHERE CONTENTID = :1
/

Explained.

Plan hash value: 2494504609

-----------------------------------------------------------------------------------
| Id  | Operation            | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |     1 |     6 |  3178   (9)| 00:00:05 |
|*  1 |  INDEX FAST FULL SCAN| PS_CDM_LIST |     1 |     6 |  3178   (9)| 00:00:05 |
-----------------------------------------------------------------------------------

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

   1 - filter("CONTENTID"=TO_NUMBER(:1))
```

Resolving the Lock Chain to the Ultimate Blocking Session

The second longest running blocking statement is one of the statements that we found in the first place, so this shows that we have a chain of locks, and we need to resolve that back to the blocking statement that is not itself blocked.

```
SELECT * FROM table(dbms_xplan.display_awr('652mx4tffq415',1888029394,NULL,'ADVANCED'));
```

```
SQL_ID 652mx4tffq415
--------------------
UPDATE PSAPMSGPUBSYNC SET LASTUPDDTTM=SYSDATE WHERE QUEUENAME=:1
```

If one session is held by a second session which is itself blocked by a third session, I am more interested in what the third session is doing. The following SQL updates the blocking session data recorded in the first session that indicates the session to point to the third session. I don’t need to find the ASH data for the third session. It might not exist because the third session might not be active on the database (because the user or client process is busy with non-database activity) while it continues to hold the lock.

If I run the SQL repeatedly until no more rows are updated, I will be able to associate the time spent waiting on a lock with the session that is ultimately responsible for the lock.

```
MERGE INTO my_ash u
USING (
SELECT      /*+LEADING(A) USE_NL(B C)*/ a.snap_id, a.dbid, a.instance_number
,           a.sample_id, a.sample_time
,           a.session_id, a.session_serial#
,           b.blocking_session, b.blocking_session_serial#, b.blocking_session_status
FROM        my_ash a
INNER JOIN my_ash b
ON          b.snap_id = a.snap_id
AND         b.dbid = a.dbid
AND         b.instance_number = a.instance_number
AND         b.sample_id = a.sample_id
AND         b.sample_time = a.sample_time
AND         b.session_id = a.blocking_session
AND         b.session_serial# = a.blocking_session_serial#
AND         b.event = 'enq: TX - row lock contention'
AND         b.session_id != a.session_id
AND         b.session_serial# != a.session_serial#
AND         b.blocking_session != a.session_id
AND         b.blocking_session_serial# != a.session_serial#
WHERE       a.event = 'enq: TX - row lock contention'
) s
ON (        u.snap_id = s.snap_id
AND         u.dbid = s.dbid
AND         u.instance_number = s.instance_number
AND         u.sample_id = s.sample_id
AND         u.sample_time = s.sample_time
AND         u.session_id = s.session_id
AND         u.session_serial# = s.session_serial#)
WHEN MATCHED THEN UPDATE
SET u.blocking_session = s.blocking_session
,   u.blocking_session_serial# = s.blocking_session_serial#
,   u.blocking_session_status = s.blocking_session_status
/
```

So this moves the emphasis further onto the query of PS_CDM_LIST.

```
                SQL Plan
SQL_ID        Hash Value   ASH_SECS
------------- ---------- ----------
5st32un4a2y92 2494504609      12840 (was 10670)
652mx4tffq415 1888029394       5030 (was  7030)
7qxdrwcn4yzhh 3723363341        320 (was   270)
```

Which Tables Account for My I/O?

ASH holds object number data. But I want to work in terms of tables. So, I am going to produce my own version of DBA_OBJECTS. I want to be able to easily group all the objects in a table, its indexes, their partitions and sub-partitions

```
CREATE TABLE DMK_OBJECTS
(    OBJECT_ID NUMBER NOT NULL,
     OWNER VARCHAR2(30) NOT NULL,
     OBJECT_NAME VARCHAR2(128) NOT NULL,
     SUBOBJECT_NAME VARCHAR2(30),
     PRIMARY KEY (OBJECT_ID)
/

insert into dmk_objects
SELECT  object_id, owner, object_name, subobject_name
FROM dba_objects
where object_type like 'TABLE%'
union all
SELECT  o.object_id, i.table_owner, i.table_name, o.subobject_name
FROM dba_objects o, dba_indexes i
where o.object_type like 'INDEX%'
and i.owner = o.owner
and i.index_name = o.object_name
/
```

So, for a single process identified by process instance number, I want to take the ash entries for that process that relate to the db file wait events, and I want to see which tables they relate to.

```
SELECT /*+LEADING(r x h) USE_NL(h)*/
           r.prcsinstance
,          o.owner, o.object_name
,          (CAST(r.enddttm AS DATE)-CAST(r.begindttm AS DATE))*86400
exec_secs
,          SUM(10) ash_secs
FROM       dba_hist_snapshot x
,          dba_hist_active_sess_history h
,          sysadm.psprcsrqst r
,          dmk_objects o
WHERE      x.end_interval_time >= r.begindttm
AND        x.begin_interval_time <= r.enddttm
AND        h.sample_time BETWEEN r.begindttm AND r.enddttm
AND        h.snap_id = x.snap_id
AND        h.dbid = x.dbid
AND        h.instance_number = x.instance_number
AND        h.module = r.prcsname
AND        h.action LIKE 'PI='||r.prcsinstance||'%'
AND        h.event LIKE 'db file%'
AND        r.prcsinstance = 2256605
AND        h.current_obj# = o.object_id
GROUP BY   r.prcsinstance, r.prcsname, r.begindttm, r.enddttm
,          o.owner, o.object_name
having SUM(10) >= 60
```

This process spends a lot of time reading GP_RSLT_ACUM.

```
Process                                 Exec     ASH
Instance OWNER    OBJECT_NAME           Secs     Secs
-------- -------- -------------------- ------ -------
 2256605 SYSADM   PS_GP_RSLT_ACUM        5469     590
 2256605 SYSADM   PS_GP_RSLT_PIN         5469     310
 2256605 SYSADM   PS_GP_PYE_PRC_STAT     5469     170
 2256605 SYSADM   PS_JOB                 5469      30
                                              -------
sum                                              1100
```

We can then get the execution plans for the individual statements

```
SELECT 'SELECT * FROM table(dbms_xplan.display_awr('''||sql_id||''','||sql_plan_hash_value||',NULL,''ADVANCED''));'
FROM       (
           SELECT /*+LEADING(r x h) USE_NL(bh)*/
                      r.prcsinstance
           ,          o.owner, o.object_name
           ,          h.sql_id, h.sql_plan_hash_value
           ,          (CAST(r.enddttm AS DATE)-CAST(r.begindttm AS DATE))*86400 exec_secs
           ,          SUM(10) ash_secs
           FROM       dba_hist_snapshot x
           ,          dba_hist_active_sess_history h
```

```
            ,           sysadm.psprcsrqst r
            ,           my_ash_objects o
            WHERE       x.end_interval_time >= r.begindttm
            AND         x.begin_interval_time <= r.enddttm
AND         h.sample_time BETWEEN r.begindttm AND r.enddttm
            AND         h.snap_id = x.snap_id
            AND         h.dbid = x.dbid
            AND         h.instance_number = x.instance_number
            AND         h.module = r.prcsname
            AND         o.object_name = 'PS_GP_RSLT_ACUM'
            AND         h.action LIKE 'PI='||r.prcsinstance||'%'
            AND         h.event LIKE 'db file%'
            AND         r.prcsinstance = 2256605
            AND         h.current_obj# = o.object_id
            GROUP BY r.prcsinstance, r.prcsname, r.begindttm, r.enddttm
            ,           o.owner, o.object_name
            ,           h.sql_id, h.sql_plan_hash_value
--          having SUM(10) >= 60
            ORDER BY ash_secs DESC
            ) x
ORDER BY ash_secs DESC
/
```

```
SELECT * FROM table(dbms_xplan.display_awr('5n5tu62039ak2',843197476,NULL,'ADVANCED'));
SELECT * FROM table(dbms_xplan.display_awr('ggwkkzmw1wmfs',3417552465,NULL,'ADVANCED'));
SELECT * FROM table(dbms_xplan.display_awr('g1yupgb61zndq',3420404643,NULL,'ADVANCED'));
```

This is the beginning of the top statement

```
INSERT INTO … SELECT …
FROM PS_XGF_ABS14_TMP4 A, PS_GP_RSLT_ACUM B, ps_GP_PIN C, ps_gp_pye_prc_stat P,ps_gpgb_ee_rslt G, PS_GP_CALENDAR L
WHERE B.PIN_NUM = C.PIN_NUM AND A.PROCESS_INSTANCE =2256605 AND P.EMPLID = A.EMPLID AND
P.EMPL_RCD = A.EMPL_RCD AND B.ACM_FROM_DT = A.PERIOD_BEGIN_DT AND B.USER_KEY1 > ' '
AND B.USER_KEY1 =to_char(G.HIRE_DT,'YYYY-MM-DD')
AND C.PIN_NM IN ('AE PHO_TAKE', 'AE PHO B_TAKE')
…
```

Across an entire system, for the last week which tables are the cause of the most I/O?

```
SELECT     /*+LEADING(x h) USE_NL(h)*/
           o.owner, o.object_name
,          SUM(10) ash_secs
FROM       dba_hist_snapshot x
,          dba_hist_active_sess_history h
,          dmk_objects o
WHERE      x.end_interval_time   >= SYSDATE-7
AND        x.begin_interval_time <= SYSDATE
AND        h.sample_time         >= SYSDATE-7
AND        h.sample_time         <= SYSDATE
AND        h.snap_id = x.snap_id
AND        h.dbid = x.dbid
AND        h.instance_number = x.instance_number
AND        h.event LIKE 'db file%'
AND        h.current_obj# = o.object_id
group      by o.owner, o.object_name
having     SUM(10) >= 3600
order      by ash_secs desc
```

This is just to put things into context. I am going to look at GP_RSLT_ACUM, because I know it is the output of the payroll calc process, and it may be a case for doing a selective extract into a reporting table.

```
                                     ASH
OWNER    OBJECT_NAME                Secs
-------- -------------------- -------
SYSADM   PS_TL_RPTD_TIME        800510
SYSADM   PS_TL_PAYABLE_TIME     327280
SYSADM   PS_GP_RSLT_ACUM        287870
SYSADM   PS_SCH_DEFN_DTL        161690
SYSADM   PS_SCH_DEFN_TBL        128070
SYSADM   PS_GP_RSLT_PIN         124560
SYSADM   PS_GP_PYE_PRC_STAT      92410
SYSADM   PS_SCH_ADHOC_DTL        88810
…
```

Which processes hit this table?

```
    SELECT /*+LEADING(x) USE_NL(h)*/
           o.owner, o.object_name
    ,      h.module
--  ,      h.sql_id, h.sql_plan_hash_value
    ,      SUM(10) ash_secs
    FROM   dba_hist_snapshot x
    ,      dba_hist_active_sess_history h
    ,      dmk_objects o
    WHERE  x.end_interval_time   >= SYSDATE-7
    AND    x.begin_interval_time <= SYSDATE
    AND    h.sample_time         >= SYSDATE-7
    AND    h.sample_time         <= SYSDATE
    AND    h.snap_id = x.snap_id
    AND    h.dbid = x.dbid
    AND    h.instance_number = x.instance_number
    AND    h.event LIKE 'db file%'
    AND    h.current_obj# = o.object_id
    AND    o.object_name = 'PS_GP_RSLT_ACUM'
--  AND    h.module != 'GPPDPRUN'
--  AND    h.module = 'DBMS_SCHEDULER'
    GROUP BY o.owner, o.object_name
    , h.module
--  , h.sql_id, h.sql_plan_hash_value
    having SUM(10) >= 900
    ORDER BY ash_secs DESC
```

So these processes spend this long reading the accumulator table and its index

```
                                                    ASH
OWNER    OBJECT_NAME          MODULE               Secs
-------- -------------------- ----------------- -------
SYSADM   PS_GP_RSLT_ACUM      XGF_HOL_MGMT        79680
SYSADM   PS_GP_RSLT_ACUM      DBMS_SCHEDULER      37810
SYSADM   PS_GP_RSLT_ACUM      SQL*Plus            37060
SYSADM   PS_GP_RSLT_ACUM      GPGBHLE             30710
SYSADM   PS_GP_RSLT_ACUM      GPPDPRUN            27440
SYSADM   PS_GP_RSLT_ACUM      XGF_AE_AB007        21440
SYSADM   PS_GP_RSLT_ACUM      SQL Developer       11210
SYSADM   PS_GP_RSLT_ACUM      GPGBEPTD             7240
SYSADM   PS_GP_RSLT_ACUM      XGF_CAPITA           5850
SYSADM   PS_GP_RSLT_ACUM      GPGB_PSLIP_X         5030
SYSADM   PS_GP_RSLT_ACUM      GPGB_EDI             4880
```

Who is using this index?

Or, to put it another way, I want to change or drop this index, who and what will I impact?

The challenge is is certainly not exclusive to PeopleSoft, but in PeopleSoft, the Application Designer tool makes it very easy for developers to add indexes to tables. Sometimes, too easy! I often find tables with far more indexes than are good for them.

There are several concerns:

- Indexes are maintained during data modification. The more indexes you have, the greater the overhead.
- If you have too many indexes, Oracle might choose to use the wrong one, resulting in poorer performance.
- There is of course also a space overhead for each index, but this is often of less concern.

If you can get rid of an index, Oracle doesn't store, maintain or use it.

In some cases, I have wanted to remove unnecessary indexes, and in others to adjust indexes. However, this immediately raises the question of where are these indexes used, and who will be impacted by the change. Naturally, I turn to the Active Session History (ASH) to help me find the answers.

As we have already discussed ASH reports the object number, file number, block number and (from 11g) row number being accessed by physical file operations. These values are not reliable for other events because they are merely left over from the previous file event that set them. So, we can profile the amount of time spent on physical I/O on different indexes, but not other forms of DB Time, such as CPU time, spent accessing the blocks in the buffer cache.

However, if you want to find where an index is used, then this query will also identify SQL_IDs where the index is either used in the query or maintained by DML. If I am interested in looking for places where changing or deleting an index could have an impact then I am only interested in SQL query activity. ASH samples which relate to index maintenance are a false positive. Yet, I cannot simply eliminate ASH samples where the SQL_OPNAME is not SELECT because the index may be used in a query within the DML statement.

Another problem with this method is that it matches SQL to ASH by object ID. If someone has rebuilt an index, then its object number changes. A different approach is required.

Index Use from SQL Plans Captured by AWR

During an AWR snapshot the top-n SQL statements by each SQL criteria in the AWR report (Elapsed Time, CPU Time, Parse Calls, Shareable Memory, Version Count) , see [dbms_workload_repository](). The SQL plans are exposed by the view [DBA_HIST_SQL_PLAN]().

On PeopleSoft systems, I generally recommend decreasing the snapshot interval from the default of 60 minutes to 15. The main reason is that SQL gets aged out of the library cache very quickly in PeopleSoft systems. They generate lots of dynamic code, often with literal values rather than bind variables. Cursor sharing is not recommended for PeopleSoft, so different bind variables result in different SQL_IDs. The dynamic code also results in different SQL IDs even with cursor sharing (see http://blog.psftdba.com/2014/08/to-hint-or-not-to-hint-application.html). Therefore, increasing the snapshot frequency means that will capture more SQL statements and plans. This will increase total volume of the AWR repository simply because there are more snapshots. However, the overall volume of ASH data captured does not change, it just gets copied to the repository earlier.

On DBA_HIST_SQL_PLAN the object ID, owner, type and name are recorded, so I can find the plans which referenced a particular object. I am going to take an example from a PeopleSoft Financials system, and look at indexes on the PS_PROJ_RESOURCE table. These are some of the indexes on PS_PROJ_RESOURCE. We have 4 indexes that all lead on PROCESS_INSTANCE. I suspect that not all are essential, but I need to work out what is using them.

```
                          Col
INDEX_NAME                Pos COLUMN_NAME          COLUMN_EXPRESSION
------------------ ---------- -------------------- -----------------
…
PSJPROJ_RESOURCE            1 PROCESS_INSTANCE
                            2 BUSINESS_UNIT_GL
                            3 BUSINESS_UNIT
                            4 PROJECT_ID
                            5 ACTIVITY_ID
                            6 CUST_ID

PSLPROJ_RESOURCE            1 PROCESS_INSTANCE
                            2 EMPLID
                            3 EMPL_RCD
                            4 TRANS_DT

PSMPROJ_RESOURCE            1 PROCESS_INSTANCE
                            2 BUSINESS_UNIT
                            3 PROJECT_ID
                            4 ACTIVITY_ID
                            5 RESOURCE_ID

PSNPROJ_RESOURCE            1 PROCESS_INSTANCE
                            2 BUSINESS_UNIT
                            3 TIME_RPTG_CD
…
```

I find it easier to extract the ASH data to my own working storage table. For each index on PS_PROJ_RESOURCE, I am going to extract a distinct list of plan hash values. I will then extract all ASH data for those plans.

Note, that I have not joined the SQL_ID on DBA_HIST_SQL_PLAN. That is because different SQL_IDs can produce the same execution plan. The plan is equally valid for all SQL_IDs that produce the plan, not just the one where the SQL_ID also matches. Although, of course, costs may vary.

```
DROP TABLE my_ash purge
/
CREATE TABLE my_ash COMPRESS AS
WITH p AS (
         SELECT DISTINCT p.plan_hash_value, p.object#, p.object_owner, p.object_type, p.object_name
         FROM      dba_hist_sql_plan p
         WHERE     p.object_name like 'PS_PROJ_RESOURCE'
         AND      p.object_type LIKE 'INDEX%'
         AND      p.object_owner = 'SYSADM'
         )
SELECT p.object# object_id, p.object_owner, p.object_type, p.object_name
,      h.*
FROM   dba_hist_active_sess_history h
,          p
WHERE  h.sql_plan_hash_value = p.plan_hash_value
/
```

I am fortunate that PeopleSoft is a well instrumented application. Module and Action are set to fairly sensible values that will tell me whereabouts in the application the ASH sample relates.

In the following query I have omitted any ASH data generated by SQL*Plus, Toad, or SQL Developer, and also any generated by Oracle processes to prevent statistics collection being included.

```
Set pages 999 lines 150 trimspool on
break on object_name skip 1
compute sum of ash_secs on object_name
column ash_secs heading 'ASH|Secs' format 9999999
column module format a20
column action format a32
column object_name format a18
column max_sample_time format a19 heading 'Last|Sample'
column sql_plans heading 'SQL|Plans' format 9999
column sql_execs heading 'SQL|Execs' format 99999
WITH h AS (
        SELECT         object_name
        ,       CASE WHEN h.module IS NULL THEN
REGEXP_SUBSTR(h.program,'[^.@]+',1,1)
                       WHEN h.module LIKE 'PSAE.%' THEN
REGEXP_SUBSTR(h.module,'[^.]+',1,2)
                         ELSE REGEXP_SUBSTR(h.program,'[^.@]+',1,1)
               END as module
        ,       CASE WHEN h.action LIKE 'PI=%' THEN NULL
                    ELSE h.action
                    END as action
        ,       CAST(sample_time AS DATE) sample_time
        ,       sql_id, sql_plan_hash_value, sql_exec_id
        FROM    my_ash h
)
SELECT object_name, module, action
,      sum(10) ash_secs
,      COUNT(DISTINCT sql_plan_hash_value) sql_plans
,      COUNT(DISTINCT sql_id||sql_plan_hash_value||sql_exec_id) sql_execs
,      MAX(sample_time) max_sample_time
FROM   h
WHERE NOT lower(module) IN('oracle','toad','sqlplus','sqlplusw')
AND       NOT lower(module) LIKE 'sql%'
GROUP BY object_name, module, action
ORDER BY SUBSTR(object_name,4), object_name, ash_Secs desc
/
Spool off
```

I now have a profile of how much each index is used. In this particular case, I found something using every index. It is possible that you will not find anything that uses some indexes.

```
                                                                             ASH   SQL    SQL Last
OBJECT_NAME        MODULE               ACTION                              Secs Plans  Execs Sample
------------------ -------------------- -------------------------------- ------- ----- ------ -------------------
…
PSJPROJ_RESOURCE   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step24.S        7300     1     66 06:32:57 27/08/2014
                   PC_PRICING           GF_PBINT_AE.CallmeA.Step24.S          40     1      2 08:38:57 22/08/2014
******************                                                       -------
sum                                                                         7340

PSLPROJ_RESOURCE   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step28.S        1220     1     53 06:33:17 27/08/2014
******************                                                       -------
sum                                                                         1220

PSMPROJ_RESOURCE   PC_TL_TO_PC          GF_PBINT_AE.XxBiEDM.Step07.S          60     2      6 18:35:18 20/08/2014
******************                                                       -------
sum                                                                           60

PSNPROJ_RESOURCE   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step26.S        6720     1     49 18:53:58 26/08/2014
                   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step30.S        3460     1     60 06:33:27 27/08/2014
                   GF_OA_CMSN           GF_OA_CMSN.01INIT.Step01.S          2660     1     47 19:19:40 26/08/2014
                   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step06.S        1800     1     52 18:53:28 26/08/2014
                   PC_TL_TO_PC          GF_PBINT_AE.CallmeG.Step01.S        1740     1     61 06:34:17 27/08/2014
                   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step02.S        1680     1     24 18:53:18 26/08/2014
                   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step10.S        1460     1     33 17:26:26 22/08/2014
                   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step08.S         920     1     26 17:26:16 22/08/2014
                   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step36.S         460     1     18 18:26:38 20/08/2014
                   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Step09.S         420     1     16 06:33:07 27/08/2014
                   PC_PRICING           GF_PBINT_AE.CallmeG.Step01.S         200     1     10 08:09:55 22/08/2014
                   PC_AP_TO_PC          GF_PBINT_AE.CallmeH.Step00A.S        170     1     17 21:53:26 21/08/2014
                   PC_PRICING           GF_PBINT_AE.CallmeA.Step36.S          20     1      1 08:02:46 05/08/2014
                   PC_PRICING           GF_PBINT_AE.CallmeA.Step30.S          20     1      1 13:42:48 04/08/2014
                   PC_PRICING           GF_PBINT_AE.CallmeA.Step06.S          20     1      1 15:58:35 28/07/2014
                   PC_TL_TO_PC          GF_PBINT_AE.CallmeA.Pseudo.S          20     1      1 19:45:11 06/08/2014
******************                                                       -------
sum                                                                        21770
…
```

The next stage is to look at individual SQL statements

This query looks for which SQL statement is using a particular index on PROJ_RESOURCE. If I can't find the SQL which cost the most time, then just choose another SQL with the same plan

- I have found that sometimes a plan is captured by AWR, but the SQL statement is not. Personally, I think that is a bug. Working around it has made the following query quite complicated.

```
Break on object_name skip 1
column ash_secs heading 'ASH|Secs' format 9999999
Set long 50000
Column cmd Format a200
Spool dmk
```

```
WITH h AS (
            SELECT  h.object_name
            ,       CASE WHEN h.module IS NULL THEN REGEXP_SUBSTR(h.program,'[^.@]+',1,1)
                         WHEN h.module LIKE 'PSAE.%' THEN REGEXP_SUBSTR(h.module,'[^.]+',1,2)
                         ELSE REGEXP_SUBSTR(h.program,'[^.@]+',1,1)
                    END as module
            ,       CASE WHEN h.action LIKE 'PI=%' THEN NULL
                         ELSE h.action
                    END as action
            ,       h.sql_id, h.sql_plan_hash_value
            ,       t.command_type --not null if plan and statement captured
            FROM    my_ash h
                        LEFT OUTER JOIN (
                                    SELECT t1.*
                                    FROM dba_hist_sqltext t1
                                    ,    dba_hist_sql_plan p1
                                    WHERE t1.sql_id = p1.sql_id
                                    AND p1.id = 1
                                    ) t
                        ON   t.sql_id = h.sql_id
                        AND  t.dbid = h.dbid
            WHERE   h.object_name IN('PSMPROJ_RESOURCE')
            AND     h.object_Type = 'INDEX'
            AND     h.object_owner = 'SYSADM'
            And     NOT lower(module) IN('oracle','toad','sqlplus','sqlplusw')
            AND     NOT lower(module) LIKE 'sql%'
), x AS ( --aggregate DB time by object and statement
SELECT    object_name, sql_id, sql_plan_hash_value
,           sum(10) ash_secs
,           10*COUNT(command_type) sql_secs  --DB time for captured statements only
FROM      h
WHERE NOT lower(module) IN('oracle','toad','sqlplus','sqlplusw')
AND       NOT lower(module) LIKE 'sql%'
GROUP BY  object_name, sql_id, sql_plan_hash_value
), y AS ( --rank DB time per object and plan
SELECT      object_name, sql_id, sql_plan_hash_value
,           ash_secs
,           SUM(ash_secs) over (partition by object_name, sql_plan_hash_value) plan_ash_secs
,           row_number() over (partition by object_name, sql_plan_hash_value ORDER BY sql_Secs DESC) ranking
FROM        x
), z AS (
SELECT object_name
, CASE WHEN t.sql_text IS NOT NULL THEN y.sql_id
       ELSE (SELECT t1.sql_id
             FROM   dba_hist_sqltext t1
             ,      dba_hist_sql_plan p1
             WHERE  t1.sql_id = p1.sql_id
             AND    p1.plan_hash_value = y.sql_plan_hash_value
             AND    rownum = 1) --if still cannot find statement just pick any one
  END AS sql_id
, y.sql_plan_hash_value, y.plan_ash_secs
, CASE WHEN t.sql_text IS NOT NULL THEN t.sql_text
       ELSE (SELECT t1.sql_Text
             FROM   dba_hist_sqltext t1
             ,      dba_hist_sql_plan p1
             WHERE  t1.sql_id = p1.sql_id
             AND    p1.plan_hash_value = y.sql_plan_hash_value
             AND    rownum = 1) --if still cannot find statement just pick any one
  END AS sql_text
from y
 left outer join dba_hist_sqltext t
 on t.sql_id = y.sql_id
WHERE ranking = 1 --captured statement with most time
)
SELECT *
--'SELECT * FROM
table(dbms_xplan.display_awr('''||sql_id||''','||sql_plan_hash_value||',NULL,''ADVANCED''))/*'||object_name||':'||plan_ash_Secs||'*/;' cmd
FROM z
ORDER BY object_name, plan_ash_secs DESC
/
Spool off
```

So now I can see the individual SQL statements.

```
PSJPROJ_RESOURCE   f02k23bqj0xc4        3393167302         7340 UPDATE PS_PROJ_RESOURCE C SET (C.Operating_Unit, C.CHARTFIELD1, C.PRODUCT, C.CLA
                                                                SS_FLD, C.CHARTFIELD2, C.VENDOR_ID, C.contract_num, C.contract_line_num, …

PSLPROJ_RESOURCE   2fz0gcb2774y0         821236869         1220 UPDATE ps_proj_resource p SET p.deptid = NVL (( SELECT j.deptid FROM ps_job j WH
                                                                ERE j.emplid = p.emplid AND j.empl_rcd = p.empl_rcd AND j.effdt = ( SELECT MAX (…

PSMPROJ_RESOURCE   96cdkb7jyq863         338292674           50 UPDATE PS_GF_BI_EDM_TA04 a SET a.GF_ni_amount = ( SELECT x.resource_amount FROM
                                                                PS_PROJ_RESOURCE x WHERE x.process_instance = …

                   1kq9rfy8sb8d4        4135884683           10 UPDATE PS_GF_BI_EDM_TA04 a SET a.GF_ni_amount = ( SELECT x.resource_amount FROM
                                                                PS_PROJ_RESOURCE x WHERE x.process_instance = …

PSNPROJ_RESOURCE   ga2x2u4jw9p0x        2282068749         6760 UPDATE PS_PROJ_RESOURCE P SET (P.RESOURCE_TYPE, P.RESOURCE_SUB_CAT) = …

                   9z5qsq6wrr7zp        3665912247         3500 UPDATE PS_PROJ_RESOURCE P SET P.TIME_SHEET_ID = …
```

Ultimately, I have needed to look through the SQL plans that do use an index to decide whether I need to keep that index, or to decide whether the statement would perform adequately using another index. In this case, on this particular system, I think the index PSMPROJ_RESOURCE would be adequate for this statement, and I would consider dropping PSLPROJ_RESOURCE.

The decision also requires some background knowledge about the system. I carried on with examination of SQL and execution plan to determine whether each index is really needed or another index (or even no index at all) would do as well.

Getting Rid of Indexes

So, I am going to jump forward to the point where I have decided that I want drop the J, L and N indexes on PROJ_RESOURCE and just keep M. Obviously this needs to be tested carefully in all the places that reference the index.

- If all the testing is successful and you decide to go ahead and drop the index in production, you might prefer to make it invisible first for a while. It is likely that the indexes you choose to examine are large and will take time to rebuild. An invisible index will not be used by the Optimizer, but it will continue to be maintained during DML. If there are any unfortunate consequences, you can immediately make the index visible without having to rebuild it.

Limitations of Method

- AWR does not capture all SQLs, nor all SQL plans. First the SQL has to be in the library cache and then it must be one of the top-n. A SQL that is efficient because it uses an appropriate index may not be captured, and will not be detected by this approach.
- ASH data is purged after a period of time, by default 31 days. If an index is only used by a process that has not run within the retention period, then it will not be detected by this approach[22]. This is another reason to retain ASH and AWR in a repository for a longer period. I have heard 400 days suggested, so that you have ASH for a year and a month.
  - However, this also causes the SYSAUX tablespace to be become very large, so I would suggest regularly moving the data to a separate database. I know one customer who has built a central AWR repository for all their production and test databases and automated transfer of data. This repository has been of immense diagnostic value.

[22] However, if you only need an index during an annual process, perhaps it would be better to build it for that process and drop it again afterwards, rather than have it in place for the whole year?

Did my Execution Plan Change?

We were experiencing a problem with a query in a particular report. We fixed it by adding a hint. I wanted to prove that when the hint was put into production, the execution plan changed. This query is very similar to the one described in Batch Processes (see page 15), but here I want to list all the queries run by all instances of a named report, and see if the exection plan changed.

```
SELECT /*+LEADING(r f d x h) USE_NL(h)*/
            r.prcsinstance
,           r.begindttm
,           h.sql_id
--,         h.sql_child_number
,           h.sql_plan_hash_value
,           (CAST(r.enddttm AS DATE)-CAST(r.begindttm AS DATE))*86400 exec_secs
,           SUM(10)g ash_secs
FROM        dba_hist_snapshot x
,           dba_hist_active_sess_history h
,           sysadm.psprcsrqst r
,       sysadm.ps_cdm_file_list f
,       sysadm.psxprptdefn d
WHERE       x.end_interval_time >= r.begindttm
AND     x.begin_interval_time <=r.enddttm
AND         h.sample_time BETWEEN r.begindttm AND r.enddttm
AND         h.snap_id = x.snap_id
AND         h.dbid = x.dbid
AND         h.instance_number = x.instance_number
AND         h.module = r.prcsname
AND         h.action LIKE 'PI='||r.prcsinstance||'%'
AND     r.prcsinstance = f.prcsinstance
AND     NOT f.cdm_file_type IN('AET','TRC','LOG')
AND     d.report_defn_id = SUBSTR(f.filename,1,instr(f.filename,'.')-1)
AND     d.report_defn_id = 'XGF_WK_LATE'
AND         r.prcsname = 'PSXPQRYRPT'
AND         r.begindttm >= TRUNC(SYSDATE)
ORDER BY begindttm
```

And we can see that after the fix was applied and the users were told they could start to run this report again, the execution plan changed and the run time was much better.

```
PRCSINSTANCE BEGINDTTM           SQL_ID        SQL_PLAN_HASH_VALUE  EXEC_SECS   ASH_SECS
------------ ------------------- ------------- ------------------- ---------- ----------
     1964975 08:30:52 22/01/2010 46smbgcfcrb8d          2602481067      20379      20080
     1965250 09:08:51 22/01/2010 fpftdx2405zyq          2602481067      20983      20690
     1968443 16:42:51 22/01/2010 3rxad5z3ccusv          3398716340        105         80
     1968469 16:47:21 22/01/2010 3rxad5z3ccusv          3398716340         90         70
     1968485 16:50:19 22/01/2010 3rxad5z3ccusv          3398716340         62         40
     1968698 17:40:01 22/01/2010 0ku8f514k3nt0          3398716340         76         50
     1968866 18:19:19 22/01/2010 cbmyvpsxzyf5n          3398716340        139        120
     1968966 18:34:24 22/01/2010 5jb1sgmjc7436          3398716340        187        170
```

So, not only have I diagnosed a problem with ASH, I have also proven that the fix, when applied to production has successfully resolved the issue.

What was the Effect of Plan Stability

I have experienced unstable execution plans with processing of Payroll calculations. The performance of the larger pay group is fine, but some of the execution plans for the smaller paygroups are different, and performance can be poor.

A set of stored outlines were created for a full payroll identification and calculation process for the larger payroll, and applied to all subsequent payrolls. Now, I want to prove not only that the outlines were used, but that they have a beneficial effect.

I have three test scenarios.

1. A large streamed payroll calculation was run. It ran without using outlines for 2h 42m, which can considered to be good performance (in fact I used this process to collect the stored outlines).

2. A small non-streamed payroll calculation without outlines. This ran for over 8 hours before it was cancelled. Hence, I don't have data for all statements for this scenario.

3. A small non-streamed payroll calculation again, but this time with outlines enabled. It ran for 2h5m. Not great, considering it has a lot fewer payees than a single stream of the large payroll, but better than scenario 2.

I can use the ASH data to see whether the execution plan changed, and what effect that had on performance.

The SQL to perform the comparison looks horrendous, but it is effectively the usual query for each test scenario in in-line views that are then joined together.

```
set pages 40
column sql_plan_hash_value  heading 'sql_plan_hash_value' format 999999999999
column sql_plan_hash_value2 heading 'sql_plan_hash_value' format a12
SELECT /*+ LEADING(@q1 r1@q1 x1@q1 h1@q1) USE_NL(h1@q1)
           LEADING(@q2 r2@q2 x2@q2 h2@q2) USE_NL(h2@q2)
           LEADING(@q3 r3@q3 x3@q3 h3@q3) USE_NL(h3@q3) */
         q1.sql_id
,        q1.sql_plan_hash_value, q1.ash_secs
,        DECODE(q1.sql_plan_hash_value,q2.sql_plan_hash_value,'**SAME**',
                                       q2.sql_plan_hash_value) sql_plan_hash_value2
,        q2.ash_secs
,        DECODE(q1.sql_plan_hash_value,q3.sql_plan_hash_value,'**SAME**',
                                       q3.sql_plan_hash_value) sql_plan_hash_value2
,        q3.ash_secs
FROM     (
         SELECT   /*+qb_name(q1)*/
                  h1.sql_id
         ,        h1.sql_plan_hash_value
         ,        (NVL(r1.enddttm,SYSDATE)-r1.begindttm)*86400 exec_secs
         ,        SUM(10) ash_secs
         FROM     dba_hist_snapshot x1
         ,        dba_hist_active_sess_history h1
         ,        sysadm.psprcsrqst r1
         WHERE    x1.end_interval_time >= r1.begindttm
         AND      x1.begin_interval_time <= NVL(r1.enddttm,SYSDATE)
         AND      h1.sample_time BETWEEN r1.begindttm AND NVL(r1.enddttm,SYSDATE)
         AND      h1.Snap_id = x1.Snap_id
         AND      h1.dbid = x1.dbid
         AND      h1.instance_number = x1.instance_number
         AND      h1.module  like r1.prcsname
         AND      h1.action LIKE 'PI='||r1.prcsinstance||'%'
         AND      r1.prcsname = 'GPPDPRUN'
         AND      r1.prcsinstance = 2524397
         GROUP BY r1.prcsname, r1.begindttm, r1.enddttm, h1.sql_id, h1.sql_plan_hash_value
         ) Q1
INNER JOIN  (
         SELECT   /*+qb_name(q2)*/
                  h2.sql_id
         ,        h2.sql_plan_hash_value
         ,        (NVL(r2.enddttm,SYSDATE)-r2.begindttm)*86400 exec_secs
         ,        SUM(10) ash_secs
         FROM     dba_hist_snapshot x2
         ,        dba_hist_active_sess_history h2
         ,        sysadm.psprcsrqst r2
         WHERE    x2.end_interval_time >= r2.begindttm
         AND      x2.begin_interval_time <= NVL(r2.enddttm,SYSDATE)
         AND      h2.sample_time BETWEEN r2.begindttm AND NVL(r2.enddttm,SYSDATE)
         AND      h2.Snap_id = x2.Snap_id
         AND      h2.dbid = x2.dbid
         AND      h2.instance_number = x2.instance_number
         AND      h2.module  like r2.prcsname
         AND      h2.action LIKE 'PI='||r2.prcsinstance||'%'
         AND      r2.prcsname = 'GPPDPRUN'
         AND      r2.prcsinstance =  2524456
         GROUP BY r2.prcsname, r2.begindttm, r2.enddttm, h2.sql_id, h2.sql_plan_hash_value
         ) Q2
ON q1.sql_id = q2.sql_id
INNER JOIN  (
         SELECT  /*+qb_name(q3)
                  h3.sql_id
         ,        h3.sql_plan_hash_value
         ,        (NVL(r3.enddttm,SYSDATE)-r3.begindttm)*86400 exec_secs
         ,        SUM(1) ash_secs
         FROM     v$active_Session_history h3[23]
         ,        sysadm.psprcsrqst r3
         WHERE    h3.sample_time BETWEEN r3.begindttm AND NVL(r3.enddttm,SYSDATE)
         AND      h3.module  like r3.prcsname
         AND      h3.action LIKE 'PI='||r3.prcsinstance||'%'
         AND      r3.prcsname = 'GPPDPRUN'
         AND      r3.prcsinstance =  2524456
         GROUP BY r3.prcsname, r3.begindttm, r3.enddttm, h3.sql_id, h3.sql_plan_hash_value
         ) Q3
ON q1.sql_id = q3.sql_id
order by q3.ash_secs desc, q1.sql_id
/
```

```
SQL_ID        SCENARIO 1            ASH_SECS SCENARIO 2     ASH_SECS SCENARIO 3     ASH_SECS
```

[23] This query was run soon after test scenario 3 was run so it uses *v$active_session_history*.

```
------------- ------------------- ---------- ------------ ---------- ------------ ----------
4uzmzh74rdrnz          2514155560        280 3829487612        28750 **SAME**           5023[24]
4n482cm7r9qyn          1595742310        680 869376931           140 **SAME**            889[25]
2f66y2u54ru1v          1145975676        630                         **SAME**            531
1n2dfvb3jrn2m          1293172177        150                         **SAME**            150
652y9682bqqvp          3325291917         30                         **SAME**            110
d8gxmqp2zydta          1716202706         10 678016679            10 **SAME**             32
2np47twhd5nga          3496258537         10                         **SAME**             27
4ru0618dswz3y[26]      2621940820         10                         539127764            22
4ru0618dswz3y           539127764        100                         **SAME**             22
4ru0618dswz3y          3325291917         10                         539127764            22
4ru0618dswz3y          1403673054        110                         539127764            22
gnnu2hfkjm2yd          1559321680         80                         **SAME**             19
fxz4z38pybu3x          1478656524         30                         4036143672           18
2xkjjwvmyf99c          1393004311         20                         **SAME**             18
a05wrd51zy3kj          2641254321         10                         **SAME**             15
```

[24] On the small payroll calculation, without outlines, this statement move than 100 times longer. It had not completed by this stage – the process was cancelled. With outlines enabled this statement used the same execution plan as in scenario 1. It didn't perform that well compared to the large payroll calculation; clearly more work is required for this statement. However, at least it did complete and it did result in improved performance for the small payroll.

[25] This is an example of a statement that performed better on the small payroll without an outline. So, sometimes it is better to let the optimiser change the plan!

[26] This statement executed with 4 different execution plans during the large payroll, but once the outline was applied only one was used, and this seems to be

Which line in the Execution Plan?

Again from 11g, the line in the execution plan is recorded in the ASH data in SQL_PLAN_LINE_ID. I can also group the ASH data by this column and determine not just which statement consumes the most time, but which operation in the exection plan for that statement is consuming the time. I usually do this for one SQL statement at a time.

```
select /*+leading(r x h) use_nl(h)*/
  r.prcsinstance, H.SQL_plan_hash_value, h.sql_plan_line_id
, sum(10) ash_secs
from DBA_HIST_SNAPSHOT x
, DBA_HIST_ACTIVE_SESS_HISTORY h
, sysadm.psprcsrqst r
WHERE X.END_INTERVAL_TIME >= r.begindttm
AND X.BEGIN_INTERVAL_TIME <= NVL(r.enddttm,SYSDATE)
And h.sample_time between r.begindttm AND NVL(r.enddttm,SYSDATE)
and h.SNAP_id = X.SNAP_id
and h.dbid = x.dbid
and h.instance_number = x.instance_number
and h.module  = r.prcsname
and h.action LIKE 'PI='||r.prcsinstance||'%'
And r.begindttm >= TRUNC(SYSDATE)
and r.prcsname = 'CM_CSTACCTG'
and h.sql_id = 'a47fb0x1b23jn'
group by H.SQL_plan_hash_value, r.prcsinstance, h.sql_plan_line_id
ORDER BY prcsinstance, ASH_SECS DESC
```

I now have a profile of a single SQL statement by plan line number.

```
PRCSINSTANCE SQL_PLAN_HASH_VALUE SQL_PLAN_LINE_ID   ASH_SECS
------------ ------------------- ---------------- ----------
     4945802           483167840               25       2410
                       483167840               24       1190
                       483167840               26        210
                       483167840               20        190
                       483167840               21         30
                       483167840               16         20
                       483167840               23         10
                       483167840               22         10
                       483167840               18         10
                       483167840                          10
                       483167840                7         10
```

The plan line IDs can be related back to the execution plan.

```
Plan hash value: 483167840

---------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name               | Rows  | Bytes | Cost (%CPU)| Time     |    TQ  |IN-OUT| PQ Distrib |
---------------------------------------------------------------------------------------------------------------------------------------
…
|  14 |         NESTED LOOPS                |                    |       |       |            |          |  Q1,04 | PCWP |            |
|  15 |          NESTED LOOPS               |                    |  3988 |   669K|  113K   (1)| 00:06:08 |  Q1,04 | PCWP |            |
|  16 |           HASH JOIN SEMI            |                    |  3851 |   481K|  112K   (1)| 00:06:05 |  Q1,04 | PCWP |            |
|  17 |            PX RECEIVE               |                    |  3771K|   233M| 61175   (1)| 00:03:19 |  Q1,04 | PCWP |            |
|  18 |             PX SEND HASH            | :TQ10003           |  3771K|   233M| 61175   (1)| 00:03:19 |  Q1,03 | P->P | HASH       |
|  19 |              PX BLOCK ITERATOR      |                    |  3771K|   233M| 61175   (1)| 00:03:19 |  Q1,03 | PCWC |            |
|  20 |               TABLE ACCESS FULL     | PS_CM_DEPLETE      |  3771K|   233M| 61175   (1)| 00:03:19 |  Q1,03 | PCWP |            |
|  21 |            BUFFER SORT              |                    |       |       |            |          |  Q1,04 | PCWC |            |
|  22 |             PX RECEIVE              |                    |  6058K|   364M| 50906   (1)| 00:02:46 |  Q1,04 | PCWP |            |
|  23 |              PX SEND HASH           | :TQ10001           |  6058K|   364M| 50906   (1)| 00:02:46 |        | S->P | HASH       |
|  24 |               INDEX FULL SCAN       | PS_CM_DEPLETE_COST |  6058K|   364M| 50906   (1)| 00:02:46 |        |      |            |
|  25 |           INDEX UNIQUE SCAN         | PS_TRANSACTION_INV |     1 |       |     1   (0)| 00:00:01 |  Q1,04 | PCWP |            |
|  26 |          TABLE ACCESS BY INDEX ROWID| PS_TRANSACTION_INV |     1 |    44 |     1   (0)| 00:00:01 |  Q1,04 | PCWP |            |
…
---------------------------------------------------------------------------------------------------------------------------------------
```

Recursive SQL

Sometimes a SQL statement causes another SQL statement to run behind the scenes. During SQL parse, Oracle may issue SQL to retrieve information from the catalogue that is usually refered to as 'recursive SQL'. Other examples include SQL that is executed within a trigger, or within a PL/SQL procedure.

From Oracle 11gR2, there is a new column in the ASH data; TOP_LEVEL_SQL_ID. This is the ID of the SQL statement that spawned the recursive SQL.

```
Select * From (
select /*+leading(r x h) use_nl(h)*/
  r.prcsinstance
, h.top_level_sql_id
, h.sql_id, h.sql_plan_hash_value
, (r.enddttm-r.begindttm)*86400 exec_secs
, COUNT(DISTINCT sql_exec_id) num_execs
, SUM(10) ash_secs
, 10*COUNT(DISTINCT sample_id) elap_secs
, COUNT(DISTINCT r.prcsinstance) PIs
from DBA_HIST_SNAPSHOT x
, DBA_HIST_ACTIVE_SESS_HISTORY h
, sysadm.psprcsrqst r
WHERE X.END_INTERVAL_TIME >= r.begindttm
AND X.BEGIN_INTERVAL_TIME <= NVL(r.enddttm,SYSDATE)
And h.sample_time between r.begindttm AND NVL(r.enddttm,SYSDATE)
and h.SNAP_id = X.SNAP_id
and h.dbid = x.dbid
and h.instance_number = x.instance_number
and h.module = r.prcsname
and h.action LIKE 'PI='||r.prcsinstance||'%'
and r.prcsinstance = 4604485
and h.top_level_sql_id = 'bvnq31hbmpzzy'
group by r.prcsinstance, r.prcsname, r.begindttm, r.enddttm
, h.top_level_sql_id
, h.sql_id, h.sql_plan_hash_value
ORDER BY ASH_SECS DESC
) order by ash_secs desc
/
```

Here we can see that two recursive statements were spawned by *bvnq31hbmpzzy*, and most of the time was spent in them.

```
PRCSINSTANCE TOP_LEVEL_SQL SQL_ID        SQL_PLAN_HASH_VALUE  EXEC_SECS  NUM_EXECS   ASH_SECS  ELAP_SECS        PIS
------------ ------------- ------------- ------------------- ---------- ---------- ---------- ---------- ----------
     4604485 bvnq31hbmpzzy 35cpmm408n5qj          1757521524       1069         79        790        790          1
     4604485 bvnq31hbmpzzy bvnq31hbmpzzy          1757521524       1069          1         70         70          1
```

In this example *35cpmm408n5qj* is an insert statement that is issued by a PL/SQL block. I can tell that because the bind variable numbr is prefixed with a 'B'.

```
select sql_id, sql_text
from dbA_hist_sqltext
where sql_id = '35cpmm408n5qj'

SQL_ID        SQL_TEXT
------------- --------------------------------------------------------------------------------
35cpmm408n5qj INSERT INTO PS_GHG_A_BI_CITM VALUES ( :B34 , SYSDATE , :B33 , :B1 , :B2 , :B3 ,
              :B4 , :B5 , :B6 , :B7 , :B8 , :B9 , :B10 , :B11 , :B12 , :B13 , :B14 , :B15 , :B
              16 , :B17 , :B18 , :B19 , :B20 , :B21 , :B22 , :B23 , :B24 , :B25 , :B26 , :B27
              , :B28 , :B29 , :B30 , :B31 , :B32 )
```

In fact, the insert statement comes from a standard PeopleSoft auditing trigger that is executed for each row processed on the original table. We can only count 79 executions because there are only 79 rows of data, the Application Engine trace shows that over 100,000 rows were updated on the table with the trigger.

Top SQL ID can also simply refer to the originating PL/SQL call.

Temporary Space Overhead

From 11gR2, ASH data includes information about memory utilisation in a column called TEMP_SPACE_ALLOCATED. Let me give you a real life practical example.

A Financials customer runs four concurrent instances of the cost accounting process. Two of them complete successfully, but two fail regularly with ORA-1652: Unable to extend temp segment … but complete successfully when run in isolation. The question is what is consuming the temporary tablespace and why.

```
Select * From (
 select /*+leading(r x h) use_nl(h)*/
  r.prcsinstance
, h.sql_id, h.sql_plan_hash_value
, (r.enddttm-r.begindttm)*86400 exec_secs
, count(distinct sql_exec_id) num_execs
, sum(10) ash_secs
, 10*count(distinct sample_id) elap_secs
, round(max(temp_space_Allocated)/1024/1024,0) tempMb
from DBA_HIST_SNAPSHOT x
, DBA_HIST_ACTIVE_SESS_HISTORY h
, sysadm.psprcsrqst r
WHERE X.END_INTERVAL_TIME >= r.begindttm
AND X.BEGIN_INTERVAL_TIME <= NVL(r.enddttm,SYSDATE)
And h.sample_time between r.begindttm AND NVL(r.enddttm,SYSDATE)
and h.SNAP_id = X.SNAP_id
and h.dbid = x.dbid
and h.instance_number = x.instance_number
and h.module  = r.prcsname
and h.action LIKE 'PI='||r.prcsinstance||'%'
And r.begindttm >= TRUNC(SYSDATE)
and r.prcsname = 'CM_CSTACCTG'
group by r.prcsinstance, r.prcsname, r.begindttm, r.enddttm
, h.sql_id, h.sql_plan_hash_value
having sum(10) > (NVL(r.enddttm,SYSDATE)-r.begindttm)*86400/100*5 --5%
ORDER BY ASH_SECS DESC
) order by ash_secs desc
/
```

This report shows the maximum temporary segment consumption of each SQL statement in each process. With a temporary tablespace of 300Gb it is easy to see why 2 processes doing this is enough to cause trouble.

```
PRCSINSTANCE SQL_ID        SQL_PLAN_HASH_VALUE  EXEC_SECS  NUM_EXECS   ASH_SECS  ELAP_SECS     TEMPMB
------------ ------------- ------------------- ---------- ---------- ---------- ---------- ----------
     4945802 a47fb0x1b23jn           483167840                     1       3900       3900        134
     4945803 a47fb0x1b23jn          3805993318                     1       3420       3420        134
     4945803 51c7zqy4ywmh1          3992646197                     1       1330       1330
     4945802 51c7zqy4ywmh1          3992646197                     1       1140       1140
     4945802 6sx8vfc0uc8zz          1628923514                     1        690        690
     4945803 6sx8vfc0uc8zz          1628923514                     1        680        680
     4945803 86b1vy6mprjpq          2955729951                     1        490        490
     4945802 86b1vy6mprjpq          2955729951                     1        470        470
     4945803 6033hbhdan9b8          3380418010                     1        480        480
…
```

There are two execution plans in play for the same problem statement in different instances of the process. I could also have profiled this by line number of plan to identify exactly which operation in the plan was consuming memory.

Things That Can Go Wrong

DISPLAY_AWR reports old costs

This is not really something that goes wrong, but it is a word of warning.

Here is an output from *display_awr*. Note the cost.

```
SELECT AWPATH_ID, AWTHREAD_ID
 FROM PS_SAC_AW_STEPINST
WHERE AWPRCS_ID = :1 AND SETID = :2
AND EFFDT = TO_DATE(:3,'YYYY-MM-DD') AND STAGE_NBR = :4 AND AWSTEP_STATUS <> :5  AND
AWTHREAD_ID IN (SELECT AWTHREAD_ID FROM PS_PV_REQ_AW WHERE PARENT_THREAD = 601330)
GROUP BY AWTHREAD_ID, AWPATH_ID
ORDER BY AWTHREAD_ID, AWPATH_ID

Plan hash value: 1898065720

---------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                    |       |       |  1165 (100)|          |
|   1 |  SORT GROUP BY               |                    |     3 |   216 |  1165   (2)| 00:00:14 |
|   2 |   TABLE ACCESS BY INDEX ROWID| PS_PV_REQ_AW       |     1 |    10 |     3   (0)| 00:00:01 |
|   3 |    NESTED LOOPS              |                    |     3 |   216 |  1164   (2)| 00:00:14 |
|   4 |     TABLE ACCESS FULL        | PS_SAC_AW_STEPINST |   167 | 10354 |   663   (4)| 00:00:08 |
|   5 |     INDEX RANGE SCAN         | PS_PV_REQ_AW       |     1 |       |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------


   1 - SEL$5DA710D3
   2 - SEL$5DA710D3 / PS_PV_REQ_AW@SEL$2
   4 - SEL$5DA710D3 / PS_SAC_AW_STEPINST@SEL$1
   5 - SEL$5DA710D3 / PS_PV_REQ_AW@SEL$2

Outline Data
-------------


  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('10.2.0.4')
      OPT_PARAM('_b_tree_bitmap_plans' 'false')
      OPT_PARAM('_complex_view_merging' 'false')
      OPT_PARAM('_unnest_subquery' 'false')
      OPT_PARAM('optimizer_dynamic_sampling' 4)
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$5DA710D3")
      UNNEST(@"SEL$2")
      OUTLINE(@"SEL$1")
      OUTLINE(@"SEL$2")
      FULL(@"SEL$5DA710D3" "PS_SAC_AW_STEPINST"@"SEL$1")
      INDEX(@"SEL$5DA710D3" "PS_PV_REQ_AW"@"SEL$2" ("PS_PV_REQ_AW"."AWTHREAD_ID"
              "PS_PV_REQ_AW"."AWPRCS_ID"))
      LEADING(@"SEL$5DA710D3" "PS_SAC_AW_STEPINST"@"SEL$1" "PS_PV_REQ_AW"@"SEL$2")
      USE_NL(@"SEL$5DA710D3" "PS_PV_REQ_AW"@"SEL$2")
      END_OUTLINE_DATA
  */

Peeked Binds (identified by position):
--------------------------------------


   1 - :1 (VARCHAR2(30), CSID=31): 'Requisition'
   2 - :2 (VARCHAR2(30), CSID=31): 'SHARE'
   3 - :3 (VARCHAR2(30), CSID=31): '1901-01-01'
   4 - :4 (VARCHAR2(30), CSID=31): '5'
   5 - :5 (VARCHAR2(30), CSID=31): 'F'

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

This is a plan I collected with EXPLAIN PLAN FOR and dbms_xplan.display. Same plan, but different cost. The cost in the plan produced by DISPLAY_AWR is the cost when the statement was first captured by AWR.

```
Plan hash value: 1898065720

----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |     3 |   216 |   136K  (1)| 00:27:16 |
|   1 |  SORT GROUP BY                |                     |     3 |   216 |   136K  (1)| 00:27:16 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| PS_PV_REQ_AW        |     1 |    10 |     3   (0)| 00:00:01 |
|   3 |    NESTED LOOPS               |                     |     3 |   216 |   136K  (1)| 00:27:16 |
|*  4 |     TABLE ACCESS FULL         | PS_SAC_AW_STEPINST | 45158 |  2734K|   667   (4)| 00:00:09 |
|*  5 |     INDEX RANGE SCAN          | PS_PV_REQ_AW        |     1 |       |     2   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------------

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

   2 - filter("PARENT_THREAD"=601330)
   4 - filter("STAGE_NBR"=TO_NUMBER(:4) AND "AWSTEP_STATUS"<>:5 AND "AWPRCS_ID"=:1 AND
              "SETID"=:2 AND "EFFDT"=TO_DATE(:3,'YYYY-MM-DD'))
   5 - access("AWTHREAD_ID"="AWTHREAD_ID")
```

Sometimes, when I use explain plan for I don't get the same plan. That is a bit of an alarm bell, but I can force the same plan by using the profile of hints in the plan produced by DISPLAY_AWR

Statement not in Library Cache

In an active system, especially one that routinely doesn't use bind variables, statements will get aged out of the library cache.

```
SELECT * FROM table(dbms_xplan.display_cursor('gpdwr389mg61h',0,'ADVANCED'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
SQL_ID: gpdwr389mg61h, child number: 0 cannot be found
```

Try looking in AWR with the dbms_xplan.display_awr function. You may still not find it because it had already been aged out at the time of the AWR snapshot. If you do find it remember that the costs could be old.

Only Some Statements are in the Library Cache

You've seen examples where literal values mean that each statement is different. So we aggregate by sql_plan_hash_value. This is a different variant on the theme. The innermost query sums time by SQL_ID and SQL_PLAN_HASH_VALUE, but we also outer join to DBA_HIST_SQLTEXT to see if we have captured the SQL text and plan.

Then I use an analytic function to find the rank statement within each execution plan, but notice I am ranking by time for statements in the AWR repository.

I still want the plans which have the most time.

```
Select *
FROM     (
         SELECT   ROW_NUMBER()[27] over (PARTITION BY x.sql_plan_hash_value ORDER BY x.awr_secs desc) as ranking
         ,        x.sql_id, x.sql_plan_hash_value
         ,        SUM(x.ash_secs) over (PARTITION BY x.sql_plan_hash_value) tot_ash_secs
         ,        SUM(x.awr_secs) over (PARTITION BY x.sql_plan_hash_value) tot_awr_secs
         ,        COUNT(distinct sql_id) over (PARTITION BY x.sql_plan_hash_value) sql_ids
         FROM     (
                  SELECT   h.sql_id
                  ,        h.sql_plan_hash_value
                  ,        SUM(10)[28] ash_secs
                  ,        10*count(t.sql_id)[29] awr_Secs
                  FROM     dba_hist_snapshot x
         ,        dba_hist_active_sess_history h
                           LEFT OUTER JOIN dba_hist_sqltext t
                           ON t.sql_id = h.sql_id
         WHERE    x.end_interval_time   >= TO_DATE('201003080830','yyyymmddhh24mi')
         AND      x.begin_interval_time <= TO_DATE('201003081200','yyyymmddhh24mi')
         AND      h.sample_time         >= TO_DATE('201003080830','yyyymmddhh24mi')
         AND      h.sample_time         <= TO_DATE('201003081200','yyyymmddhh24mi')
         AND      h.snap_id = x.snap_id
         AND      h.dbid = x.dbid
         AND      h.instance_number = x.instance_number
                  AND      h.module = 'WMS_RUN_TADM'
                  GROUP BY h.sql_id, h.sql_plan_hash_value
                  ) x
         ) y
where    y.ranking = 1
ORDER BY tot_ash_secs desc, ranking
/
```

[27] I am using ROW_NUMBER not rank because I want an arbitary ranked first statement, not all the equally first statements.

[28] So here I am counting time for statement in the ASH repository.

[29] Here I am counting time for statements all found in the AWR repository.

So now, I know that I can get plans for the SQL IDs with non-zero AWR time. There are still some statements for which I can get neither the SQL nor the execution plan.

```
                          SQL Plan
   RANKING SQL_ID        Hash Value TOT_ASH_SECS TOT_AWR_SECS    SQL_IDS
---------- ------------- ---------- ------------ ------------ ----------
         1 1wfhpn9k2x3hq          0         7960         4600         13
         1 2wsan9j1pk3j2 1061502179         4230         4230          1
         1 bnxddum0rrvyh  918066299         2640         1200        179
         1 02cymzmyt4mdh  508527075         2070            0         45[30]
         1 5m0xbf7vn8490 2783301143         1700            0         49
         1 0jfp0g054cb3n 4135405048         1500            0         47
         1 11bygm2nyqh0s 3700906241         1370            0         27
         1 6qg99cfg26kwb 3058602782         1300         1300          1
…
```

I can do the usual trick of generating the commands to get the SQL

```
SELECT 'SELECT * FROM
table(dbms_xplan.display_awr('''||sql_id||''','||sql_plan_hash_value||',NULL,''ADVANCED''))/*'||tot_ash_secs||','||
tot_awr_secs||'*/;'
FROM     (
         SELECT   ROW_NUMBER() over (PARTITION BY x.sql_plan_hash_value ORDER BY x.awr_secs desc) as ranking
         ,        x.sql_id, x.sql_plan_hash_value
         ,        SUM(x.ash_secs) over (PARTITION BY x.sql_plan_hash_value) tot_ash_secs
         ,        SUM(x.awr_secs) over (PARTITION BY x.sql_plan_hash_value) tot_awr_secs
         ,        COUNT(distinct sql_id) over (PARTITION BY x.sql_plan_hash_value) sql_ids
         FROM     (
                  SELECT   h.sql_id
                  ,        h.sql_plan_hash_value
                  ,        SUM(10) ash_secs
                  ,        10*count(t.sql_id) awr_Secs
                  FROM     dba_hist_snapshot x
         ,        dba_hist_active_sess_history h
                           LEFT OUTER JOIN dba_hist_sqltext t
                           ON t.sql_id = h.sql_id
         WHERE    x.end_interval_time   >= TO_DATE('201003080830','yyyymmddhh24mi')
         AND      x.begin_interval_time <= TO_DATE('201003081200','yyyymmddhh24mi')
         AND      h.sample_time         >= TO_DATE('201003080830','yyyymmddhh24mi')
         AND      h.sample_time         <= TO_DATE('201003081200','yyyymmddhh24mi')
         AND      h.snap_id = x.snap_id
         AND      h.dbid = x.dbid
         AND      h.instance_number = x.instance_number
                  AND      h.module = 'WMS_RUN_TADM'
                  GROUP BY h.sql_id, h.sql_plan_hash_value
                  ) x
         ) y
where    y.ranking = 1
```

[30] So we had 207 samples, representing 2070 seconds of SQL for statement with this execution plan. There are 45 distinct SQL_IDs, we don’t know how many executions wer are talking about, it is probably one per SQL_ID, but I don’t know that until 11g.

```
ORDER BY tot_ash_secs desc, ranking
/
```

```
SELECT * FROM table(dbms_xplan.display_awr('1wfhpn9k2x3hq',NULL,NULL,'ADVANCED'))/*7960,4600*/;
SELECT * FROM table(dbms_xplan.display_awr('2wsan9j1pk3j2',1061502179,NULL,'ADVANCED'))/*4230,4230*/
SELECT * FROM table(dbms_xplan.display_awr('bnxddum0rrvyh',918066299,NULL,'ADVANCED'))/*2640,1200*/;
SELECT * FROM table(dbms_xplan.display_awr('aaurjw06dyt5b',508527075,NULL,'ADVANCED'))/*2070,0*/;
SELECT * FROM table(dbms_xplan.display_awr('2s2xyadkmzxmv',2783301143,NULL,'ADVANCED'))/*1700,0*/;
SELECT * FROM table(dbms_xplan.display_awr('gkky737xp8v8z',4135405048,NULL,'ADVANCED'))/*1500,0*/;
SELECT * FROM table(dbms_xplan.display_awr('9sd7bjs6wc7xq',3700906241,NULL,'ADVANCED'))/*1370,0*/;
…
```

Lots of Shortlived Non-Shareable SQL

I have done the usual query to sum the time by SQL_ID, and I get one row per SQL ID, so instead I will GROUP BY plan hash value. So the SQL is different every time, but quite similar because they share plan hash values.

We are working from AWR history, so one sample every 10 seconds. We get one sample for each SQL_ID. So clearly I have lots of similar but different statements that don't take very long. I imagine a loop with litteral values instead of bind variables!

```
PRCSINSTANCE NUM_SQL_ID SQL_PLAN_HASH_VALUE  EXEC_SECS   ASH_SECS
------------ ---------- ------------------- ---------- ----------
    50007687        169           953836181       3170       1690
    50007687         50           807301148       3170        500
    50007687         22          4034059499       3170        220
    50007687         14          2504475139       3170        140
    50007687          2                   0       3170         70
    50007687          1          1309703960       3170         20
    50007687          1          3230852326       3170         10
    50007687          1          3257716453       3170         10
    50007687          1          3852975016       3170         10
    50007687          1          3205663729       3170         10
    50007687          1          2791534567       3170         10
    50007687          1          2098696903       3170         10
    50007687          1          1880529843       3170         10
    50007687          1          1173536273       3170         10
    50007687          1          1089066969       3170         10
    50007687          1           301402716       3170         10
```

Actually, I can get the execution plan for any of these statements in the AWR history, so in this variant of the query I have joined to DBA_HIST_SQLTEXT to see which SQL_IDs I do have information for (I can switch that to a left outer join to get back to the usual behaviour).

```
SELECT /*+LEADING(r x h) USE_NL(h)*/
       r.prcsinstance
,      COUNT(distinct h.sql_id) num_sql_id
,      h.sql_plan_hash_value
,      (CAST(r.enddttm AS DATE)-CAST(r.begindttm AS DATE))*86400 exec_secs
,      SUM(10) ash_secs
FROM   dba_hist_snapshot x
,      dba_hist_active_sess_history h
 INNER /*LEFT OUTER*/ JOIN DBA_HIST_SQLTEXT q
 ON q.dbid = h.dbid and q.sql_id = h.sql_id
,      sysadm.psprcsrqst r
WHERE x.end_interval_time >= r.begindttm
AND   x.begin_interval_time <= r.enddttm
AND   h.sample_time BETWEEN r.begindttm AND r.enddttm
AND   h.snap_id = x.snap_id
AND   h.dbid = x.dbid
AND   h.instance_number = x.instance_number
AND   h.module = r.prcsname
AND   h.action LIKE 'PI='||r.prcsinstance||'%'
AND   r.prcsinstance = 50007687
GROUP BY r.prcsinstance, r.prcsname, r.begindttm, r.enddttm
, h.sql_plan_hash_value
ORDER BY ash_secs DESC
```

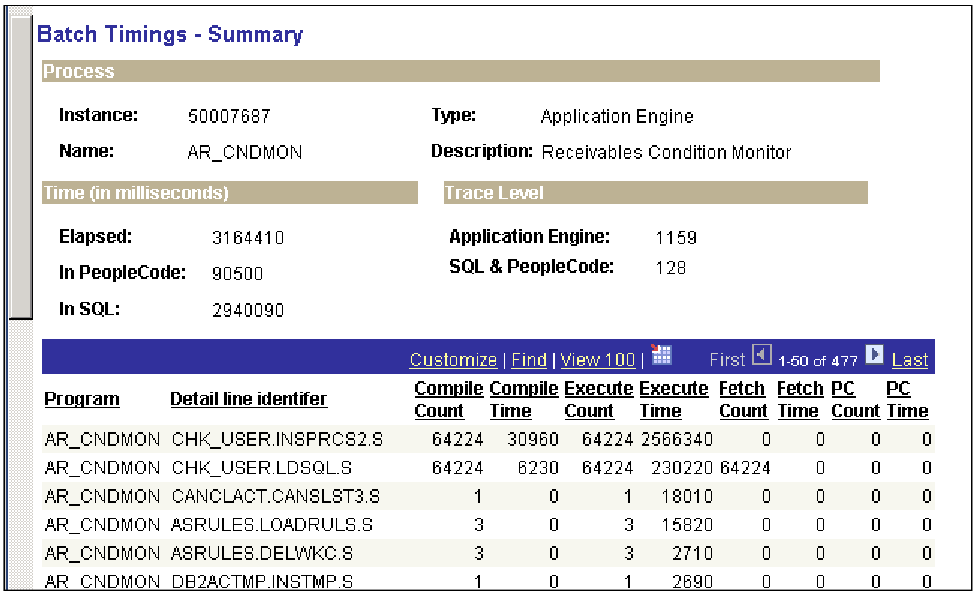
So the few that I have a plan for, are not very significant.

```
PRCSINSTANCE NUM_SQL_ID SQL_PLAN_HASH_VALUE  EXEC_SECS   ASH_SECS
------------ ---------- ------------------- ---------- ----------
    50007687          1                   0       3170         10
    50007687          1          3205663729       3170         10
    50007687          1          2791534567       3170         10
```

This is the Application Engine batch timings report for the same process. ASH suggests that the top execution plan had 169 exections, but remember that is a sample every 10 seconds.

The truth is much worse. The batch timings say there is a step that is executed 64224 times. It took 2566 seconds, so that is only 40ms per execution. So I am only sampling 1 in 250 executions, so no wonder I don't have many of them in the AWR repository. They are getting aged out too quickly.

It was also compiled 64224 times, and that tells me that this step does not have reuse statement, possible because there is dynamic SQL in play.



Batch Timings - Summary

Process

Instance:	50007687	**Type:**	Application Engine
Name:	AR_CNDMON	**Description:**	Receivables Condition Monitor

Time (in milliseconds)

Elapsed:	3164410
In PeopleCode:	90500
In SQL:	2940090

Trace Level

Application Engine:	1159
SQL & PeopleCode:	128

Program	Detail line identifer	Compile Count	Compile Time	Execute Count	Execute Time	Fetch Count	Fetch Time	PC Count	PC Time
AR_CNDMON	CHK_USER.INSPRCS2.S	64224	30960	64224	2566340	0	0	0	0
AR_CNDMON	CHK_USER.LDSQL.S	64224	6230	64224	230220	64224	0	0	0
AR_CNDMON	CANCLACT.CANSLST3.S	1	0	1	18010	0	0	0	0
AR_CNDMON	ASRULES.LOADRULS.S	3	0	3	15820	0	0	0	0
AR_CNDMON	ASRULES.DELWKC.S	3	0	3	2710	0	0	0	0
AR_CNDMON	DB2ACTMP.INSTMP.S	1	0	1	2690	0	0	0	0

I could criticise the kind of programming that leads to this, but it also shows a scenario where ASH will be of limited benefit.

This is a situation where I might want to use SQL trace to see what is going on in these statements. On the other hand, 40ms isn't bad for a SQL statement, how much faster can I make it.

Error ORA-06502

I have no idea why display_awr produces ORA-6502, but sometimes it does. It seems to be something to do with very large SQL statements. But you still get the execution plan.

```
SELECT * FROM table(dbms_xplan.display_awr('9vnan5kqsh1aq', 2262951047,NULL,'ADVANCED'));
```

```
SQL_ID 9vnan5kqsh1aq
--------------------
An uncaught error happened in prepare_sql_statement : ORA-06502: PL/SQL: numeric or value error

Plan hash value: 2262951047

--------------------------------------------------------------------------------------------------------------
| Id  | Operation                                 | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                          |                      |       |       |     1 (100)|          |
|   1 |  HASH GROUP BY                            |                      |     1 |   164 |     1 (100)| 00:00:01 |
…
```

The text is there, so you can go can get it FROM the AWR cache yourself.

```
SELECT sql_text FROM dba_hist_sqltext where sql_id = '9vnan5kqsh1aq'
```

Error ORA-01422

Sometimes, dbms_xplan fails because there are two SQL statements with the same SQL_ID.

```
An uncaught error happened in prepare_sql_statement : ORA-01422: exact fetch returns more than requested number of rows
```

This usually happens because the database has been cloned (from Production) and renamed, and then the same SQL statement has been captured by an AWR snapshot. The answer is to delete at least the duplicate rows from *sys.wrh$sqltext*.

```
delete
from sys.wrh$_sqltext t1
where t1.dbid != (select d.dbid from v$database d)
and exists(select 'x'
 from sys.wrh$_sqltext t2
 where t2.dbid = (select d.dbid from v$database d)
 and t2.sql_id = t1.sql_id)
```

Error ORA-44002

I have seen this with Global Temporary Tables and with direct path mode (the APPEND hint).

```
PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
ERROR: cannot get definition for table 'BZTNCMUX31XP5'
ORA-44002: invalid object name
```

Appendix

Further reading

- Sifting through the ASHes, Graham Wood, Oracle (http://www.oracle.com/technology/products/manageability/database/pdf/twp03/PPT_active_session_history.pdf)

- The ASHes of (DB) Time, Graham Wood at UKOUG2009 (http://www.ukoug.org/lib/show_document.jsp?id=11472).

    - And you can watch the video of Graham giving this presentation at MOW2009 on the Oracle Table Website

    - http://www.oaktable.net/media/mow2010-graham-wood-ashes-time-part1

    - http://www.oaktable.net/media/mow2010-graham-wood-ashes-time-part-2

- Doug Burns has written some excellent material many subjects including ASH on his Oracle Blog (http://oracledoug.com/serendipity/index.php?/plugin/tag/ASH).

- Introduction to DBMS_XPLAN (http://www.go-faster.co.uk/Intro_DBMS_XPLAN.ppt), UKOUG2008

    - With acknowledgements to 10g/11g DBMS_XPLAN, Carol Dacko, Collaborate 08