

AWR Mining V2

Trend Analysis

Maris Elsins
The Pythian Group Inc.
Riga, Latvia

Keywords:

AWR, Performance, Troubleshooting, Tuning, Oracle Database, Trends, Trend analysis

Introduction

AWR is a very important and useful source of information in situations when overall performance analysis of the system during a specific period of time needs to be reviewed. It's typically done by creating an AWR report and taking a look at the top wait events followed by a "drill down" into the specific category of interest. For example, if the top wait event is "db file sequential read", then the categories of interest might be "SQL ordered by Reads" and "Segments by Physical Reads", based on which the top segments and queries can be found.

While it's really easy to find the top SQL statements of information about resources consumed during the reporting period, it's important to remember that the resource consumption information sometimes is not granular enough with the default snapshot interval of AWR (1 hour), and it becomes even less granular (evened out) if reports for a longer time period are created. For example, in a report of 8am to 5pm, seeing that 10000 seconds (2.7 hours) were spent on CPU does not present much valuable information, as it's not clear if there were any spikes during which most of the time on CPU was spent or the CPU was evenly used during the whole day.

The purpose of this article is to show how one can increase usefulness of the information available in AWR beyond just looking at the AWR reports. We'll take a look how the information can be extracted and visualized to provide deeper insight into the past and future of the database performance.

AWR information as a "dot"

In many cases when we're interested in how certain system components perform, the AWR report describes this by a single number. For example, if we need to know how well the IO performed on September 9, we may create an AWR report for the whole day (see Illustration 1) and look at the "Foreground Wait Events" section (see Illustration 2) to find that the average wait time for a "db file sequential read" was 0.84 milliseconds.

	Snap Id	Snap Time	Sessions	Cursors/Session
Begin Snap:	20534	09-Sep-16 00:00:38	317	15.8
End Snap:	20582	10-Sep-16 00:00:29	312	15.8
Elapsed:		1,439.85 (mins)		
DB Time:		2,679.42 (mins)		

Illustration 1: Reporting interval for September 9, 2016

Foreground Wait Events

- s - second, ms - millisecond - 1000th of a second
- Only events with Total Wait Time (s) >= .001 are shown
- ordered by wait time desc, waits desc (idle events last)
- %Timeouts: value of 0 indicates value was < .5%. Value of null is truly 0

Event	Waits	%Time -outs	Total Wait Time (s)	Avg wait (ms)	Waits /txn	% DB time
db flash cache single block physical read	99,102,156		20,469	0.21	21.84	12.73
db flash cache multiblock physical read	18,608,091		9,755	0.52	4.10	6.07
cursor: pin S wait on X	1,705		5,782	3391.28	0.00	3.60
TCP Socket (KGAS)	66,258	17	4,891	73.82	0.01	3.04
log file sync	2,325,941		4,533	1.95	0.51	2.82
direct path write temp	2,755,278		3,933	1.43	0.61	2.45
direct path read	3,571,633		3,806	1.07	0.79	2.37
direct path read temp	3,502,393		1,696	0.48	0.77	1.05
log buffer space	20,502		1,173	57.21	0.00	0.73
db file sequential read	1,237,682		1,034	0.84	0.27	0.64
enq: TX - row lock contention	596		682	1144.09	0.00	0.42
read by other session	2,818,802		650	0.23	0.62	0.40
control file sequential read	976,939		514	0.53	0.22	0.32

Illustration 2: Foreground Wait Events on September 9, 2016

Sure enough “0.84 ms” doesn’t sound bad, however we can’t draw too much conclusions from this information as nothing’s said about what the performance was before or after September 9. It may be that the it’s much slower than the overall average seen in this system.

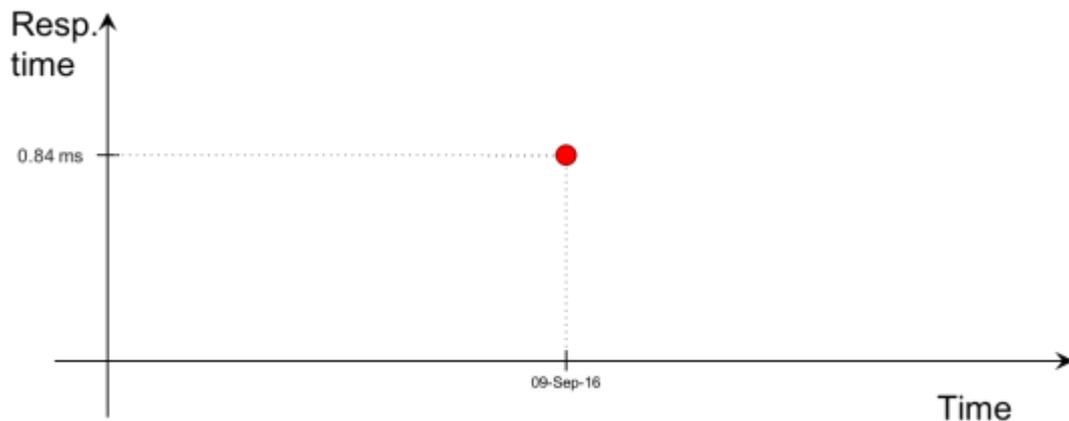


Illustration 3: Value as a dot on a timeline

This is why I’ve compared this bit of information from AWR to a “dot” – we have a time (September 9) and a value (0.84 ms) that can be easily put on time-value graph like I’ve done in Illustration 3, but nothing’s said about the past or the future performance.

Comparing two “dots”

One may think that this problem of not seeing the past and the future could be resolved by using the AWR Difference report that allows comparing two periods. In illustration 4 I’ve created the difference report for September 2 and September 9.

Snapshot Set	Begin Snap Id	Begin Snap Time	End Snap Id	End Snap Time	Avg Active Users	Elapsed Time (min)	DB time (min)
1st	20198	02-Sep-16 00:00:39 (Fri)	20246	03-Sep-16 00:00:49 (Sat)	1.6	1,440.2	2,294.2
2nd	20534	09-Sep-16 00:00:38 (Fri)	20582	10-Sep-16 00:00:29 (Sat)	1.9	1,439.9	2,679.4
%Diff					17.0	-0.0	16.8

Wait Events

• Ordered by absolute value of 'Diff' column of '% of DB time' descending (idle events last)

Event	Wait Class	% of DB time			# Waits/sec (Elapsed Time)			Total Wait Time (sec)			Avg Wait Time (ms)		
		1st	2nd	Diff	1st	2nd	%Diff	1st	2nd	%Diff	1st	2nd	%Diff
direct path write temp	User I/O	0.34	2.46	2.12	3.97	32.05	707.30	466.41	3,947.07	746.27	1.36	1.43	5.10
db flash cache single block physical read	User I/O	10.79	12.84	2.04	843.79	1,155.65	36.96	14,856.02	20,636.84	38.91	0.20	0.21	5.00
direct path read	User I/O	4.38	2.37	-2.01	78.59	41.39	-47.33	6,023.72	3,807.73	-36.79	0.89	1.06	19.10
cursor: pin S wait on X	Concurrency	1.90	3.60	1.70	0.02	0.02	0.00	2,612.99	5,782.13	121.28	1,786.05	3,391.28	89.88
log file sync	Commit	1.58	2.62	1.24	26.34	26.94	2.28	2,171.29	4,536.02	108.91	0.95	1.95	105.26
db flash cache multiblock physical read	User I/O	5.15	6.08	0.93	160.33	215.61	34.48	7,084.69	9,771.19	37.92	0.51	0.52	1.96
library cache lock	Concurrency	1.04	0.16	-0.89	0.01	0.01	0.00	1,437.74	251.74	-82.49	1,840.89	342.98	-81.37
direct path read temp	User I/O	0.27	1.05	0.78	7.07	40.54	473.41	377.80	1,695.79	348.86	0.62	0.48	-22.58
log buffer space	Configuration	0.08	0.73	0.65	0.02	0.24	1,100.00	108.97	1,177.77	980.82	58.71	57.22	-2.54
log file sequential read	System I/O	0.22	0.75	0.54	0.98	1.31	33.67	298.89	1,209.77	304.75	3.52	10.70	203.96
db file sequential read	User I/O	1.31	0.83	-0.47	29.25	21.66	-25.95	1,798.68	1,341.07	-25.44	0.71	0.72	1.41
control file sequential read	System I/O	1.95	1.61	-0.34	20.78	20.93	0.72	2,684.25	2,580.70	-3.86	1.49	1.43	-4.00

Illustration 4: AWR Difference report

The difference report gives us two dots on the timeline, that may give us an impression of knowing how values change over time. See Illustration 5, where the values from the difference reports are graphed.

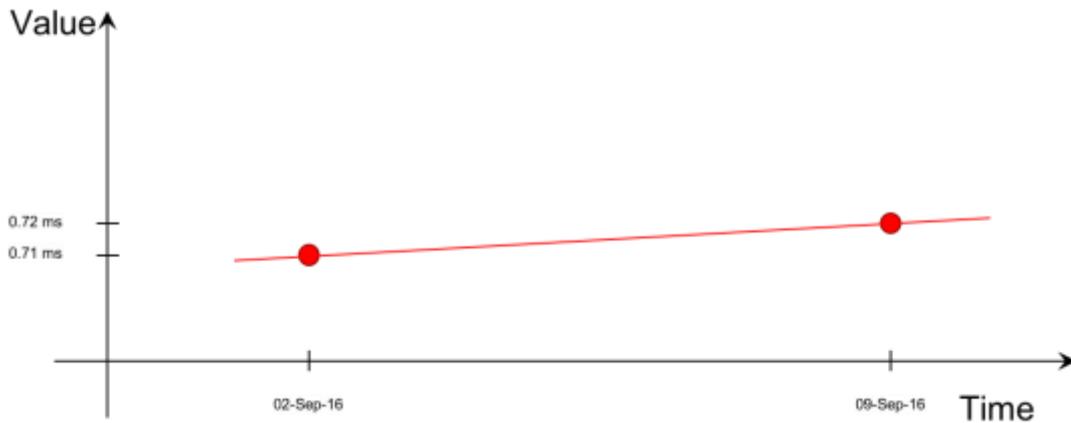


Illustration 5: Graph with the values from the Difference report

In reality, however, there's nothing that supports the "trend" that we're discovered by looking at the values from the comparison report. The reality may be different because the AWR's Difference report does not reveal the nature of the metrics at times outside the compared intervals, and the real situation may close to one displayed in Illustration 6.

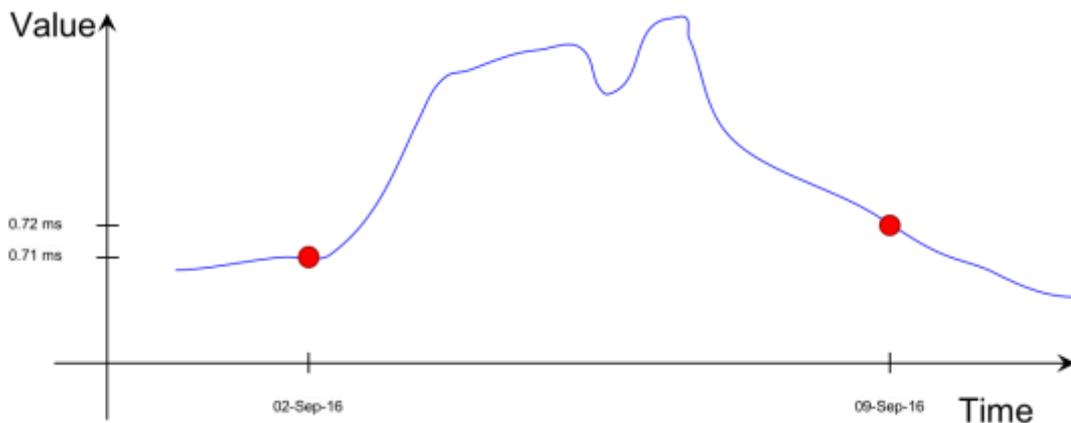


Illustration 6: Unknown values outside the reporting intervals of the AWR Difference report

Mining trend data from AWR Performance views

The DBA_HIST_% views in the database can be queried to access the AWR data directly. Depending on the version of the database the total number of available views differs, as each of the views hold information for one particular feature or metric. For example, DBA_HIST_SQLSTAT provides statistics (block gets, block reads, CPU time, elapsed time, etc.) for top queries in the database, DBA_HIST_SYSSTAT provides values of every system statistic at time of each snapshot, and so on. Depending on the information you're looking for, you'll need to find the correct DBA_HIST_% view that's outside the scope of this article¹.

Simplest way of showing how to extract data from AWR is providing an example. Here we'll try to extract information about how many commits happened in the system over time. "user commit" is a system statistic that's represented in DBA_HIST_SYSSTAT AWR performance view. Thus we can build a very simple query to show all the "user commits" values that are stored in AWR. The basic query is displayed in Listing 1, where values are ordered by snapsid (additionally in a RAC database one may be interested in displaying INSTANCE_NUMBER column too as the data is captured per instance, here we're looking at a single-instance system).

```
SELECT v.snap_id,
       v.value
FROM   dba_hist_sysstat v
WHERE  v.stat_name = 'user commits'
ORDER BY v.snap_id;
```

SNAP_ID	VALUE
12621	625766
12622	626608
12623	627853
12624	629348
12625	630130
12626	630904

....

Listing 1: A basic "user commits" query from DBA_HIST_SYSSTAT

At the moment the query does not provide much useful data because the values are cumulative since the start of the instance and the snap_ids don't allow us to see when the snapshot was taken. In Listing 2 I've expanded the query to use analytic function lag() for calculation of the deltas between snapshots, thus it will be clear how many "user commits" actually happened during each interval between the snapshots. Additionally, from_snap is retrieved via the lag() function to clearly display the start and the end snapshot between for the reporting interval.

```
SELECT Lag(snap_id) over (ORDER BY v.snap_id) from_snap,
       v.snap_id to_snap,
       v.value - Lag(v.value) over (ORDER BY v.snap_id) delta_value
FROM   dba_hist_sysstat v
WHERE  v.stat_name = 'user commits'
ORDER BY v.snap_id;
```

FROM_SNAP	TO_SNAP	DELTA_VALUE
	12621	
12621	12622	842
12622	12623	1245
12623	12624	1495
12624	12625	782
12625	12626	774

...

Listing 2: Delta values and interval endpoints

Next, in Listing 3, we're joining DBA_HIST_SNAPSHOT view, that provides information about the snapshots, from which we're particularly interested in the time of the snapshot, this instead of snap_id we're displaying end_interval_time from DBA_HIST_SNAPSHOT.

¹ The description in DBA_HIST_% views can be found in Oracle Database Reference manual <https://docs.oracle.com/database/121/REFRN/title.htm> (for version 12.1)

```

SELECT Lag(s.end_interval_time) over (PARTITION BY s.dbid, s.startup_time ORDER BY v.snap_id)
from_time,
       s.end_interval_time to_time,
       v.value - Lag(v.value) over (PARTITION BY s.dbid, s.startup_time ORDER BY v.snap_id)
delta_value
FROM   dba_hist_sysstat v,
       dba_hist_snapshot s
WHERE  v.stat_name = 'user commits'
       AND s.snap_id = v.snap_id
ORDER BY s.end_interval_time;

```

FROM_TIME	TO_TIME	DELTA_VALUE
09-SEP-16 04.00.44.539000000 PM	09-SEP-16 05.00.58.620000000 PM	842
09-SEP-16 05.00.58.620000000 PM	09-SEP-16 06.00.22.386000000 PM	1245
09-SEP-16 06.00.22.386000000 PM	09-SEP-16 07.00.35.070000000 PM	1495
09-SEP-16 07.00.35.070000000 PM	09-SEP-16 08.00.45.075000000 PM	782
09-SEP-16 08.00.45.075000000 PM	09-SEP-16 09.00.54.573000000 PM	774
09-SEP-16 09.00.54.573000000 PM	09-SEP-16 10.00.04.401000000 PM	768
...		

Listing 3: Snapshot timings as interval endpoints

Surely, we may need to filter the data by snapshot time as we may not be interested in the whole history that's available in the AWR. Listing 4 displays the data between September 11 and September 16 (non-inclusive).

```

SELECT *
FROM   (SELECT Lag(s.end_interval_time) over (PARTITION BY s.dbid, s.startup_time ORDER BY
v.snap_id) from_time,
       s.end_interval_time to_time,
       v.value - Lag(v.value) over (PARTITION BY s.dbid, s.startup_time ORDER BY
v.snap_id) delta_value
FROM   dba_hist_sysstat v,
       dba_hist_snapshot s
WHERE  v.stat_name = 'user commits'
       AND s.snap_id = v.snap_id)
WHERE  from_time BETWEEN To_date('11092016', 'DDMMYYYY') AND To_date('16092016', 'DDMMYYYY')
ORDER BY from_time;

```

FROM_TIME	TO_TIME	DELTA_VALUE
11-SEP-16 12.00.05.638000000 AM	11-SEP-16 01.00.23.855000000 AM	785
11-SEP-16 01.00.23.855000000 AM	11-SEP-16 02.00.40.073000000 AM	1057
11-SEP-16 02.00.40.073000000 AM	11-SEP-16 03.00.46.109000000 AM	1096
...		
15-SEP-16 09.00.04.192000000 PM	15-SEP-16 10.00.16.087000000 PM	786
15-SEP-16 10.00.16.087000000 PM	15-SEP-16 11.00.35.078000000 PM	779
15-SEP-16 11.00.35.078000000 PM	16-SEP-16 12.00.51.092000000 AM	831

Listing 4: Filtering by date

And finally, one may consider aggregating some of the data based on the needs. In this example I'll aggregate the data by date as I'm interested in finding the "busiest day" of the week. Listing 5 displays the aggregation.

```

SELECT Trunc(from_time, 'DD') t_from_time,
       SUM(delta_value) sum_delta_value
FROM   (SELECT Lag(s.end_interval_time) over (PARTITION BY s.dbid, s.startup_time ORDER BY
v.snap_id) from_time,
       s.end_interval_time to_time,
       v.value - Lag(v.value) over (PARTITION BY s.dbid, s.startup_time ORDER BY
v.snap_id) delta_value
FROM   dba_hist_sysstat v,

```

```

        dba_hist_snapshot s
    WHERE v.stat_name = 'user commits'
        AND s.snap_id = v.snap_id)
WHERE from_time BETWEEN To_date('11092016', 'DDMMYYYY') AND To_date('16092016', 'DDMMYYYY')
GROUP BY Trunc(from_time, 'DD')
ORDER BY t_from_time;

```

T_FROM_TIME	SUM_DELTA_VALUE
11.09.2016 00:00:00	21602
12.09.2016 00:00:00	29173
13.09.2016 00:00:00	32772
14.09.2016 00:00:00	30477
15.09.2016 00:00:00	26799

Listing 5: Aggregating the data by date

Once the data is retrieved it can be put on a graph to make it more simple to notice the differences. Illustration 7 clearly shows that September 13 was the busiest of the days.

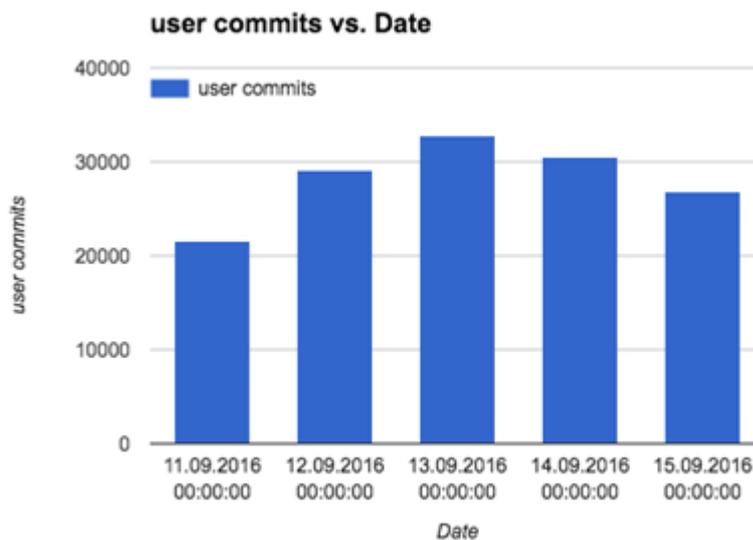


Illustration 7: Graphing the extracted data

Similarly, one can build queries for any other data that can be found in AWR. It is not necessary to write the whole query at once (like one displayed in Listing 5), building it bit by bit slowly by adding filters/data one by one based on the requirements sometimes is much simpler way of retrieving the required information.

Let's return for the situation described at the beginning of the article. How did "db file sequential read" performance change between September 2 and 9? We can extract data from AWR about every snapshot between these two dates.

Listing 6 shows a query I use to report trend of wait event statistics over time. Its text is displayed too, but indeed it's built based on the same principles we discussed above. The query takes 3 arguments – wait event name, reporting interval in days (last n days), and the aggregation interval in hours. In the listing I've reported last 10 days by displaying hourly statistics.

```

$ cat awr_wait_trend.sql
def event_name="&1"
def days_history="&2"
def interval_hours="&3"
select to_char(time,'DD.MM.YYYY HH24:MI:SS') time, event_name, sum(delta_total_waits)
total_waits, round(sum(delta_time_waited/1000000),3) total_time_s,

```

```

round(sum(delta_time_waited)/decode(sum(delta_total_waits),0,null,sum(delta_total_waits))/1000,3)
avg_time_ms from
  (select hse.snap_id,
    trunc(sysdate-&days_history+1)+trunc((cast(hs.begin_interval_time as date)-(trunc(sysdate-
&days_history+1)))*24/(&interval_hours))*(&interval_hours)/24 time,
    EVENT_NAME,
    TOTAL_WAITS-(lead(TOTAL_WAITS,1) over(partition by hs.STARTUP_TIME, EVENT_NAME order by
hse.snap_id)) delta_total_waits,
    TIME_WAITED_MICRO-(lag(TIME_WAITED_MICRO,1) over(partition by hs.STARTUP_TIME, EVENT_NAME
order by hse.snap_id)) delta_time_waited
  from DBA_HIST_SYSTEM_EVENT hse, DBA_HIST_SNAPSHOT hs
  where hse.snap_id=hs.snap_id
    and hs.begin_interval_time>=trunc(sysdate)-&days_history+1
    and hse.EVENT_NAME like '&event_name')
group by time, event_name
order by 2, to_date(time,'DD.MM.YYYY HH24:MI:SS');
...

```

```
SQL> @awr_wait_trend.sql "db file sequential read" 10 1
```

TIME (SNAP)	EVENT_NAME	TOTAL_WAITS	TOTAL_TIME_S	AVG_TIME_MS
...				
09.09.2016 00:00:00	db file sequential read	55302	50.670	.916
09.09.2016 01:00:00	db file sequential read	45372	31.303	.690
09.09.2016 02:00:00	db file sequential read	80171	68.995	.861
09.09.2016 03:00:00	db file sequential read	215704	235.555	1.092
09.09.2016 04:00:00	db file sequential read	67104	46.547	.694
09.09.2016 05:00:00	db file sequential read	172241	104.396	.606
09.09.2016 06:00:00	db file sequential read	65497	36.598	.559
09.09.2016 07:00:00	db file sequential read	76487	46.231	.604
09.09.2016 08:00:00	db file sequential read	53193	27.399	.515
09.09.2016 09:00:00	db file sequential read	78568	45.846	.584
09.09.2016 10:00:00	db file sequential read	88612	48.192	.544
09.09.2016 11:00:00	db file sequential read	71483	42.597	.596
09.09.2016 12:00:00	db file sequential read	79741	53.432	.670
09.09.2016 13:00:00	db file sequential read	89707	65.399	.729
09.09.2016 14:00:00	db file sequential read	85208	49.717	.583
09.09.2016 15:00:00	db file sequential read	125878	99.274	.789
09.09.2016 16:00:00	db file sequential read	71224	39.754	.558
09.09.2016 17:00:00	db file sequential read	78324	46.832	.598
09.09.2016 18:00:00	db file sequential read	72536	67.021	.924
09.09.2016 19:00:00	db file sequential read	49199	37.167	.755
09.09.2016 20:00:00	db file sequential read	47840	26.727	.559

Listing 6: "db file sequential reads" data

Having these outputs available it's simple to put more detailed information about September 2 and September 9 on the graph, that allows noticing details that are hidden by just having one number in the AWR report. Illustration 8 shows the differences of the captured "db file sequential read" response times between two days. Illustration 9 compares the numbers of total waits per day.

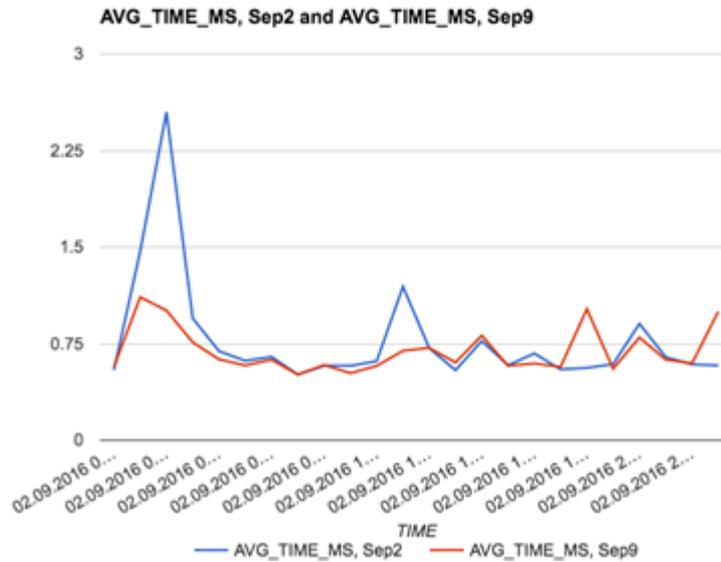


Illustration 8: “db file sequential reads” response time on September 2 and September 9

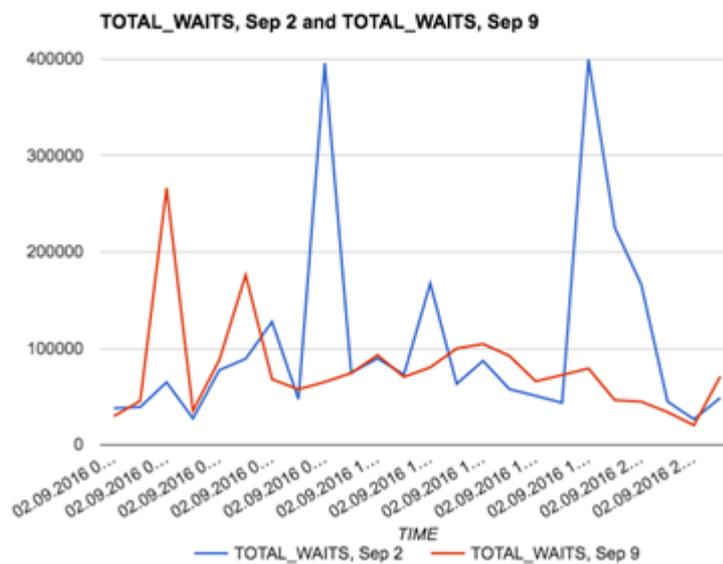


Illustration 9: “db file sequential reads” response time on September 2 and September 9

Additionally, we may graph the response times and total wait counts on the same graph to look for correlations or abnormalities during the whole week between these dates (Illustration 10).

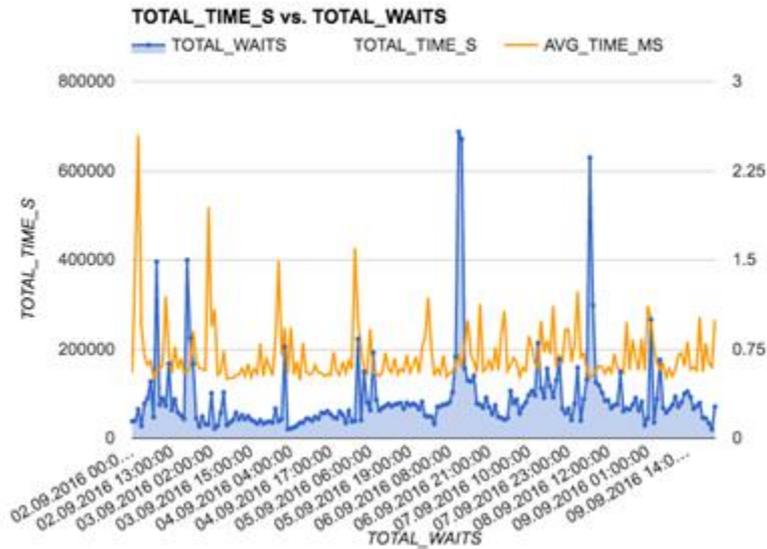


Illustration 10: “db file sequential reads” metrics – the weekly view

Mining trends from AWR: Example

This is one of uncountable examples of how AWR mining can be used to research issues and find root causes.

The Problem

- Platform: Amazon RDS for Oracle, db.m3.xlarge (4 vCPU, 15GiB), 11.2.0.4
- Alert from the client: Could you please take a look at the DB utilization in prod? We're seeing 100% utilization and 795 connections. Could you please let us know if any specific connection is causing high CPU utilization?

In this case, as it wasn't really clear what's causing the issue, I used Tanel Poder's `ashtop.sql`² to take a look at the top statements working on CPU, and found one particular query (`sql_id=4db73upm43ck1`) spinning on CPU for the whole time of 5-minute report. See Listing 7 for outputs.

```
SQL> @ashtop session_id,sql_id "event is null" sysdate-5/24/60 sysdate
```

Total Seconds	AAS	%This	SESSION_ID	SQL_ID	FIRST_SEEN	LAST_SEEN	...
300	1.0	12%	738	4db73upm43ck1	2016-06-06 10:16:40	2016-06-06 10:21:27	...
300	1.0	12%	1336	4db73upm43ck1	2016-06-06 10:16:40	2016-06-06 10:21:27	...
300	1.0	12%	1358	4db73upm43ck1	2016-06-06 10:16:40	2016-06-06 10:21:27	...
300	1.0	12%	745	4db73upm43ck1	2016-06-06 10:16:40	2016-06-06 10:21:27	...
300	1.0	12%	882	4db73upm43ck1	2016-06-06 10:16:40	2016-06-06 10:21:27	...
300	1.0	12%	2046	4db73upm43ck1	2016-06-06 10:16:40	2016-06-06 10:21:27	...
99	.3	4%	1472	grbz54xqkr425	2016-06-06 10:17:04	2016-06-06 10:18:43	...
64	.2	3%	1472	fy4c407vamaks	2016-06-06 10:18:52	2016-06-06 10:21:26	...
37	.1	2%	2176		2016-06-06 10:18:29	2016-06-06 10:19:50	...
22	.1	1%	977	7tdudtm4x03np	2016-06-06 10:16:40	2016-06-06 10:17:01	...
16	.1	1%	795	3b8s69ct88pur	2016-06-06 10:16:40	2016-06-06 10:18:13	...
15	.1	1%	989		2016-06-06 10:17:04	2016-06-06 10:17:19	...
13	.0	1%	632	gdhy97b59bk81	2016-06-06 10:16:46	2016-06-06 10:17:07	...
11	.0	0%	1792		2016-06-06 10:16:59	2016-06-06 10:21:09	...
11	.0	0%	2133	atmhpvm759a97	2016-06-06 10:16:50	2016-06-06 10:21:02	...

Listing 7: Using `ashtop.sql` to find the top SQL statements in last 5 minutes

² <http://blog.tanelpoder.com/files/scripts/ash/ashtop.sql>

From here there are two ways how to proceed. One – trying to take a look at the query to understand what’s wrong, probably try tuning it or maybe try understanding why suddenly this query is executed that many times and is spinning on CPU. This for sure would take some time. Another approach is to use the data stored in AWR and take a quick look at how the query executed in the past.

I’ve used a query (in Listing 8) to retrieve information about how query 4db73upm43ck1 performed in the past. Script `awr_sqlid_perf_trend_by_plan.sql` requires 3 parameters – `sql_id`, how many days in the past to retrieve the data for, aggregation interval in hours. I’ve executed the script with parameters “4db73upm43ck1 5 24” to see the daily execution statistics for past 5 days.

```
$ cat awr_sqlid_perf_trend_by_plan.sql
def sql_id="&1"
def days_history="&2"
def interval_hours="&3"

select hss.instance_number inst,
       to_char(trunc(sysdate-&days_history+1)+trunc((cast(hs.begin_interval_time as date)-
(trunc(sysdate-&days_history+1)))*24/(&interval_hours))*(&interval_hours)/24, 'dd.mm.yyyy
hh24:mi:ss') time,
       plan_hash_value,
       sum(hss.executions_delta) executions,
       round(sum(hss.elapsed_time_delta)/1000000,3) elapsed_time_s,
       round(sum(hss.cpu_time_delta)/1000000,3) cpu_time_s,
       round(sum(hss.iowait_delta)/1000000,3) iowait_s,
       round(sum(hss.clwait_delta)/1000000,3) clwait_s,
       round(sum(hss.apwait_delta)/1000000,3) apwait_s,
       round(sum(hss.ccwait_delta)/1000000,3) ccwait_s,
       round(sum(hss.rows_processed_delta),3) rows_processed,
       round(sum(hss.buffer_gets_delta),3) buffer_gets,
       round(sum(hss.disk_reads_delta),3) disk_reads,
       round(sum(hss.direct_writes_delta),3) direct_writes
from dba_hist_sqlstat hss, dba_hist_snapshot hs
where hss.sql_id='&sql_id'
      and hss.snap_id=hs.snap_id
      and hs.begin_interval_time>=trunc(sysdate)-&days_history+1
group by hss.instance_number, trunc(sysdate-&days_history+1)+trunc((cast(hs.begin_interval_time
as date)-(trunc(sysdate-&days_history+1)))*24/(&interval_hours))*(&interval_hours)/24,
plan_hash_value
having sum(hss.executions_delta)>0
order by hss.instance_number, trunc(sysdate-&days_history+1)+trunc((cast(hs.begin_interval_time
as date)-(trunc(sysdate-&days_history+1)))*24/(&interval_hours))*(&interval_hours)/24, 4 desc;

SQL> @awr_sqlid_perf_trend_by_plan.sql 4db73upm43ck1 5 24
```

TIME	PLAN_HASH_VALUE	EXECUTIONS	ELAPSED_TIME_S	CPU_TIME_S	IOWAIT_S	BUFFER_GETS	DISK_READS
02.06.2016 00:00:00	1597670781	291	278.018	254.272	22.717	19787274.000	50740.000
03.06.2016 00:00:00	1597670781	205	204.755	189.479	14.583	14986545.000	71451.000
04.06.2016 00:00:00	1597670781	3	8.275	4.483	3.936	314594.000	15790.000
04.06.2016 00:00:00	4142332636	2	6.149	3.933	2.307	119766.000	47343.000
05.06.2016 00:00:00	82218884	47	11805.808	5766.836	22.799	1247372099.000	173254.000
06.06.2016 00:00:00	82218884	127	7216.393	3751.640	9.431	522088617.000	24677.000

Listing 8: Retrieving the execution trend for query 4db73upm43ck1

After looking at the outputs it’s immediately visible that the query changed the execution plan to one that performs much worse – thus the root cause of the problem is found. There’s also an easy and quick way how this can be solved – by creating a SQL Plan Baseline for the query that allows using one of the better plans – 1597670781. The implementation of the solution is described in Listing 9, where first the good execution plan is unloaded from AWR into a SQL Tuning Set, and then – a SQL Plan Baseline is created from the execution plan in the SQL Tuning Set (There is no direct way how a baseline could be created from the plan stored in AWR).

```

exec DBMS_SQLTUNE.CREATE_SQLSET(sqlset_name => 'CR1064802', description => 'Plan for sql_id
4db73upm43ck1');

DECLARE
  cur sys_refcursor;
BEGIN
  OPEN cur FOR SELECT VALUE(P) FROM
table(dbms_sqltune.select_workload_repository(11434,12880,'sql_id=' '4db73upm43ck1' ' and
plan_hash_value=1597670781' , NULL, NULL, NULL, NULL, NULL, 'ALL')) P;
  DBMS_SQLTUNE.LOAD_SQLSET(load_option=>'MERGE',sqlset_name => 'CR1064802', populate_cursor =>
cur);
  CLOSE cur;
END;

VARIABLE cnt NUMBER
EXECUTE :cnt := DBMS_SPM.LOAD_PLANS_FROM_SQLSET(
  sqlset_name => 'CR1064802',
  basic_filter => 'sql_id=' '4db73upm43ck1' ');

```

Listing 9: Creating a SQL Plan Baseline from an execution plan stored in AWR

Obviously, after implementation of the fix, some of the running sessions needed to be terminated and the existing cursors were flushed (DBMS_SHARED_POOL.PURGE). In a while after the fix had been implemented the results were observed by running the same script (see listing 10).

```
SQL> @awr_sqlid_perf_trend_by_plan.sql 4db73upm43ck1 2 1
```

TIME	PLAN_HASH_VALUE	EXECUTIONS	ELAPSED_TIME_S	CPU_TIME_S	IOWAIT_S	BUFFER_GETS	DISK_READS
...							
06.06.2016 00:00:00	82218884	3	4.600	4.019	.608	1425936.000	965.000
06.06.2016 01:00:00	82218884	4	14.374	11.740	2.756	4726784.000	12836.000
06.06.2016 02:00:00	82218884	20	34.437	33.125	.619	7475353.000	264.000
06.06.2016 03:00:00	82218884	3	4.527	4.485	.032	1205412.000	14.000
06.06.2016 04:00:00	82218884	18	13.543	13.366	.132	1346631.000	57.000
06.06.2016 05:00:00	82218884	22	43.248	40.954	2.217	14290668.000	4439.000
06.06.2016 06:00:00	82218884	11	78.463	77.897	.311	45601072.000	79.000
06.06.2016 07:00:00	82218884	2	3.338	2.155	1.237	165446.000	3369.000
06.06.2016 08:00:00	82218884	10	9.241	9.098	.098	900049.000	61.000
06.06.2016 09:00:00	82218884	34	7010.622	3554.801	1.422	444951266.000	2593.000
06.06.2016 10:00:00	82218884	15	10942.980	4653.979	4.637	586951399.000	2078.000
06.06.2016 10:00:00	1597670781	1	7.910	2.047	.323	117577.000	28.000
06.06.2016 11:00:00	1597670781	1	1.028	1.008	.020	117468.000	28.000

Summary

Data in Automatic Workload Repository (a feature that requires a Diagnostic Pack licenses) is typically viewed by creating one of the reporting scripts that Oracle provides (AWR report, AWR Global Report, AWR Difference Report, ...). But there are other ways how one can take a closer look at the data in more detail by paying special attention to how the metrics change over time. The goal of this paper was to provide an insight into building queries for extracting the data and graphing them, as well as showing an example of how this approach can be successfully used to resolve a critical performance issue.

Contact address:

Maris Elsins
Lead Database Consultant
The Pythian Group Inc.
Putnu 14-12
Riga, LV-1058
Latvia

Phone: +1 613 565 8696 x337

Email: elsins@pythian.com

Internet: <http://www.pythian.com/blog/author/elsins/> , <https://me-dba.com>