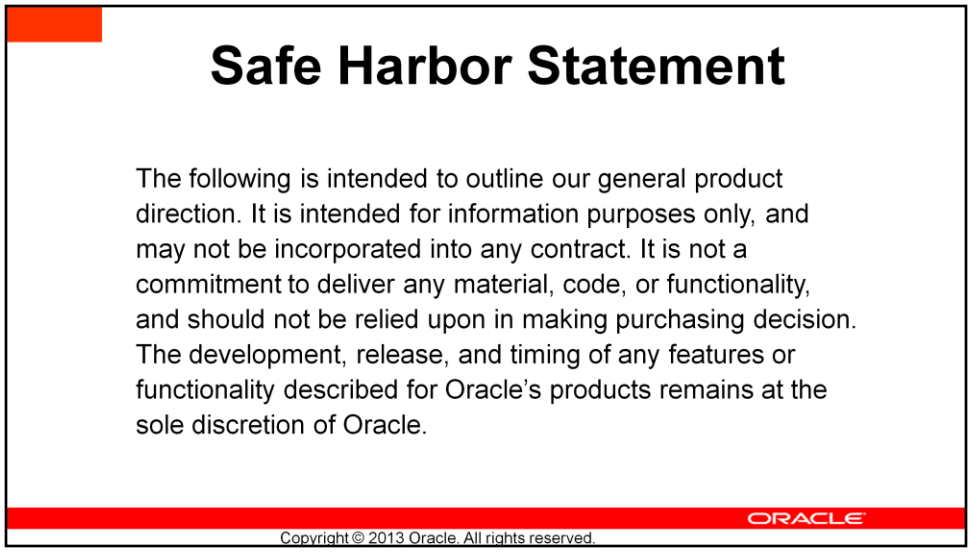


AW template – Safe Harbor Slide

HOST:

Please remember this Advisor Webcast is intended for information purposes only. Any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

AW template – Title slide

Let's get started with our presentation. Roger if you're ready, I'll turn the Webcast over to you and you can begin.

Objectives

Review Basic Concepts

Identify Bottlenecks

Characterize Problems

Identify Problem Session(s)

ORACLE

AW template – content slide

1.Provide 3-4 objectives.

2.Simple, specific, measurable

3.What should audience be able to do after the webcast?

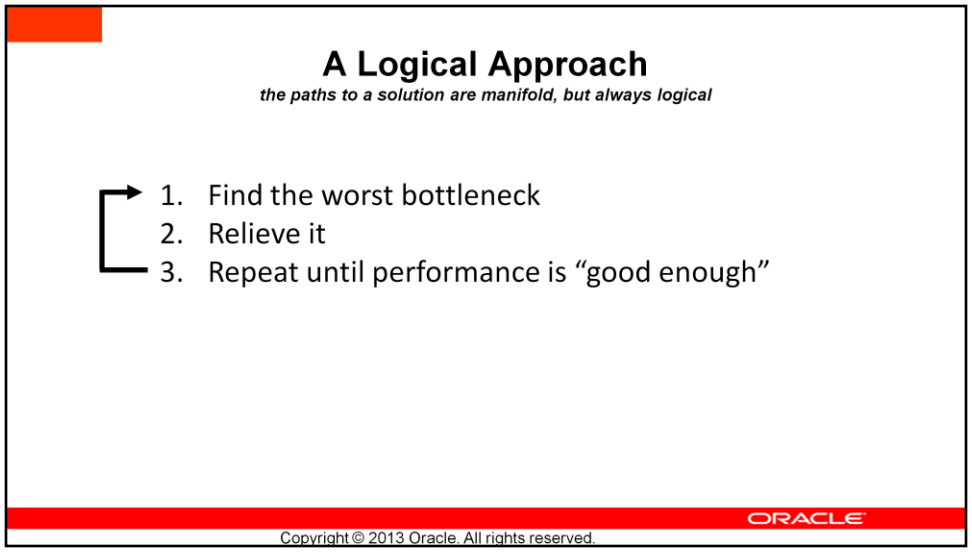
Agenda

What we can attempt in 45 minutes...

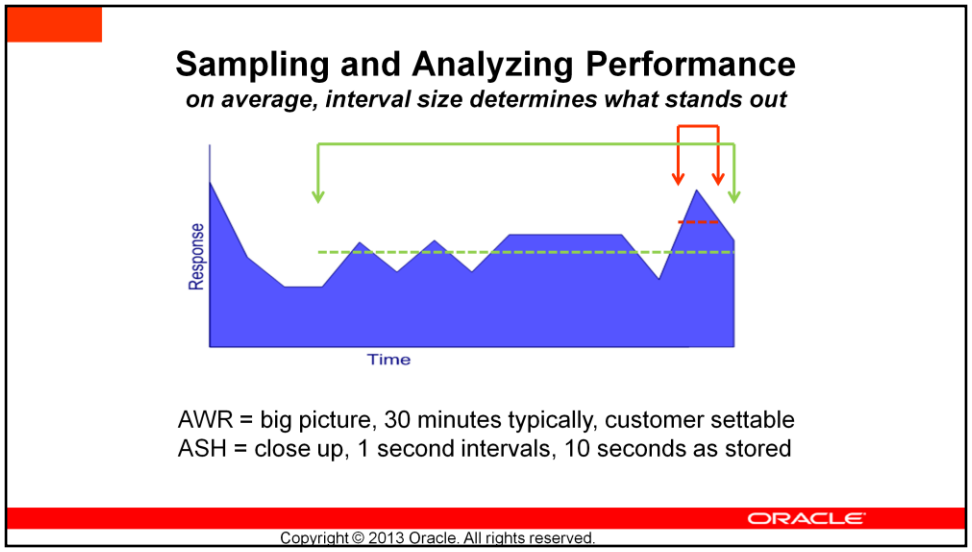
Review Basic Performance Concepts and Approach
Understand Interval Sampling
Review AWR in context of a partial case study
Continue same case with ASH
Drive toward solution– typically SQL in need of tuning
Understand limitations of both AWR and ASH

ORACLE

AW template – content slide



This is a simplified version of a performance diagnostic methodology, sans formality.
Essentially, it is a form of the scientific method, in that we assume the Causation Principle to be true, and we use logic. We base our performance diagnostic efforts on evidence, not guesswork. Well, actually, there is plenty of guesswork. But it is good, logical guesswork, not just dumb luck.
If evidence does not support our tentative hypothesis, we start over and restate the problem instead of blindly stabbing in the dark.



The two main tools we use at the instance level to identify performance issues are Application Workload Repository (AWR) and Active Session History (ASH)

It is most helpful to be able to use the two together, as we shall see.

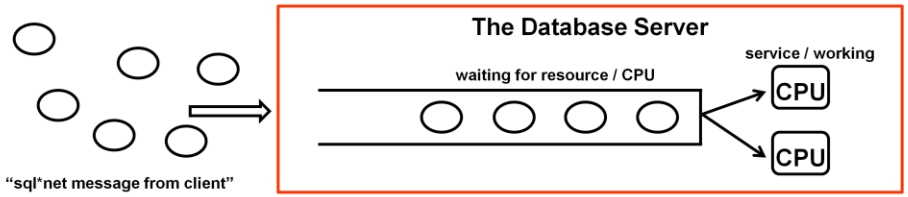
AWR is an interval sampling tool for the entire instance, normally averaging data for all sessions. It typically is set to gather data at 30 minute intervals, though this can be adjusted by the user. OLTP database samples over many hours average data for all sessions across the instance, and normally does not provide enough detail focus to identify bottlenecks.

ASH reports timed events, by session, at one second intervals. As such, it is “up close” and is able to identify session bottlenecks immediately. It is a “little picture” tool, but gathers data from the entire instance every second. The data are eventually written to the database repository, but the longer term data are only at ten second intervals, not one second. Still, quite powerful a tool and useful for day to day performance analysis, particularly for the session level details it provides.

It is essential to create AWR reports and ASH reports during periods of bad performance. It is obviously useless to report data before or after a problem, unless it is for comparing good to bad performance for similar workloads.

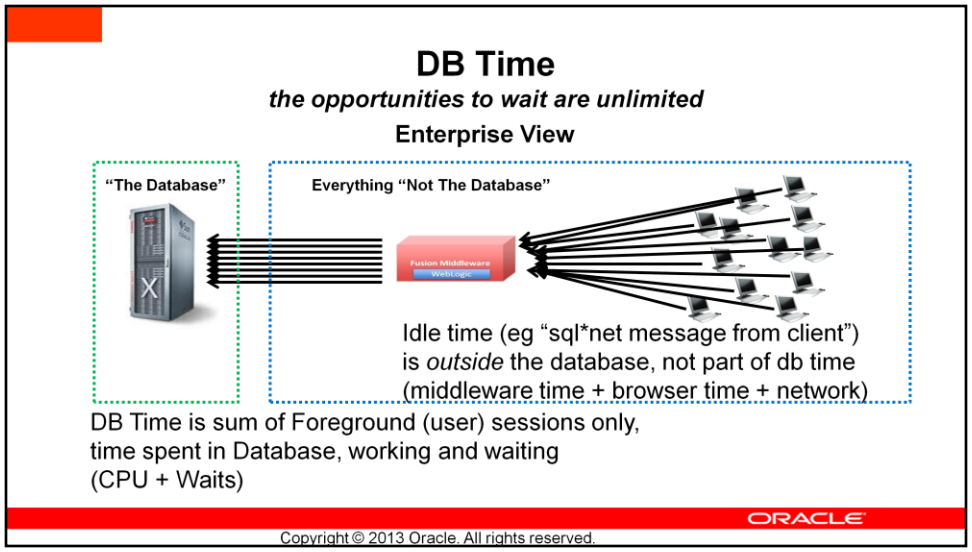
Response Model

response time = CPU time + wait time



Session time outside the database considered "idle", from viewpoint of server.

In the server, all sessions are either working (CPU), or waiting for a resource (lock, latch, pin, I/O, etc)



Response time can be measured from end user, through network and middleware, to the database and back again. Bottlenecks can exist at any point, but we don't always have the tools to measure time a request spends in the middleware application server, for example, versus the database. This lack of comprehensive instrumentation presents a bit of a dilemma for us– how do we know, for sure, where the problem is?

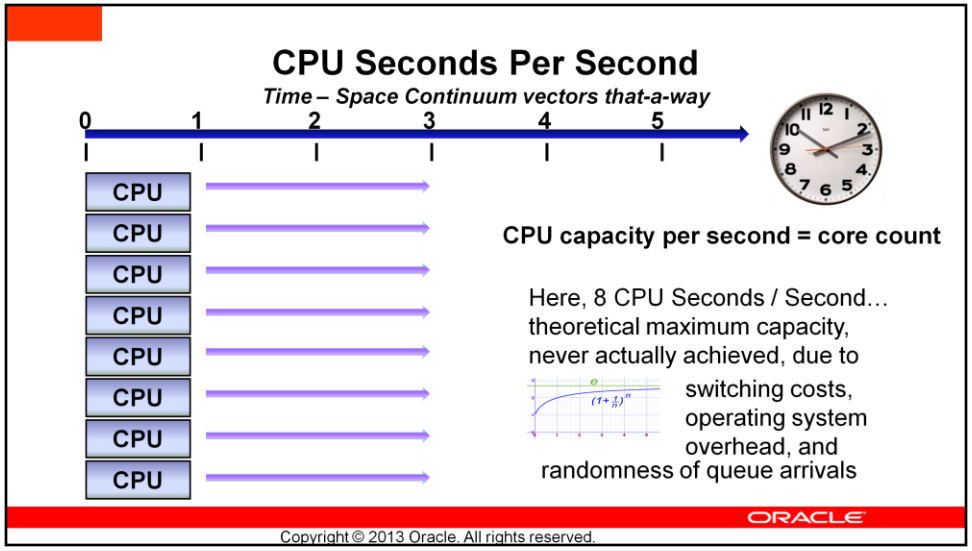
One way is to measure response time and requests through a single technology stack, and permit inferences with respect to the other stacks. That is, if we can establish, by comparing "good" versus "bad" response time, that the change primarily comes from the database, we can, without instrument the other components, ascertain our worst bottleneck is in the database server and not elsewhere. Conversely, if increased response time does not come from the database, then tuning the database will not solve to problem– we have to look at the other components.

Oracle AWR and ASH construct what we call the "db time model" for just this purpose. For a single session, or all sessions in aggregate, Oracle measures and records the times for each request that is spent actually in the database server, whether working (consuming CPU) or waiting for some resource. By viewing performance data through the lens of this db time model, we can quickly focus on the actual cause of a performance problem and get to a solution without undue delay.

As an example, suppose a single user experiences a normal response time for a given query of 5 seconds. Then, the time jumps to 25 seconds. If that extra time, by any measure, is spent inside the database, then certainly we have a database problem to solve. On the other hand, if we can determine 15 of the extra 20 seconds is spent outside the database, and only 5 of the extra seconds were spent inside the database, then attempting to tune that query within the database is pointless. The 15 needed seconds are simply not in the database, so we have to look elsewhere.

DB Time is the sum of CPU and wait times for all foreground sessions for activity inside the database. That time outside the database is referred to as "idle time". The term "idle" does not mean nothing is happening, only that, from the perspective of the database server, there is no active request to be processed. Hence "idle waits". The most common idle event is "sql*net message from client", which means the database server is waiting for a request to be sent by the client (either browser or middleware, depending on system architecture).

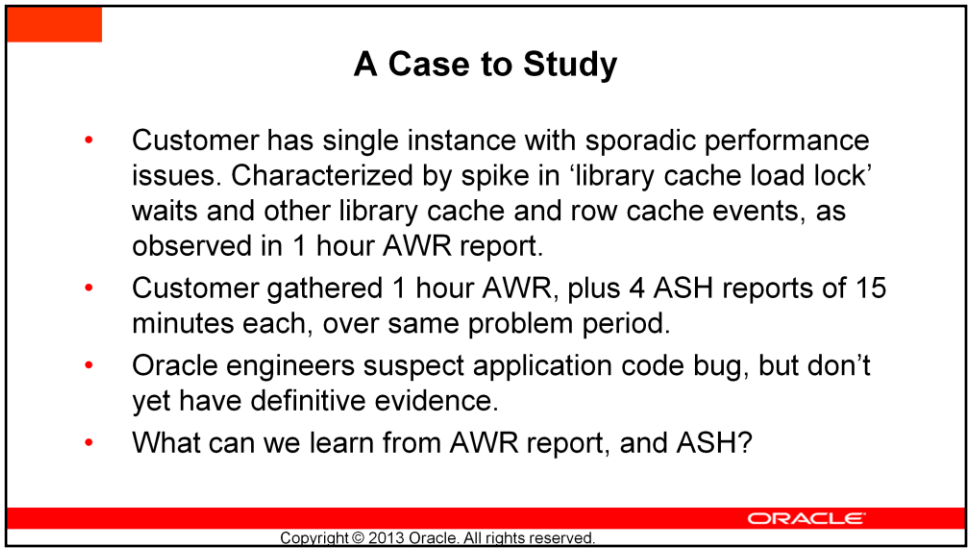
Note, DB Time is typically a value well in excess of the chronological "wall clock" time in the AWR or ASH sample interval.



Time, as reported in AWR, can be confusing. Bear in mind, while there is no specific limit to the number of sessions actively in the database (and perhaps waiting for database resources), there is a specific limit to CPU consumption. For each wall-clock second, each physical CPU can afford no more than one second of execution time. This is a theoretical maximum of CPU capacity– operating system overhead, switching costs and the queueing properties of concurrent systems assures us actual CPU consumption will be less than 100%.

Also consider Symmetric Multi-Threading (SMT) CPU architectures, sold under various names (eg Intel Hyperthreading), which provide for multiple execution threads for each physical CPU (core). In large scale servers, certain memory access operations take more than a single machine cycle, thus causing the CPU to occasionally wait for memory. SMT is designed to take advantage of that by permitting the waiting CPU to execute a different thread of program logic during that wait period, thus saving otherwise wasted CPU capacity. Now, this does not actually increase CPU capacity beyond the physical number of CPUs, it simply allows more total program execution to take place during otherwise idle periods, perhaps to a realistic 5-15 percent.

Still, cores are physical CPUs and determine the absolute limit to CPU capacity. While marketing literature may imply SMT increased CPU capacity, it does not. The limit is still core count. Unfortunately, AWR and ASH report thread count as CPU count, and this is incorrect. The CPU capacity of any server is the core count, and we will use that value in our analysis here.



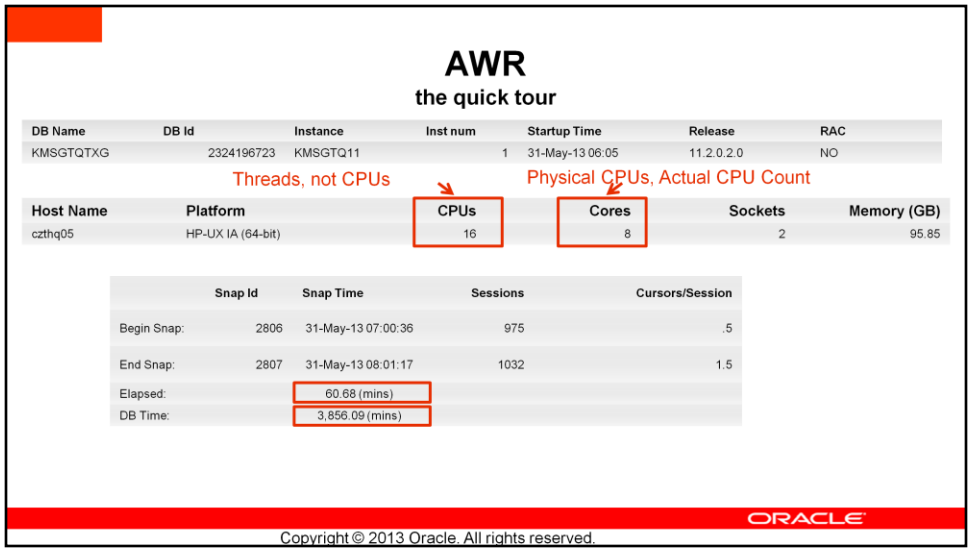
Fortunately, we have a case to study as we look at AWR and ASH. This will make it easier to sensibly understand the reports in a practical manner.

The case at hand is not purely a performance issue, but a bug. Many bugs, however, appear as performance issues in the beginning, so this is a reasonable approach. Most conventional performance issues manifest themselves as "bad SQL", or at least SQL with unusually high execution times to require some tuning.

Another factor making this case valuable is that the engineers involved gathered both AWR and ASH data for the same period, allowing us to compare and contrast both report tools and to get a more orthogonal view of the issue. Using both AWR and ASH is a great idea, as we shall see.

A final note on our approach, and the value of these tools: neither is a SQL tuning tool. They are useful for diagnosing performance issues and finding the probable cause. We look at the big picture, then down to a closer, session level view, then back to the big picture for our analysis. Our expectation is to identify some probable cause of our performance symptom so we can pursue the issue from that point to resolution. Most often we are going to find one or more SQL statements in need of tuning, and will follow this AWR/ASH effort by tracing one or more sessions using those cursors. Then we will use other tools to actually tune the SQL.

AWR and ASH are intended to get us to that bad SQL as quickly and directly as possible.



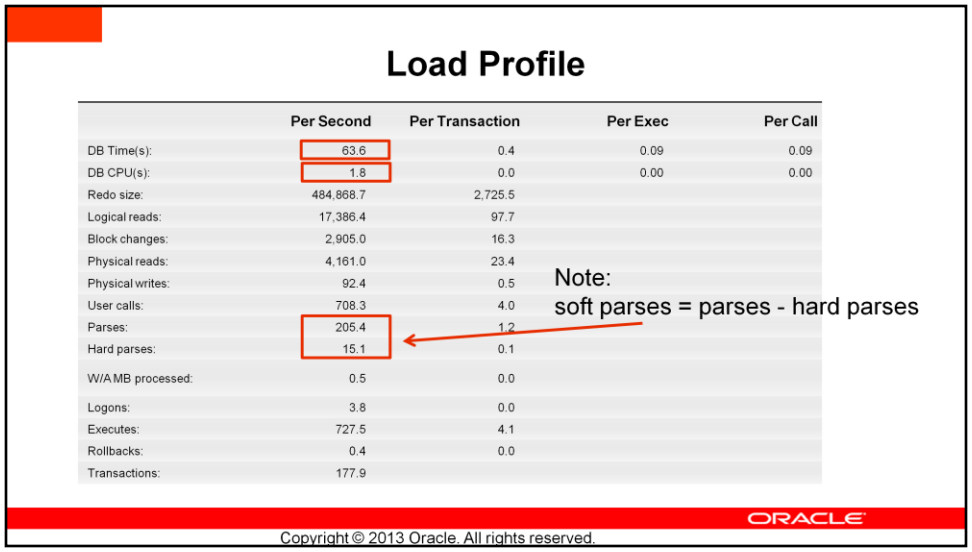
Now we will take a tour of an AWR report, taken for a one hour period.

This report is for an 11.2.0.2 database on a 64 bit platform with 8 CPUs and 16 threads, despite labeling to the contrary. The hardware is packaged with 4 cores per chip, resulting in a 2 socket configuration. The reason CPU count is higher then core count is due to Symmetric Multithreading (aka Hyperthreading on Intel hardware). For our analysis purposes, we will consider CPU count to be 8, not 16 as stated in the report.

Note the elapsed time of 60.68 minutes, and DB Time of 3,856.09 minutes. Elapsed time is “wall clock” time for the reporting interval, while DB Time is the sum of foreground session time spend in the database, either executing CPU instructions or waiting for a non-idle event, such as IO or a lock of some sort.

How is it possible for DB Time to exceed elapsed time? The answer lies in the understanding of available resources. We have 8 CPUs, which means we can, at most, consume 8 seconds of CPU per wall clock second. We never do, however, since some time is consumed by the operating system. And there is inevitably variances in load demand, so there will be times, even in busy periods, when there are fewer requests for CPU than CPU resources available.

As for wait times, while the capacity for doing work is constrained, the opportunities to wait are not. Hence, with many connected sessions waiting for internal resources to become available (and waiting for CPU), the aggregate DB Time is quite high, and is never specifically constrained.



Here we see the DB Time “per second” is 63.6. that means 63.6 seconds of time was spent by all foreground sessions, collectively, in the database, either working or waiting, for each wall clock second, on average for this one hour sample period.

CPU consumed was 1.8 seconds per wall clock second for the sample period. Considering we have 8 CPU seconds available per second, this machine is nearly idle. Well, if this is an OLTP instance, and since it is between 7 and 8 AM, this may not be surprising.

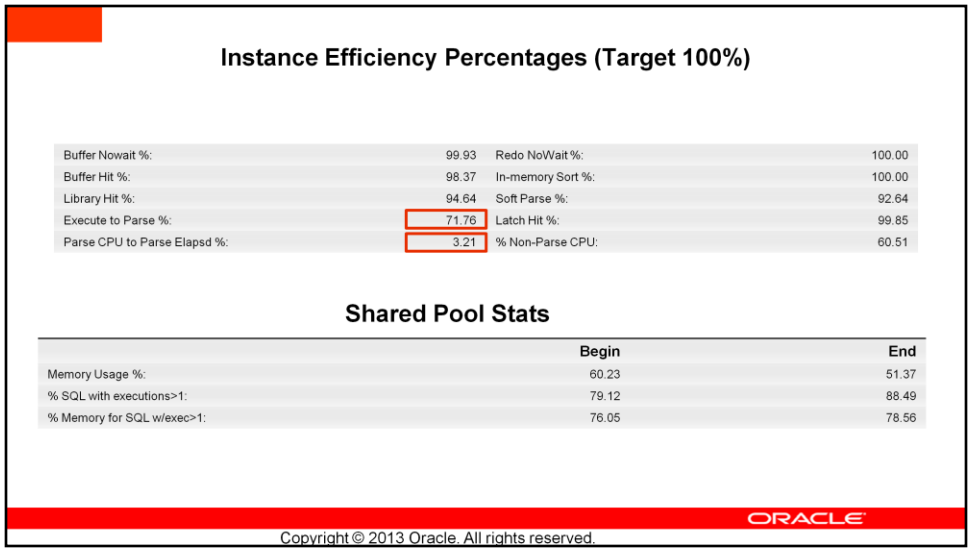
We also report data per transaction. Transaction is defined by commit boundaries, so commits per transaction will be 1, always.

Regarding “parses” versus “hard parses”, it may be useful to point out, the term “soft parse” is simply all parses, less hard parses. A parse, in its simplest (soft) form, is simply the effort of looking up a cursor or other executable-related object in the cursor cache, also known as the “library cache”. This memory store is a rather large hash table, used to store many objects, included cursors, PL/SQL objects, table and object references, etc. Conceptually, it resembles the sort of symbol table used by compilers during program compilation, and is “overloaded”. When a cursor is compiled, it is necessary to ensure dependent objects– tables, columns, user permissions, etc– exist in an appropriate form before execution of the cursor can be allowed. Once this is accomplished, the cursor is flagged as executable. As long as the cursor continues to be in an executable state, a new session can simply look up the cursor in the cache and “pin” it, then execute it without recompilation. This is a “soft parse”.

Obviously, soft parses are less time consuming than hard parses, but even soft parses are not free. Hence, the amount of parsing activity can be significant.

Lots if information is reported, and is useful for comparing busy to non-busy, and good to bad performance periods to assure we are comparing reasonably similar workloads.

Also, after a problem is solved, we might observe some numbers go down, such as logical or physical reads and block changes, while other numbers will go up, such as executions or user calls. Such differences can explain lower response time and simultaneous increased throughput.



Instance Efficiency Percentages (Target 100%)

Buffer Nowait %:	99.93	Redo NoWait %:	100.00
Buffer Hit %:	98.37	In-memory Sort %:	100.00
Library Hit %:	94.64	Soft Parse %:	92.64
Execute to Parse %:	71.76	Latch Hit %:	99.85
Parse CPU to Parse Elapsd %:	3.21	% Non-Parse CPU:	60.51

Shared Pool Stats

	Begin	End
Memory Usage %:	60.23	51.37
% SQL with executions>1:	79.12	88.49
% Memory for SQL w/exec>1:	76.05	78.56

ORACLE

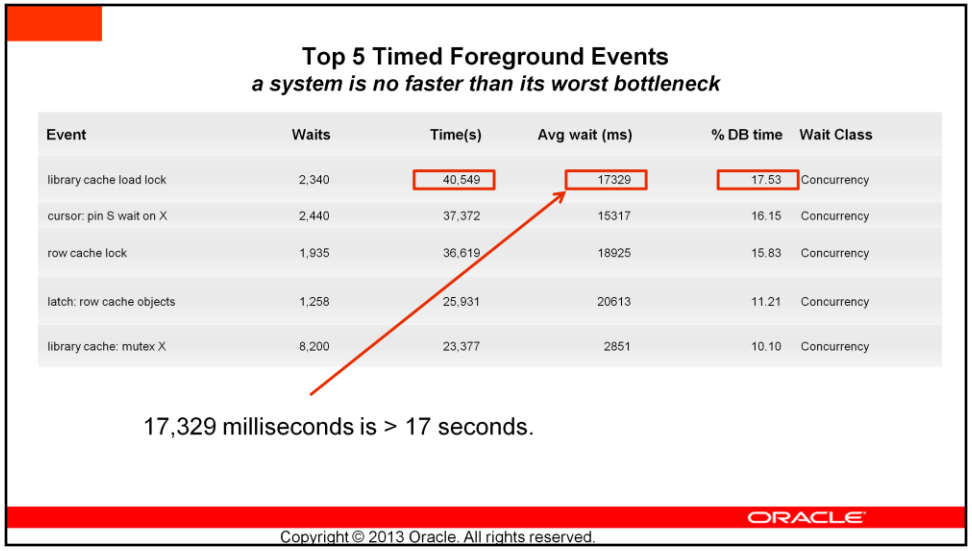
Some years ago, much was made of “ratio tuning” in which the user was told to keep adding memory to the server until a buffer hit ratio of nearly 100% was achieved. Unfortunately, it is entirely possible to have a high buffer hit ratio and still have horrible performance. The reason is, even though cache-based logical reads avoid physical IO, they are not free.

Still, it can be useful to at least consider changes in various ratios from good to bad performance. In particular, a low execute-to-parse percentage can indicate a poorly written application, that has to parse a cursor each time it wants to execute it. Ideally, a cursor will be parsed one time by the application, then executed many times before it is closed.

Continuing with our ongoing discussion of parsing, with the “execute to parse %” we can identify issues with repeatedly opening and closing cursors. When an application opens a cursor, that invokes a parse– at least a soft parse– in the database. That in turn will involve acquisition of a low level lock to protect the sessions access to the hash table structure that is the library cache. Since only one session at a time can access the cache for looking up a cursor, the underlying lock (either “library cache latch”, or “library cache mutex X” in 11g +) can become a significant bottleneck. To avoid this, applications need to open a cursor one time, then iteratively apply bind values and execute that cursor is needed for the application. The least desirable behaviour is for the application to immediately close the cursor after execution, then open it again before the next execution. So, we suggest “parse once, execute many” as a best practice, and this statistic gives you some idea of how the application is working in this respect.

Also, high Parse CPU to parse elapsed can indicate optimizer problems. Non-parse CPU means all other uses of CPU other than parsing cursors. This includes PL/SQL execution as well as buffer reading and manipulation, as well as looking up database block buffers in the buffer cache.

% memory for SQL w/ exec > 1 indicates memory used for cursors that are shared. Generally, higher is better, as is the case for % SQL with executions > 1.



In previous versions of Oracle (9.2 and earlier) we identified the top five wait events. As the DB Time model was developed, this was expanded to include CPU consumption by foreground sessions. Since CPU is not a wait event, we now call this the top five timed foreground events. Even under previous versions it was possible to make use of the DB Time model, but you had to construct it yourself by including “cpu used by this session”, summed for all sessions, and recalculating wait events to obtain the time model values.

Having this done for you is much easier, of course.

Note the Time(s), which is time of event total, expressed in whole seconds, and Avg wait (ms), the average time of the event expressed in milliseconds. We also get a report of each event as a percentage of DB Time, which is important for analyzing possible bottlenecks and their relative contribution to performance problems.

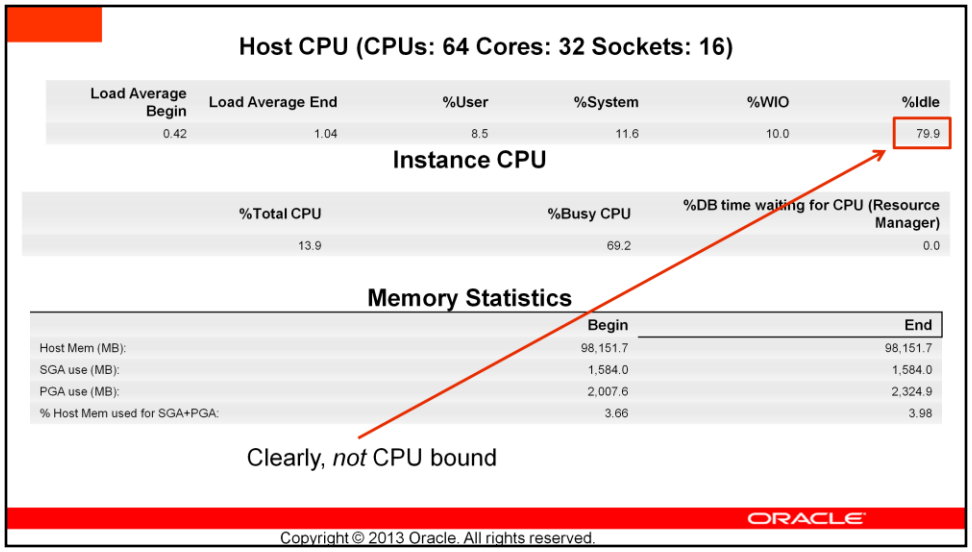
The method here is to consider the worst bottleneck first, and ignore lesser events. If you are trying to recapture 100 seconds of performance, then tuning based on an event that has a cumulative wait time of, say, 20 seconds is not going to help you. The time simply isn’t there.

Average wait time has to be taken in context. In particular, you need some understanding of what a particular wait event represents. That takes time and research, but needs to be done. In this case, “library cache load lock” means a cursor is being constructed (hard parsed) by some session, and other sessions wanting that exact same cursor must wait until it is ready. This is analogous to “read by other session” buffer busy waits in the buffer cache.

We shall examine this further to get an understanding of what is going on, by reviewing the distribution of wait times.

Also, the other top events all seem to be parse related. What is significant here is that cursor execution / elapsed time is not involved. So, the normal sort of performance data is not in this set of top events– such as “db file sequential read”, “db file scattered read”, “DB CPU” etc. So, perhaps our performance issue is really some sort of bug issue at its core.

Many bug discoveries start out as performance issues, so this exercise is nevertheless quite valuable for our educational

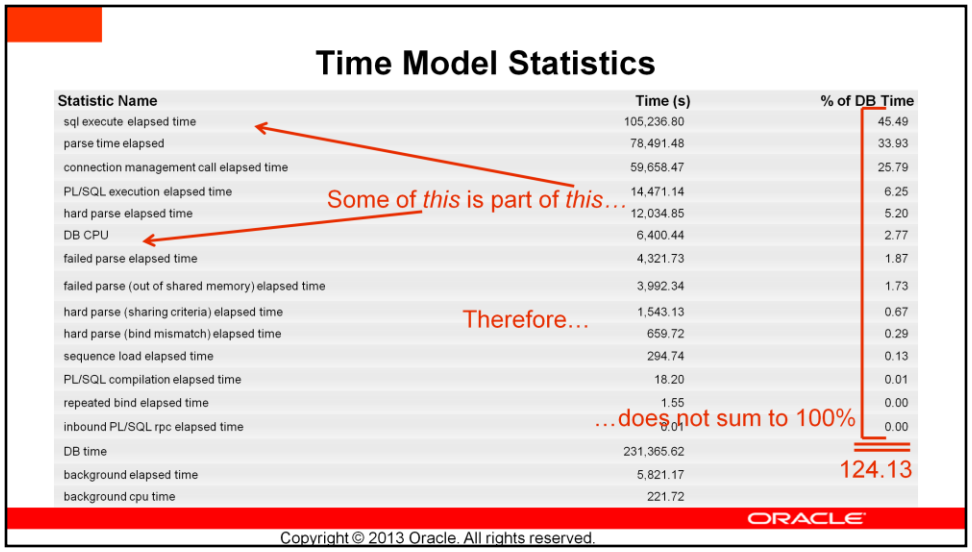


In 11g the reporting of operating system data has been improved dramatically. We can see the relative values of % user versus % system. In a healthy, busy system, we would expect most of our time to be consumed by user processes and application calls, so % user ought to be relatively high. Here it is less than % system, but since this is a nearly idle system, normal operating system housekeeping tasks are still performed, so it is unsurprising the percentage of total CPU consumption by these processes is higher than application CPU consumption. It would appear, most processes are in a suspended state, and thus are not consuming CPU anyway. What little activity on the machine exists, comes mostly from the operating system's activities.

If you have O S Watcher installed (Note 301137.1, an excellent free tool), you can sanity check these values. The right hand column of the vmstat report on most platforms shows %idle, which should approximate the value in this AWR section for the same sample periods.

%Busy CPU is the amount of machine CPU resources consumed by Oracle, for the time CPU was consumed. That is, Oracle is using about 13.9% of CPU resources that are consumed overall, irrespective of the machines idle CPU time. Presumably this machine is dedicated to Oracle, so the rest of the time was consumed by the operating system.

Also reported are memory usage values, as of the beginning and end of the sample period.



The time model statistics represent a sort of denormalized view of our elapsed time. Since DB Time is the sum of CPU time and all foreground non-idle waits, we can get a pretty good understanding of where our time was spent.

Some of these values represent the sum of other DB Time components, so the columns will not add up to DB Time. For example, sql execute elapsed time includes CPU used for SQL execution, as well as assorted wait events such as IO.

If we have a case were the top wait event is something like db file sequential read, and sql execute elapsed time dominates DB Time, then we know to look at SQL statements sorted by physical reads as well as elapsed time.

If CPU was the top timed event, and sql execute elapsed time was the top DB Time component, we would look at SQL statements not only by elapsed time, but CPU consumption.

More on this in the next pages.

Note DB Time is itself reported here, which obviously makes up 100% of DB Time. Also, we report background process elapsed times and CPU consumption, but since they do not affect our response time directly, they are not included in DT Time. We ignore background process times and events.

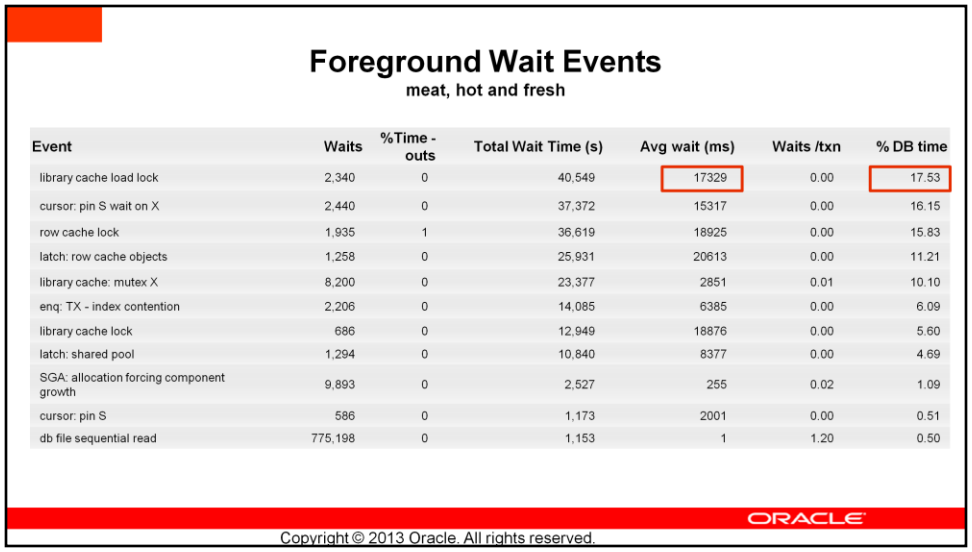
Operating System Statistics

from os, sanity check with OS Watcher, etc.

Statistic	Value	End Value
AVG_BUSY_TIME	59,677	
AVG_IDLE_TIME	237,208	
AVG_IOWAIT_TIME	29,522	
AVG_SYS_TIME	34,455	
AVG_USER_TIME	25,100	
BUSY_TIME	956,607	
IDLE_TIME	3,797,126	
IOWAIT_TIME	474,138	
SYS_TIME	553,176	
USER_TIME	403,431	
LOAD	0	1
OS_CPU_WAIT_TIME	569,163,000	
VM_IN_BYTES	4,288,512	
VM_OUT_BYTES	7,434,240	
PHYSICAL_MEMORY_BYTES	102,919,471,104	
NUM_CPUS	16	
NUM_CPU_CORES	8	

ORACLE

More operating system statistics, reported as total values for the sample period. For most, the Value column applies, as it is just a total for the period. For “load”, we show beginning and ending values.



Foreground Wait Events

meat, hot and fresh

Event	Waits	%Time - outs	Total Wait Time (s)	Avg wait (ms)	Waits /txn	% DB time
library cache load lock	2,340	0	40,549	17329	0.00	17.53
cursor: pin S wait on X	2,440	0	37,372	15317	0.00	16.15
row cache lock	1,935	1	36,619	18925	0.00	15.83
latch: row cache objects	1,258	0	25,931	20613	0.00	11.21
library cache: mutex X	8,200	0	23,377	2851	0.01	10.10
enq: TX - index contention	2,206	0	14,085	6385	0.00	6.09
library cache lock	686	0	12,949	18876	0.00	5.60
latch: shared pool	1,294	0	10,840	8377	0.00	4.69
SGA: allocation forcing component growth	9,893	0	2,527	255	0.02	1.09
cursor: pin S	586	0	1,173	2001	0.00	0.51
db file sequential read	775,198	0	1,153	1	1.20	0.50

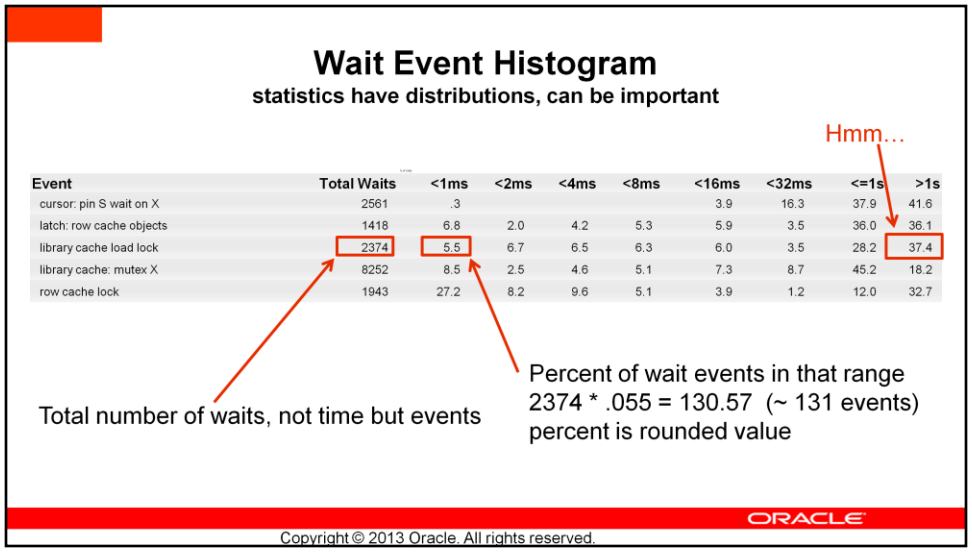
ORACLE

Here are the details of wait events, not including CPU time. They are ordered by total wait time and we need to pay attention to total wait time as well as average wait time.

Some events naturally take longer than others. So, IO events typically take milliseconds, while latch waits should be much less. If we see a latch wait that takes, on average, more time than IO events, then we might be suspicious of being CPU bound for some period.

As always, performance symptoms are to be understood in the context of the entire database. A particular wait event might not be a top time consumer, but could indicate a contribution to problems if it is unexpected. “SGA: allocation forcing component growth” means auto-tune SGA has encountered a 4031 memory error and MMAN is transferring memory. This could be a contributing factor to our issue, so it is worth noting for possible My Oracle Support searching later.

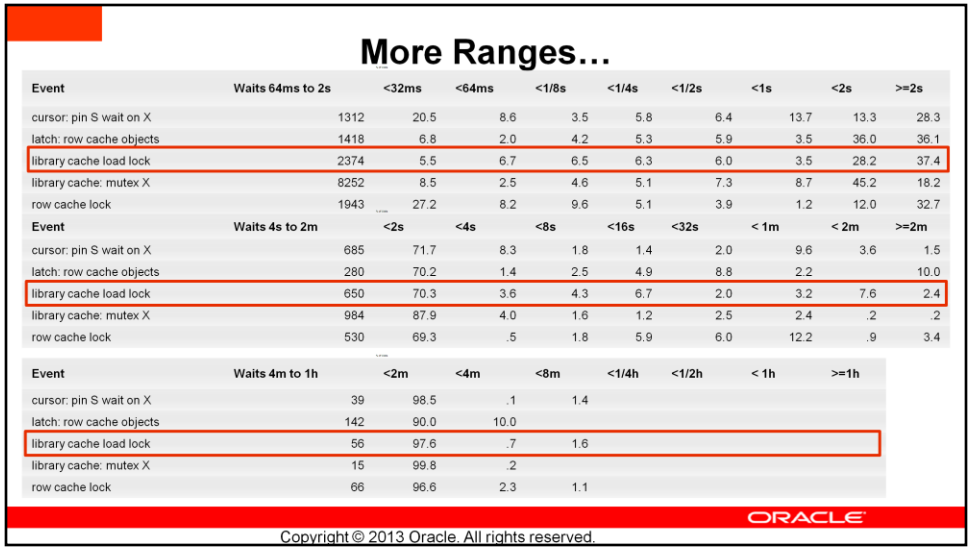
When something happens at the operating system level to consume masses of CPU resources, latch and mutex wait times can be exaggerated dramatically. In such cases we need to address the underlying cause before we consider the latch or mutex waits as reliable symptoms.



Wait events are now broken down by range of wait time, to get a histogram view of the distribution of times for each event. We normally expect a skewed distribution of times, clustering around the average. The number of statistical outliers should remain fairly low in a normal, busy system. However, under duress, the distribution becomes somewhat flatter, and the tail of outliers higher and longer. Here, we can observe, for our predominant wait event, “library cache load lock”, 37.4 waits are greater than or equal to 1 second. This represents the percentage of events in that time range, not the wait time overall. We’ll study this more closely on the next two pages.

This histogram lets you estimate and analyze intuitively, the skewness of wait time distribution. You might not use this as a main focal point for AWR analysis, but as you get used to the report and become more proficient at performance analysis, this section will start to be appreciated more.

Additional sections include broader ranges of wait values, but have the same essential meaning and purpose.

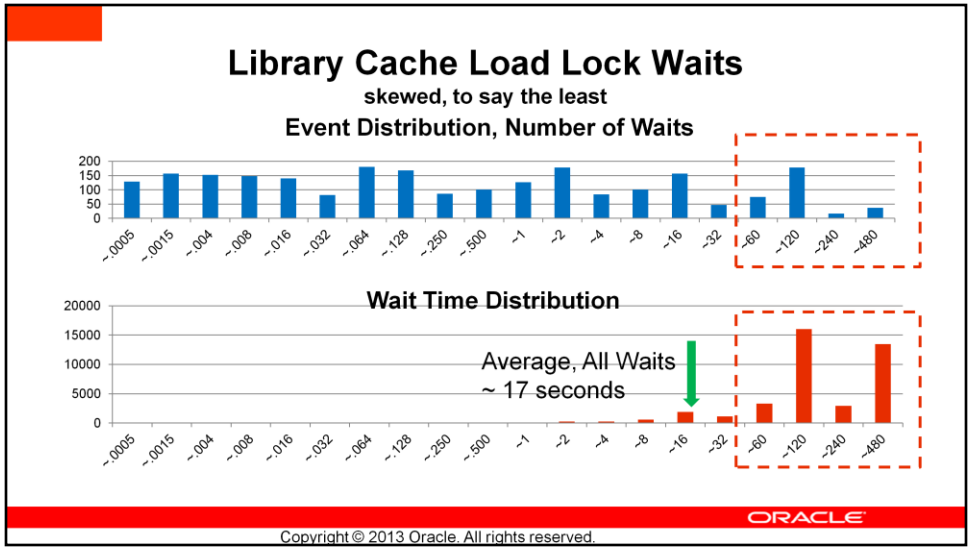


Three additional sections break down the range of waits, each bucket to the right representing the percentage of wait events for that bucket's time range. And, each bucket to the right represents double the time waited from the previous bucket. So, the entire scale is non-linear. And, since the values relate to number of events, we have to extrapolate the actual time weighting of each bucket. The time waited for each range of values is illustrated on the next page, but you really don't have to go through this exercise normally. You can simply infer, if a greater number of events is to the right of the overall average (~17 seconds, recall), then our real blocking and blocked sessions have much higher values than that average.

We deduce then, there is a subset of sessions involved in wait times far more extreme than the 17 seconds already reported as the average for library cache load locks. This might be taken as further evidence of a bug, since normal parsing simply does not take that much time. However, can cannot confirm our suspicion with this evidence alone.

Since our event represents sessions being blocked by a parsing session, we really need to know what the blocker is doing to find out what the problem is. Conventional sql tracing is unlikely to reveal this, since sql tracing reports actual sql execution, and parsing is simple a step in that process. Other than waits incurred by the parse process, no parsing details are reported. We will probably be better served with a processstate dump, including an errorstack report, to analyze our problem further.

However, for "normal" performance issues, sql trace is most often exactly what you need, from sessions incurring the worst bottleneck: our top wait event. More on this later.

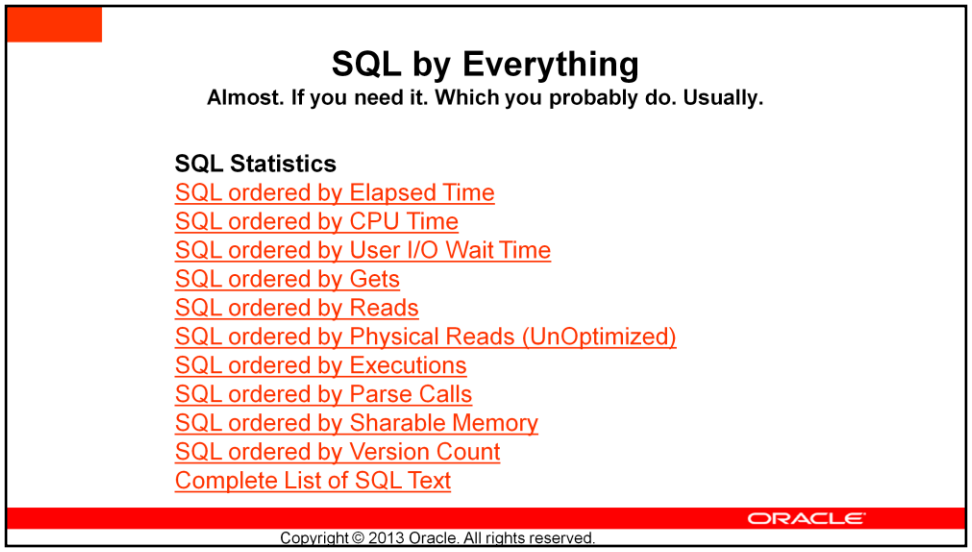


To get the distribution of wait times, we take the percentage of events for a given bucket and derive the number of events for that bucket. Then we multiply the number of events by the average time for that bucket. We cannot be entirely accurate, since there will likely be a skewed distribution of wait times within that bucket, but all we are interested in is a reasonable estimate.

As stated earlier, this is really not necessary for normal performance analysis, but seemed an interesting and illustrative exercise here. As you can see on the two charts, a subset of all sessions account for that majority of wait times. The average overall wait time, about 17 seconds, is marked by the green arrow in the bottom chart.

All too often, for low level events such as this and various latches, the time the blocker holds the resource is so short, even in fairly extreme cases, we cannot locate and dump trace for the blocking session before the wait is complete. In this case, however, typical wait times are well above one minute, so an attempt to gather detailed trace diagnostics on the blocking event seem achievable.

More on this as we proceed, later.

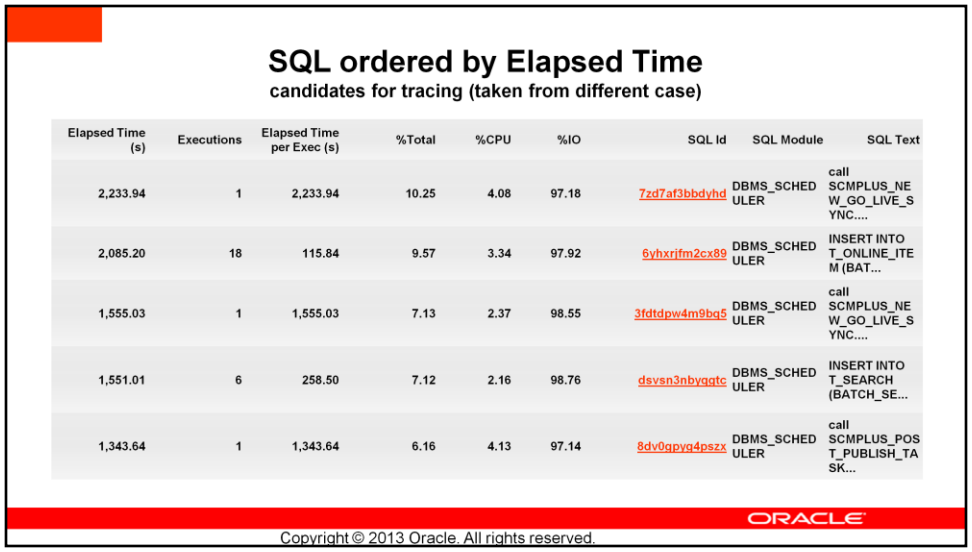


This is the typical “target” for AWR performance analysis. Once we know the overall nature of a performance problem, as surmised by analyzing top timed events and the composition of DB Time, and we have determined the operating system is functioning reasonably well, we most often want to find a session to trace to get to the nub of the problem.

SQL is ordered by elapsed time, logical reads (gets), physical IO, parses, CPU consumed, etc.

Since we are most often interested in finding non-performing SQL, this section is of particular interest. If, from our top timed events we see sql related events, and in our db time model section we observer sql execution time as the main db time consumer, then we probably have one or more cursors causing our problem. Simply review the relevant list, indentify the sql id and you are ready to trace. Almost. We still need to identify a session using that SQL, then turn on trace, perhaps using the EM console.

First, let’s look at the contents of the various “top SQL” lists.



SQL ordered by Elapsed Time

candidates for tracing (taken from different case)

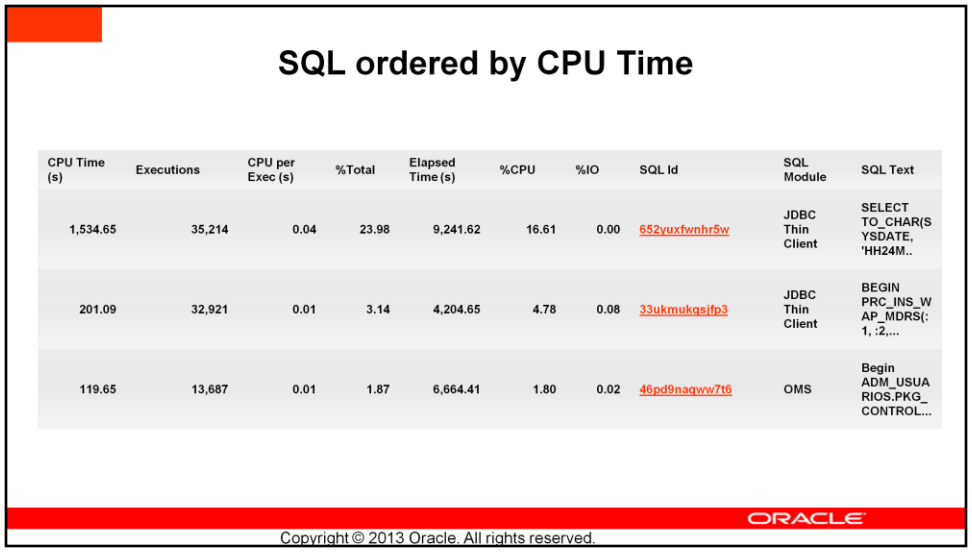
Elapsed Time (s)	Executions	Elapsed Time per Exec (s)	%Total	%CPU	%IO	SQL Id	SQL Module	SQL Text
2,233.94	1	2,233.94	10.25	4.08	97.18	7zd7af3bbdyhd	DBMS_SCHEDULER	call SCMPLUS_NEW_GO_LIVE_SYNC....
2,085.20	18	115.84	9.57	3.34	97.92	6yhxrjfm2cx89	DBMS_SCHEDULER	INSERT INTO T_ONLINE_ITEM (BAT...
1,555.03	1	1,555.03	7.13	2.37	98.55	3fdtdpw4m9bq5	DBMS_SCHEDULER	call SCMPLUS_NEW_GO_LIVE_SYNC....
1,551.01	6	258.50	7.12	2.16	98.76	dsvsn3nbyqqtc	DBMS_SCHEDULER	INSERT INTO T_SEARCH (BATCH_SE...
1,343.64	1	1,343.64	6.16	4.13	97.14	8dv0gpyg4pszx	DBMS_SCHEDULER	call SCMPLUS_POST_PUBLISH_TASK...

ORACLE

Here we have the first sort of the SQL, by elapsed time. We see the total seconds for each statement, and time per execution, as well as %total of elapsed time contributed by this statement, and then %CPU and %IO consumed. In this case, the top elapsed time SQL cursor consumed just over 10% of all elapsed time. Checking back to the DB Time page, we note total SQL Execution elapsed time was 21,673.03 seconds. At 2233 seconds, this statement represents a bit over 10% of that total, so if we really had a performance problem, this would be a good candidate for further analysis.

To do that, we use the SQL Id, find a session executing it, then trace that session at level 12 for some reasonable period of time. Use tkprof to analyze the resulting trace file and study the time data and plan for it, then tune the statement.

Actual SQL tracing and analysis is a separate topic for discussion.

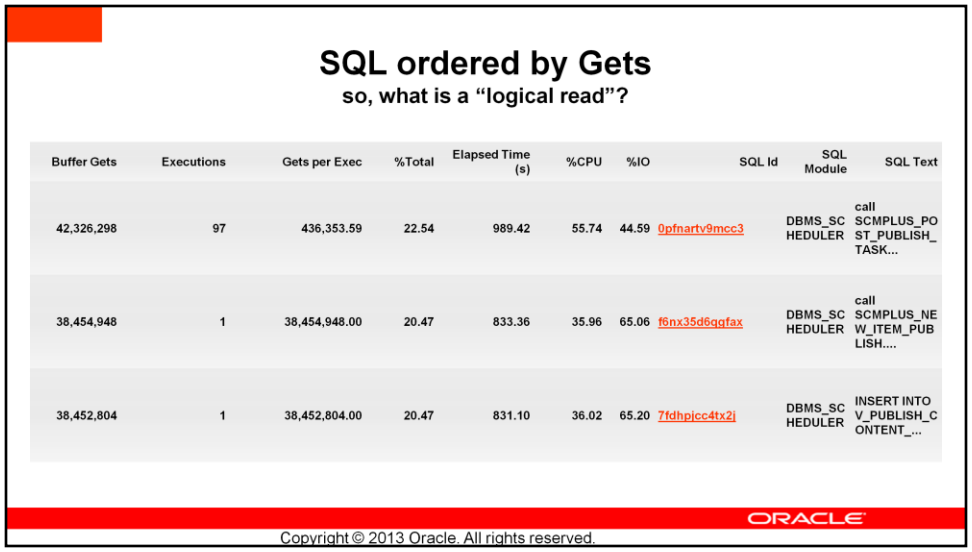


When the top timed event is DB CPU and the top DB Time component is SQL Execution, then it is likely the top SQL by CPU Time will reveal the main candidates for SQL Tuning. Here, the top SQL has a per execution consumption of only .04 seconds, but with many executions it is a heavy overall contributor to performance problems.

SQL ordered by User I/O Wait Time

User I/O Time (s)	Executions	UIO per Exec (s)	%Total	Elapsed Time (s)	%CPU	%IO	SQL Id	SQL Module	SQL Text
2,170.84	1	2,170.84	22.65	2,233.94	4.08	97.18	7zd7af3bbdyhd	DBMS_SCHEDULER	call SCMPLUS_NEW_GO_LIVE_SYNC....
2,041.91	18	113.44	21.30	2,085.20	3.34	97.92	6yhxrjfm2cx89	DBMS_SCHEDULER	INSERT INTO T_ONLINE_ITEM (BAT...
1,532.48	1	1,532.48	15.99	1,555.03	2.37	98.55	3fdtdpw4m9bq5	DBMS_SCHEDULER	call SCMPLUS_NEW_GO_LIVE_SYNC....

ORACLE



SQL ordered by Gets

so, what is a "logical read"?

Buffer Gets	Executions	Gets per Exec	%Total	Elapsed Time (s)	%CPU	%IO	SQL Id	SQL Module	SQL Text
42,326,298	97	436,353.59	22.54	989.42	55.74	44.59	0pfnartv9mcc3	DBMS_SCHEDULER	call SCMPLUS_POST_PUBLISH_TASK...
38,454,948	1	38,454,948.00	20.47	833.36	35.96	65.06	f6nx35d6qqfax	DBMS_SCHEDULER	call SCMPLUS_NEW_ITEM_PUBLISH....
38,452,804	1	38,452,804.00	20.47	831.10	36.02	65.20	7fdhpjcc4tx2j	DBMS_SCHEDULER	INSERT INTO V_PUBLISH_CONTENT_...

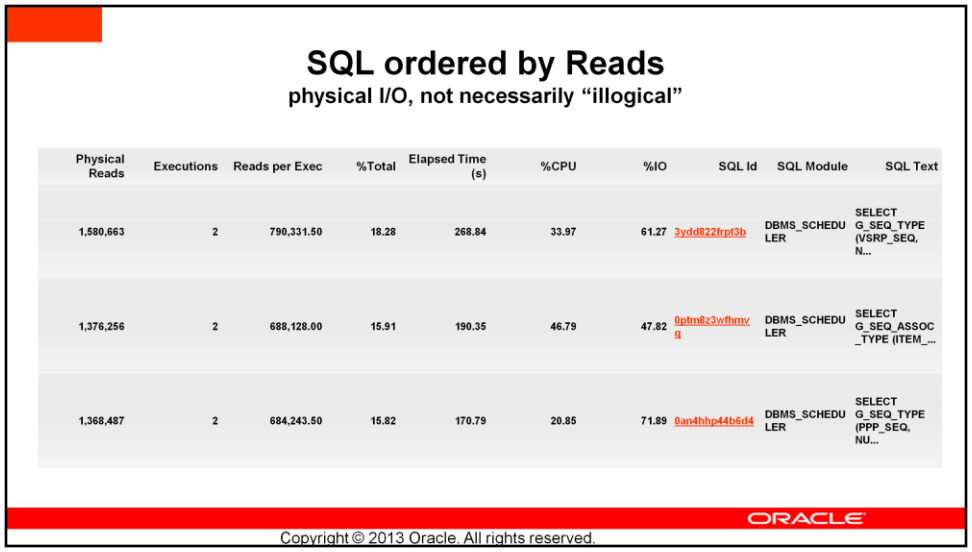
ORACLE

Logical reads are data block buffer reads that occur through the buffer cache. That is to say, we look in the buffer cache for a particular block. If we do not find it, we do a physical read of disk and place the block in the buffer cache for future use.

Logical reads may involve physical IO in this way, but do not necessarily do so. We also support direct reads, which avoid the buffer cache by reading directly from disk. Direct reads can be a performance booster in cases where an object is rarely found in the buffer cache, but generally logical, buffered reads are the best solution since this helps avoid unnecessary IO.

Nevertheless, logical reads, even when the block is found in the buffer cache, are not free. So, a poorly tuned statement, or a weak, non-selective index can dreadfully hamper performance. The typical wait events symptomatic of excess logical read are "cache buffers chain latch" waits and "buffer busy waits", along with the usual "db file sequential read" and "db file scattered read" I/O events. "cache buffers chain latch" and "buffer busy wait" events are associated with locking mechanisms used in concert to provide concurrent access to the buffer cache while ensuring consistent, non-corrupt reads. Moreover, logical reads also consume CPU.

A good tactic for finding offending SQL when either user I/O, CPU or both are present as top timed events is to review SQL order by Elapsed Time, CPU, Gets and User I/O, looking for SQL present in multiple sections. Otherwise, for I/O as a top event, look to a combination of Elapsed time and User I/O. Or, where CPU is a top event, SQL by Elapsed time and by CPU.



SQL ordered by Reads

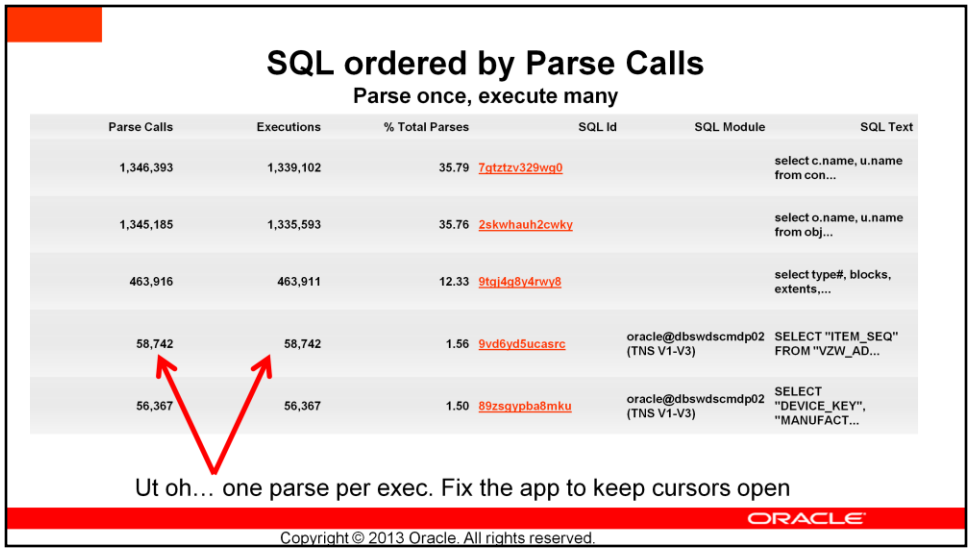
physical I/O, not necessarily "illogical"

Physical Reads	Executions	Reads per Exec	%Total	Elapsed Time (s)	%CPU	%IO	SQL Id	SQL Module	SQL Text
1,580,663	2	790,331.50	18.28	268.84	33.97	61.27	3ydd822frpt3b	DBMS_SCHEDULER	SELECT G_SEQ_TYPE (VSRP_SEQ, N...
1,376,256	2	688,128.00	15.91	190.35	46.79	47.82	0ptm8z3wfhmvg	DBMS_SCHEDULER	SELECT G_SEQ_ASSOC_TYPE (ITEM_...
1,368,487	2	684,243.50	15.82	170.79	20.85	71.89	0an4hhp44b6d4	DBMS_SCHEDULER	SELECT G_SEQ_TYPE (PPP_SEQ, NU...

ORACLE

And if the top timed events include such events as "db file scattered read" or "db file sequential read", we would want to know the top User I/O SQL. Again, it would probably be the case the top elapsed time SQL would match all or part of this list, thus confirming our suspicions. In this case, the SQL only had one execution during the AWR report interval, but consumed 22.65% of all foreground I/O.

SQL ordered by User I/O Wait Time can include direct path reads and writes to temp storage, in addition to more common I/O waits.

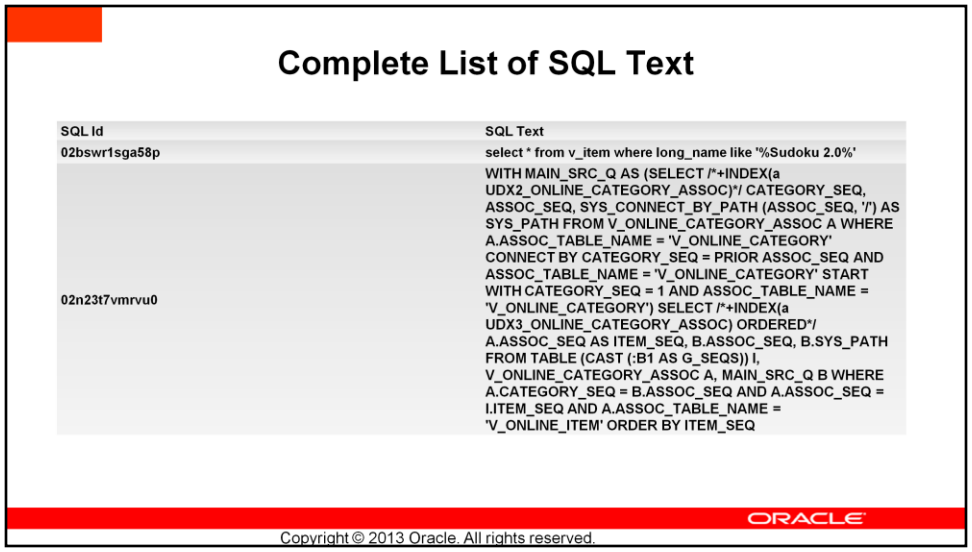


SQL ordered by parse calls also provides execution counts for comparison. The top cursors here appear to be “recursive calls”, or SQL generated by Oracle for the parsing process. But, it also appears some user SQL is listed, and we see parse calls equal to executions for them. Not a good thing, if library cache latch or mutex waits are top timed events, and performance problems are reported.

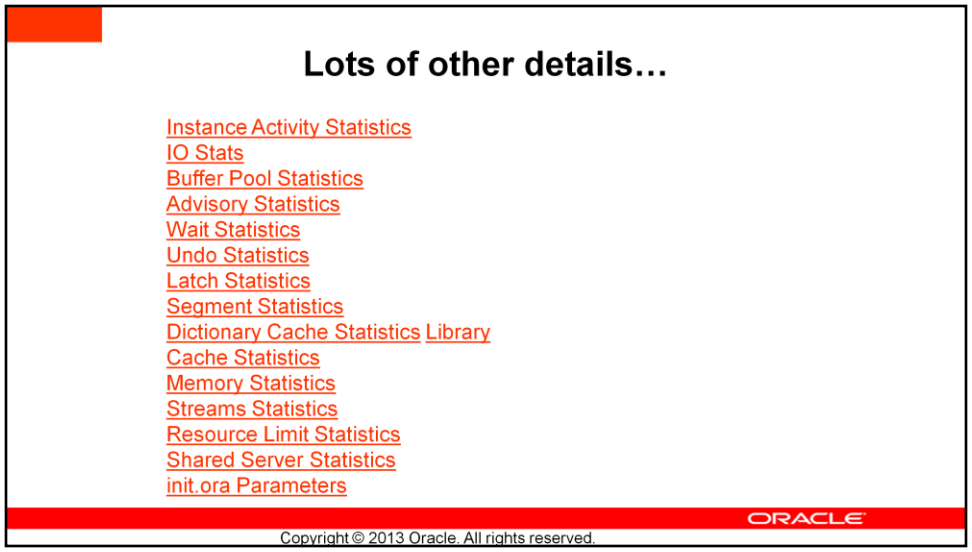
Recursive SQL is always parsed for each execution, and cannot be tuned. But application SQL needs to avoid this situation, and some coding effort is usually required. Of course, application developers don’t want to recode their applications, so a business decision is needed.

The optimal case is “parse once, execute many”. Even though these cursors are shared, and the parses are “soft”, they still are not free.

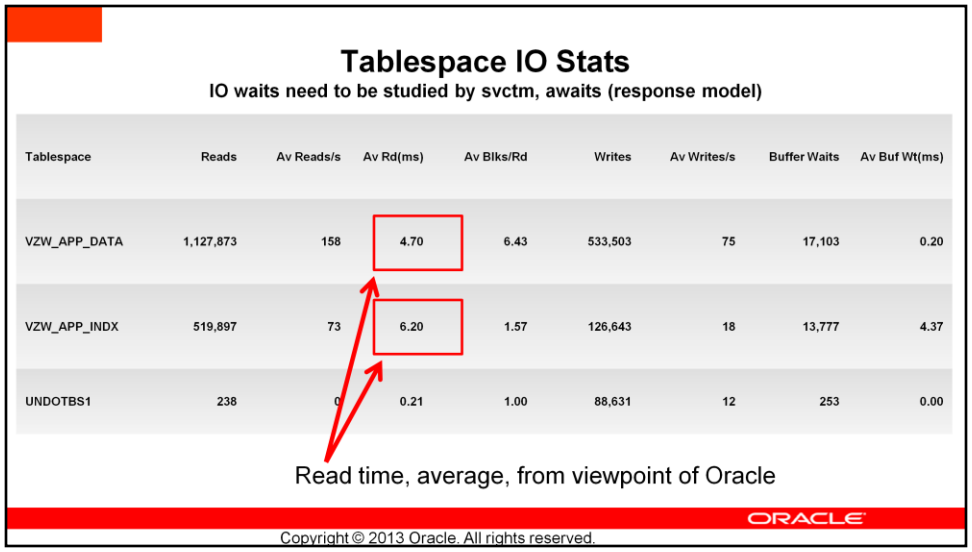
Of course, in this case, the database is nearly idle so this may be no big deal. Still, something to watch for and observe when library cache problems are present, as indicated by “library cache latch” or “library cache mutex X” top events are present.



For reference and validation.

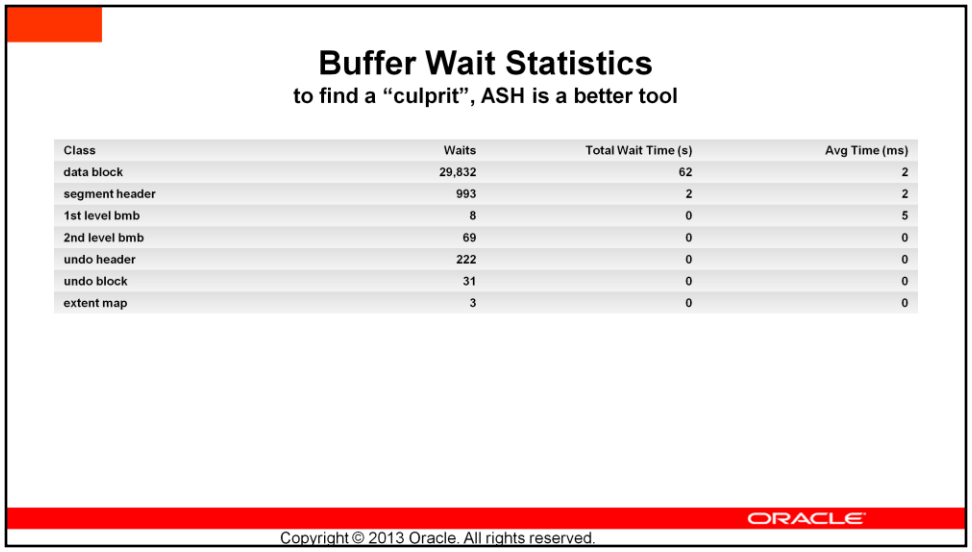


We cannot document and discuss every detail of AWR here, but go over some of the main pieces that are frequently helpful with performance issues.



I/O, from Oracle's perspective, is the time it takes to return from an I/O request to the operating system. Oracle has no way of characterizing the event in detail, whether the bottleneck is a slow disk device, a controller, or simply congestion from poorly balanced disk workload. You need to get details from your storage administrator, with information equivalent to "iostat –x" as available on Linux for attached devices.

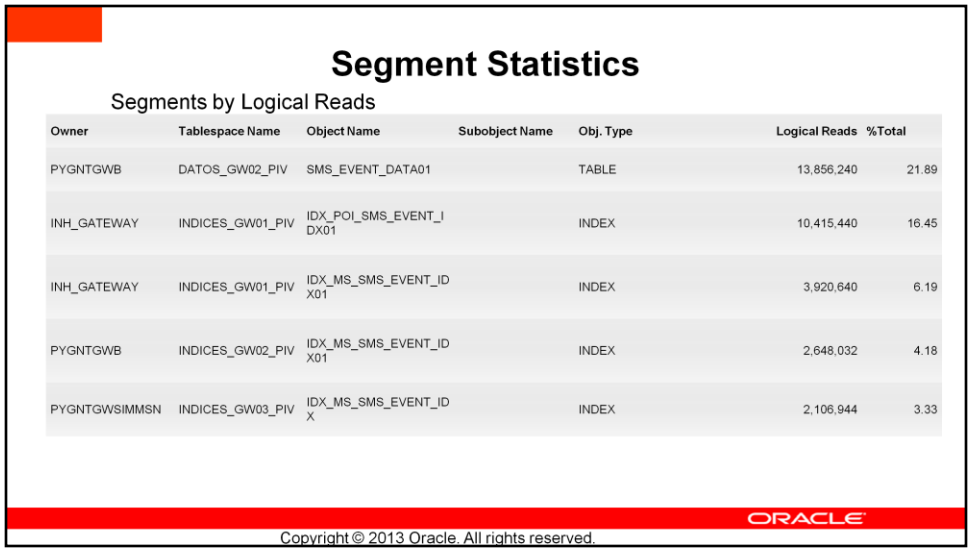
This would should you the "svctm" and "await" values for each device. Svctm is the actual disk device time to fetch an IO from disk, while await is the overall disk response time, including time spent in the request queue. Recall the response model at the beginning of this discussion for the implications of these data.



Buffer waits are related to logical reads (via the buffer cache), in which congestion is symptomized by either high cache buffers chains latch waits, or buffer busy waits. Both are concurrency control lock structures. If you see high buffer busy waits in to top five timed events, this section will allow you to characterize the issue for further investigation.

Data blocks include tables as well as indexes. However, buffer congestion on each has a different approach. Under blocks are simply rollback segments, while undo headers are the header blocks of rollback segments and are also known as transaction tables.

If you are dealing with high buffer busy waits, consider using ASH to identify the culprit session, SQL and objects, directly. Much quicker than AWR for this purpose. Sort of "cuts to the chase".



Segment Statistics

Segments by Logical Reads

Owner	Tablespace Name	Object Name	Subobject Name	Obj. Type	Logical Reads	%Total
PYGNTGWB	DATOS_GW02_PIV	SMS_EVENT_DATA01		TABLE	13,856,240	21.89
INH_GATEWAY	INDICES_GW01_PIV	IDX_POI_SMS_EVENT_IDX01		INDEX	10,415,440	16.45
INH_GATEWAY	INDICES_GW01_PIV	IDX_MS_SMS_EVENT_IDX01		INDEX	3,920,640	6.19
PYGNTGWB	INDICES_GW02_PIV	IDX_MS_SMS_EVENT_IDX01		INDEX	2,648,032	4.18
PYGNTGWSIMMSN	INDICES_GW03_PIV	IDX_MS_SMS_EVENT_IDX		INDEX	2,106,944	3.33

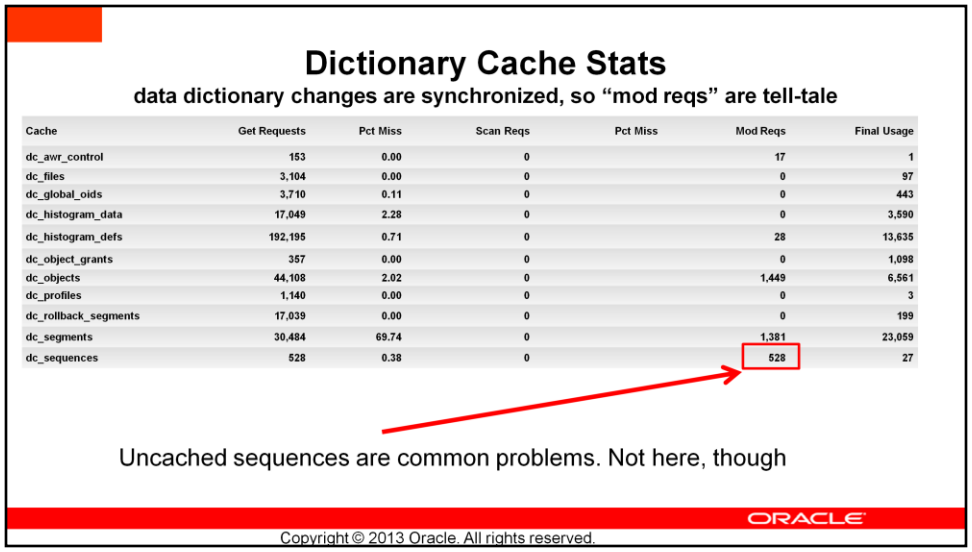
ORACLE

Details of segment names and types by wait event can be extremely helpful in tracking down all sorts of wait event-induced bottlenecks. Here is one example– logical reads. The full list of segment sort details is on the next slide.

Segment Statistics

- Segments by Logical Reads
- Segments by Physical Reads
- Segments by Physical Read Requests
- Segments by UnOptimized Reads
- Segments by Optimized Reads
- Segments by Direct Physical Reads
- Segments by Physical Writes
- Segments by Physical Write Requests
- Segments by Direct Physical Writes
- Segments by Table Scans
- Segments by DB Blocks Changes
- Segments by Row Lock Waits
- Segments by ITL Waits
- Segments by Buffer Busy Waits

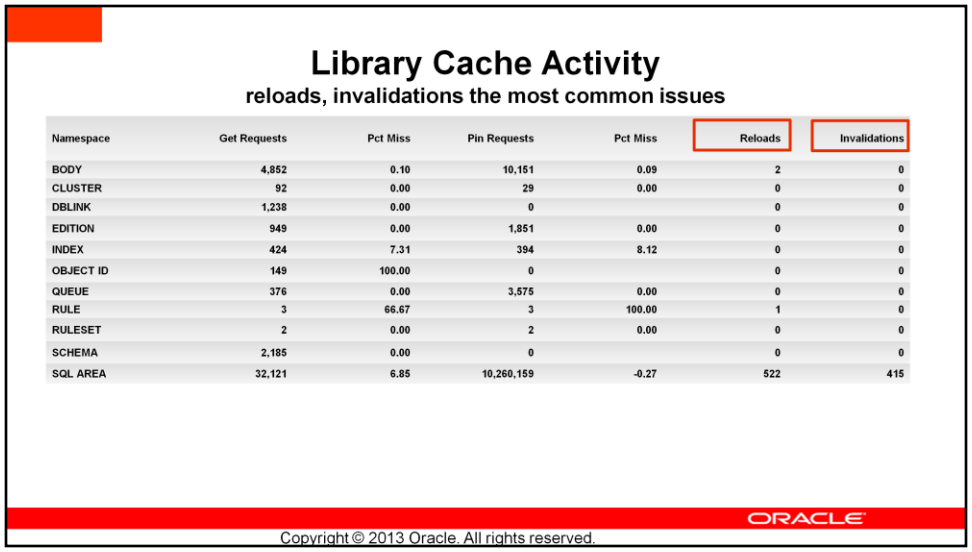
ORACLE



The data dictionary cache, also known as the “row cache”, is used to cache metadata used during cursor parsing and syntactic element resolutions.

When DDL is executed and an object is changed, such as adding a column to a table, the change not only invalidates all dependent cursors but also must be written back to the dictionary object tables on disk before work continues, to protect changes. So, heavy DDL can cause contention here.

Sequences are also stored in the data dictionary, but have a specific cache mechanism to defer that I/O. If cache is set low on a sequence, then congestion can occur and will typically be characterized by high mod request values here.

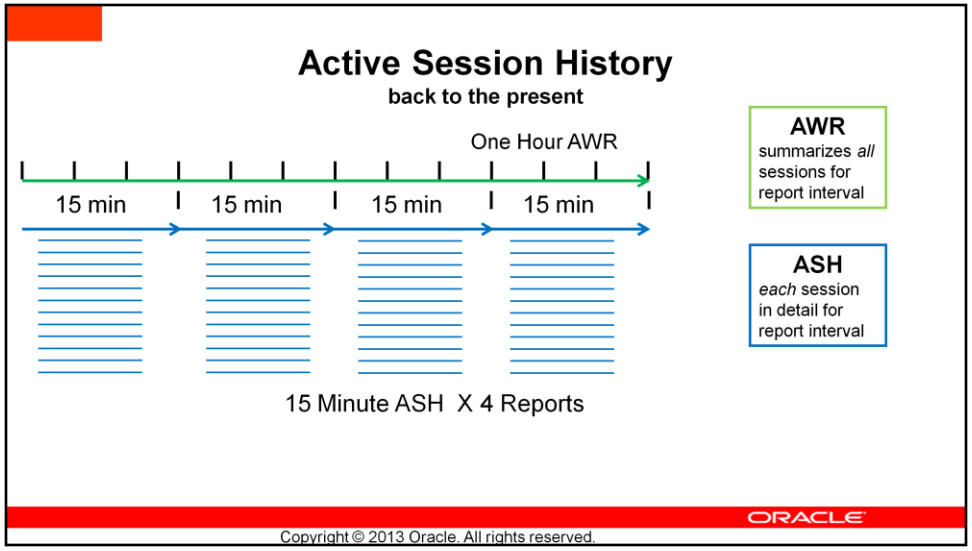


The Library Cache, also known as the “cursor cache”, is a complex hash table used to store semantic and syntactic references to all cursors and executable objects. The table not only includes the cursors themselves, but the symbolic references included, such as table references. The complexity is revealed by the numerous namespaces, which refer to sections of the table by types of objects stored.

Most important here are reloads and invalidations. Reloads occur when shared cursor gets aged out of the shared pool for some reason, most often a lack of shared pool memory.

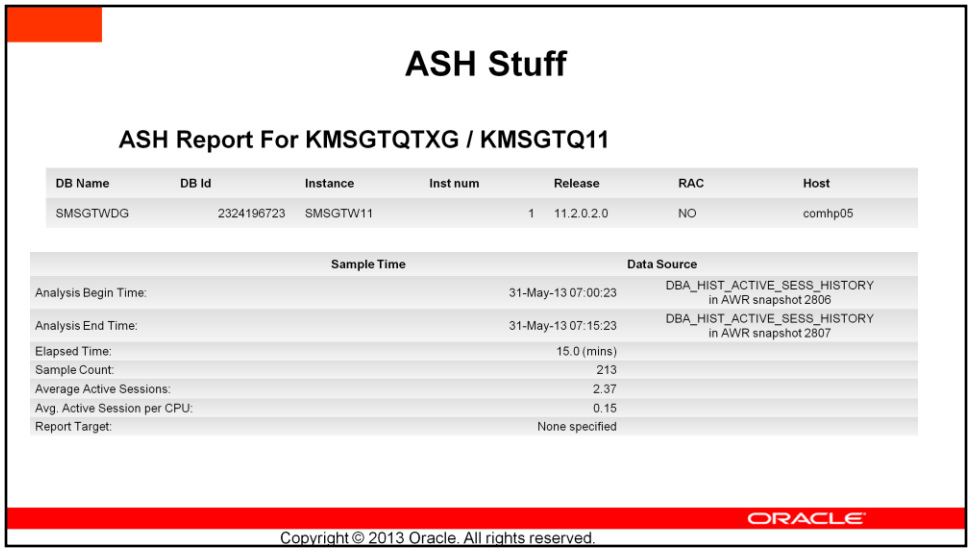
Invalidations happen when DDL is applied to dependent cursors, such as when objects are analyzed for CBO statistics updating. So, avoid such DDL during production hours.

In many cases of data warehouses, users will copy subsets of tables to a private schema (copy table as select…) and operate on that copied data with one-off custom SQL. When the user is finished, and drops the schema, those cursors are invalidated. But, since they are private and temporary, this is no big deal. So, when you see high invalidations, consider the context of usage before you react. This may be no big deal, or it might be a huge deal. The main symptoms for both reloads and invalidations will be some combination of library cache latch/mutex waits, shared pool latch waits, library cache pins, and a spike in hard parses.



Having looked at AWR for an understanding of our performance problem, let's adjust our viewpoint a few degrees and look at the problem from a slightly different perspective. While AWR sums all foreground information for a relatively long interval, Active Session History (ASH) data are captured and stored for each foreground session and much smaller intervals, providing a much more detailed view of instance activity.

In our case, we are fortunate the DBA provided by AWR and ASH reports for the same period. We have four ASH reports of fifteen minutes each, so perhaps we can get a sense of what went on *during* the one our AWR reporting period.



ASH Stuff

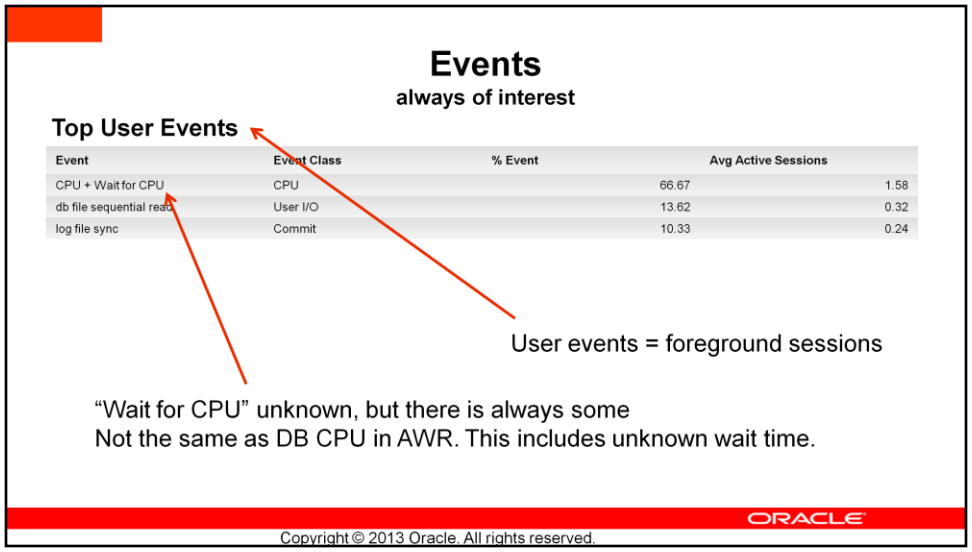
ASH Report For KMSGTQTXG / KMSGTQ11

DB Name	DB Id	Instance	Inst num	Release	RAC	Host
SMSGTWDG	2324196723	SMSGTW11	1	11.2.0.2.0	NO	comhp05

	Sample Time	Data Source
Analysis Begin Time:	31-May-13 07:00:23	DBA_HIST_ACTIVE_SESS_HISTORY in AWR snapshot 2806
Analysis End Time:	31-May-13 07:15:23	DBA_HIST_ACTIVE_SESS_HISTORY in AWR snapshot 2807
Elapsed Time:	15.0 (mins)	
Sample Count:	213	
Average Active Sessions:	2.37	
Avg. Active Session per CPU:	0.15	
Report Target:	None specified	

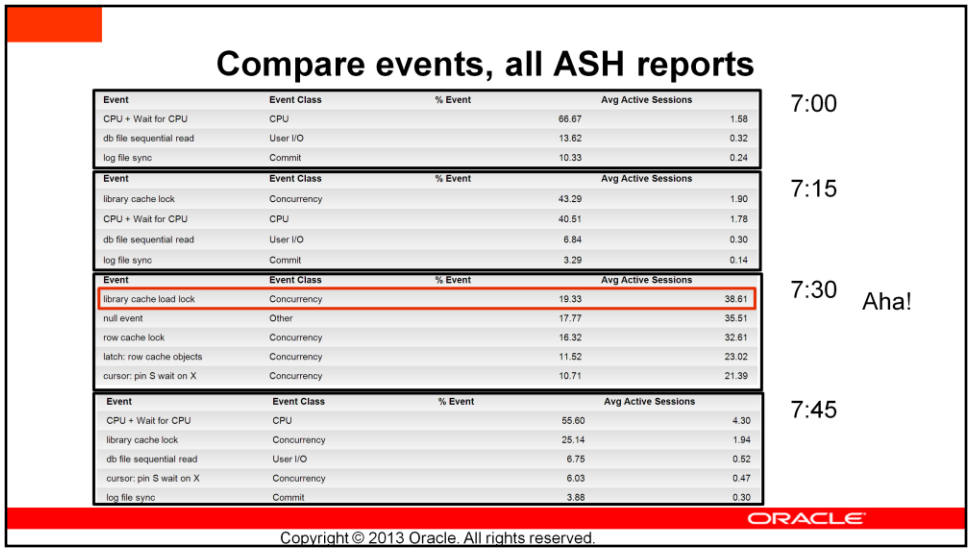
ORACLE

Typical header section, shows the DB Name, instance, db version, etc. Also, the begin and end report times and elapsed time, for comparison.

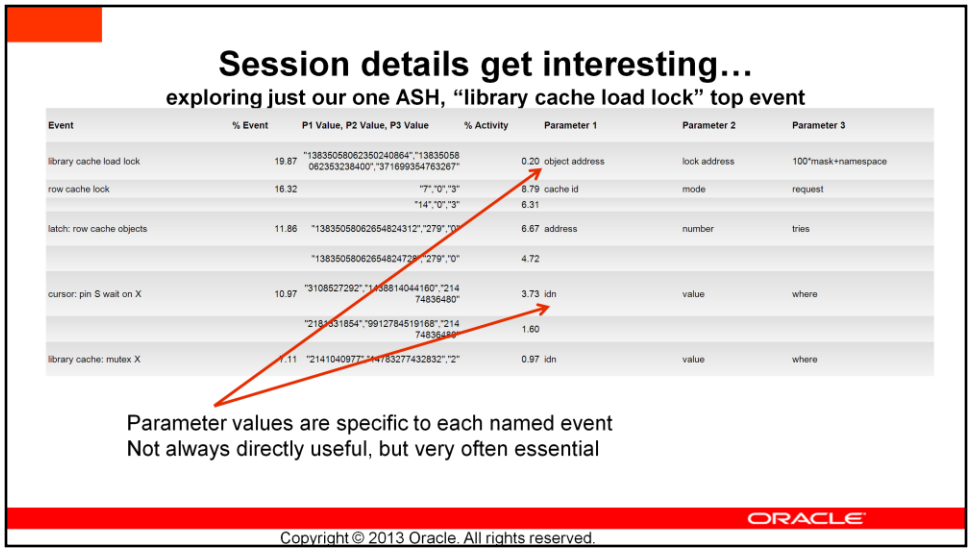


Straight to the meat– Top User Events. Note, CPU + Wait for CPU as a combined value. While AWR shows actual CPU consumption by Oracle processes, as reported by the operating system, ASH cannot distinguish between the consumption of CPU and process time spent on run queue, waiting for CPU. The “why” and “how” of this difference is is a great discussion topic for another day, however.

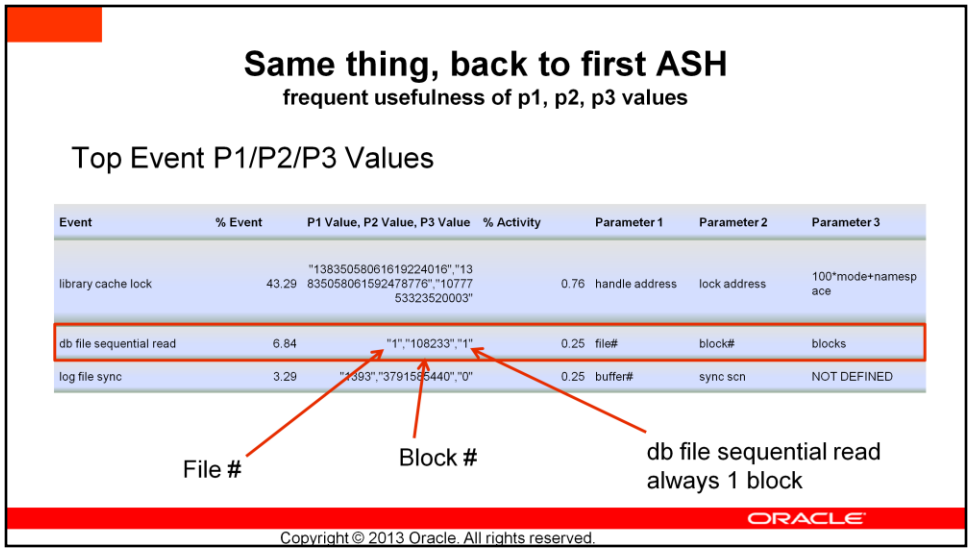
This report page is from the first 15 minute ASH, and its top events does not reflect the entire hour AWR report. Perhaps this is because the “library cache load lock” top waits in the AWR report comes from a narrower period, with time values extreme enough to influence the overall event averages in AWR. Recall our discussion of histograms in AWR showed a relatively small number of events driving the average elapsed time for library cache load locks to a high value. With more granular sampling, our ASH reports should reflect this, as we compare Top User Events for all four ASH reports.



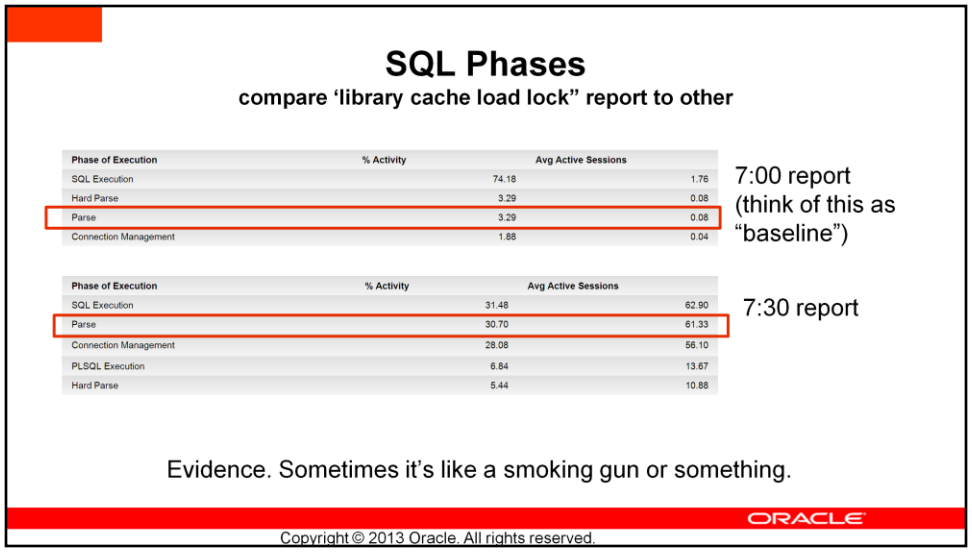
Indeed, library cache load lock only shows up in the third reporting period. Now we can zoom in on that period and try to isolate our causative issue.



An important section of ASH is the session details. Here we see not only our top events, but the parameter values that often make it nail down a problem. The P1, P2 and P3 parameters for wait events are always specific to each event. For example, an I/O related event will have file and object numbers, so you can query the appropriate view on your instance and find the exact block or blocks giving you trouble. In our case, the details of the library cache load lock are only going to be useful in analyzing a systemstate or librarycache dump trace file, which most users will find to technical and tedious. However, for many events, this information will assist directly in customer resolution of problems.

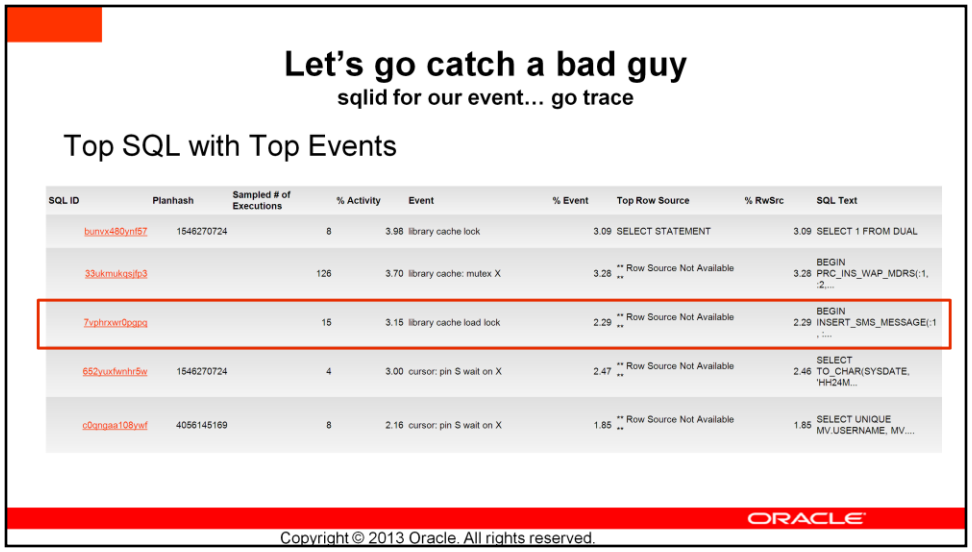


Here is an example of an I/O wait event, with details.



Since our earlier evidence from the AWR top timed events suggested parsing issues rather than SQL execution, we might expect the ASH phases of execution section to reflect this. Indeed, while the first fifteen-minute report shows parse activity as a mere, and more normal, 3.29%, the problem period we determined to the the fifteen-minute period starting at 7:30 shows parse activity has jumped to just over 30% of all activity. This supports our hypothesis the problem is not SQL execution, but some issue related to parsing. And, as we noted earlier, the top SQL section of AWR won't help us in this case.

The great thing about ASH, especially when combined with AWR for the same problem period is to multiple "orthogonal" views you get on the problem, allowing you to get a clearer picture of what is going on inside the instance at any given time.



So, having reviewed much of both AWR and ASH reports, let's look at the quickest way to identify and resolve our issue. Since we know "library cache load lock" is our main event of concern, by reviewing the Top SQL with Top Events section of our fifteen-minute ASH report. This would be the third report, with the beginning time of 7:30.

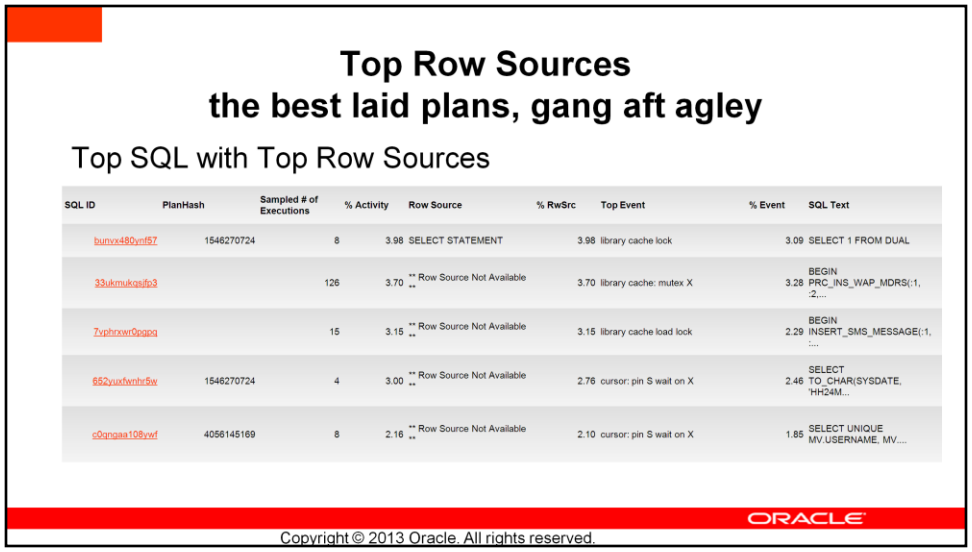
We see the SQL ID with the "library cache load lock" event, so we know which SQL is creating problems for us. Normally, we might see the plan hash, so we could then extract the SQL plan from the instance, but this is a PL/SQL begin/end block as shown in the SQL Text column, so there won't be a plan hash with it. Also, no Row Source, since that is a step in the plan, and there is no plan. For common SQL cursor problems, both those values would be useful in SQL tuning.

The simplest thing to do now is to find any session executing that SQL ID, and trace it. There are many ways to do this, but the simplest is probably by viewing the Enterprise Manager performance page and looking at the top sessions, and tracing from there. Of course, you have to do this when the problem is actually happening, so you still might have to be patient.

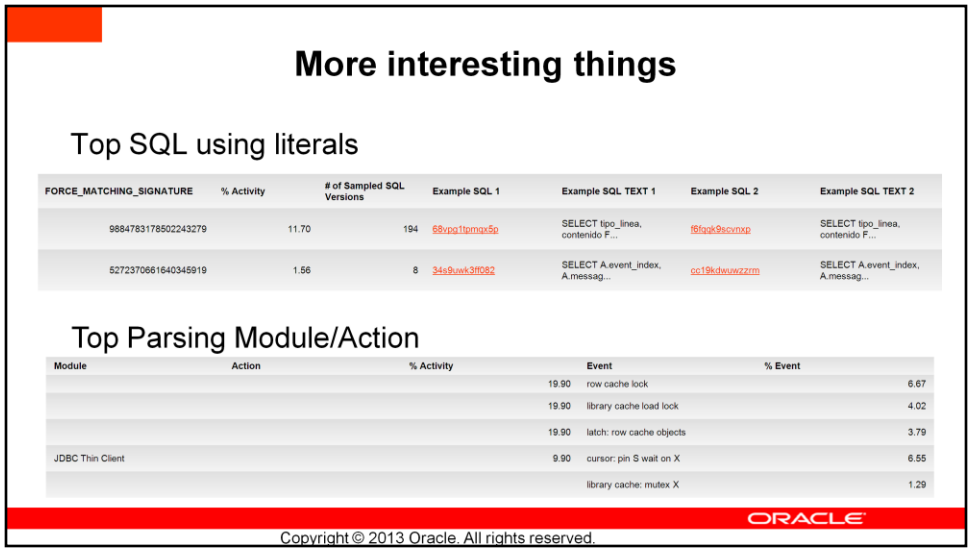
Capture the trace, format it with tkprof, and go through the tuning exercise. Or, download SQLTXplain (note 1454160.1) and use that tool. Notes at the end of this presentation refer you to tools for pursuing SQL tuning.

Our case, as noted earlier, is different. We cannot tune SQL when the problem is with parsing. A SQL Trace still might be useful for diagnostics, as well as an errorstack report (note 37871.1).

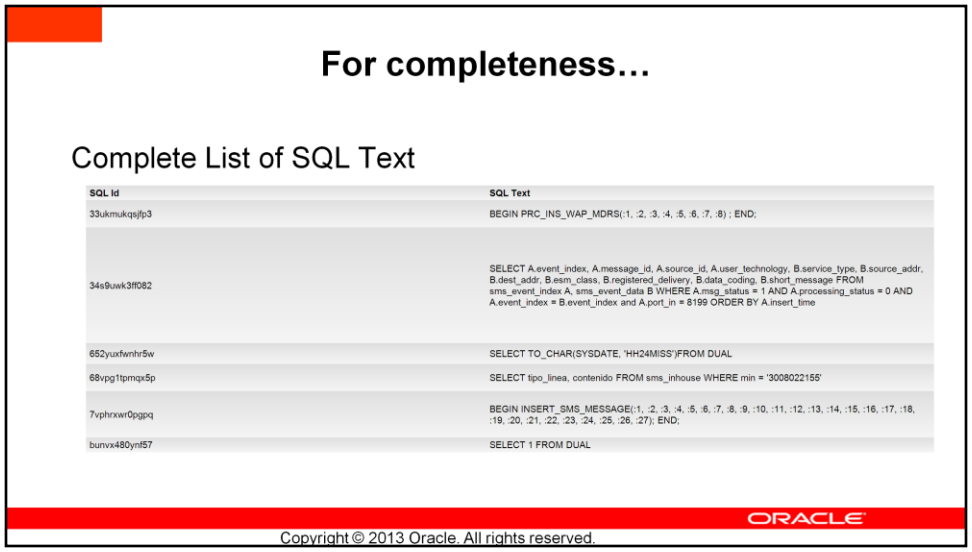
We may also have to get help from the application developer if the cause of a parse failure is rooted in the application code itself. Perhaps an error is detected by the application, but is not handled correctly. Regardless, the information shown in this ASH report will be essential for further diagnostics and resolution.



Another take on Top SQL.



For tracking down non-use of bind variables, Top SQL using literals is useful. And Top Parsing Module / Action can assist where the application has instrumentation using the DBMS Application Info package.



And, all SQL Text, for completeness.

Blockers = Bad

so, this is important, then

Top Blocking Sessions

- Blocking session activity percentages are calculated with respect to waits on enqueues, latches and "buffer busy" only
- '% Activity' represents the load on the database caused by a particular blocking session
- '# Samples Active' shows the number of ASH samples in which the blocking session was found active.
- 'XIDs' shows the number of distinct transaction IDs sampled in ASH when the blocking session was found active.

locking Sid (Inst)	% Activity	Event Caused	% Event	User	Program	# Samples Active	XIDs
3173, 7745(1)	7.75	latch: row cache objects	6.67	SYS	JDBC Thin Client	35/90 [39%]	0
1764, 1455(1)	4.50	library cache load lock	4.31	INH_SMO	JDBC Thin Client	9/90 [10%]	0
335, 4235(1)	4.38	cursor: pin S wait on X	3.73	INH_SMO	JDBC Thin Client	35/90 [39%]	0
2600, 8709(1)	3.30	library cache lock	3.09	INH_SMO	JDBC Thin Client	37/90 [41%]	0
49, 8395(1)	1.81	library cache load lock	1.81	INH_SMO	JDBC Thin Client	43/90 [48%]	0

ORACLE

Latch Waits

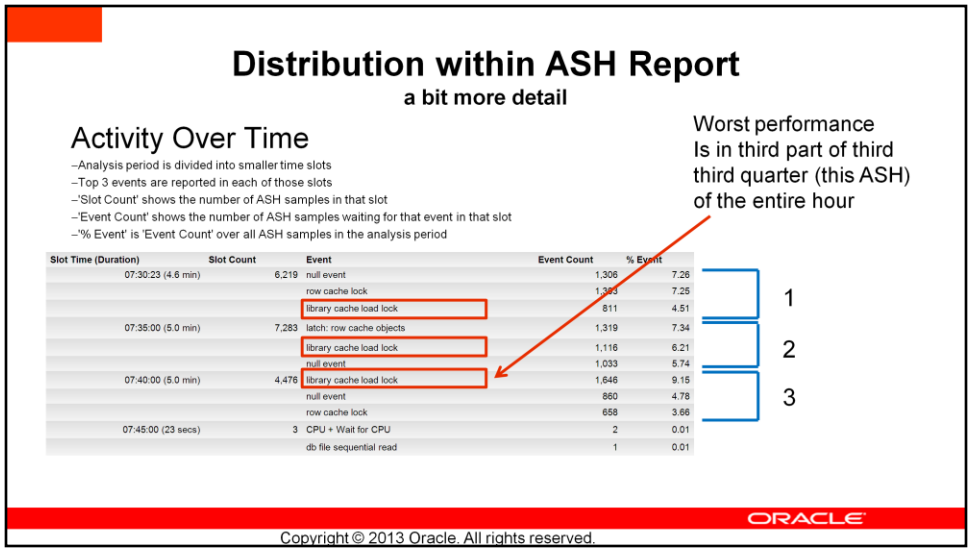
Can be important. Just not today.

Top Latches

Another distribution of waits

Latch	% Latch	Blocking Sid(Inst)	% Activity	Max Sampled Wait secs	# Waits Sampled	# Sampled Wts < 10ms	# Sampled Wts 10ms - 100ms	# Sampled Wts 100ms - 1s	# Sampled Wts > 1s
latch: row cache objects	11.86	3173, 7745(1)	6.67	0.00	0	0	0	0	0
		Held Shared	3.50	0.00	0	0	0	0	0
latch: shared pool	6.02	Held Shared	6.02	0.00	0	0	0	0	0

ORACLE

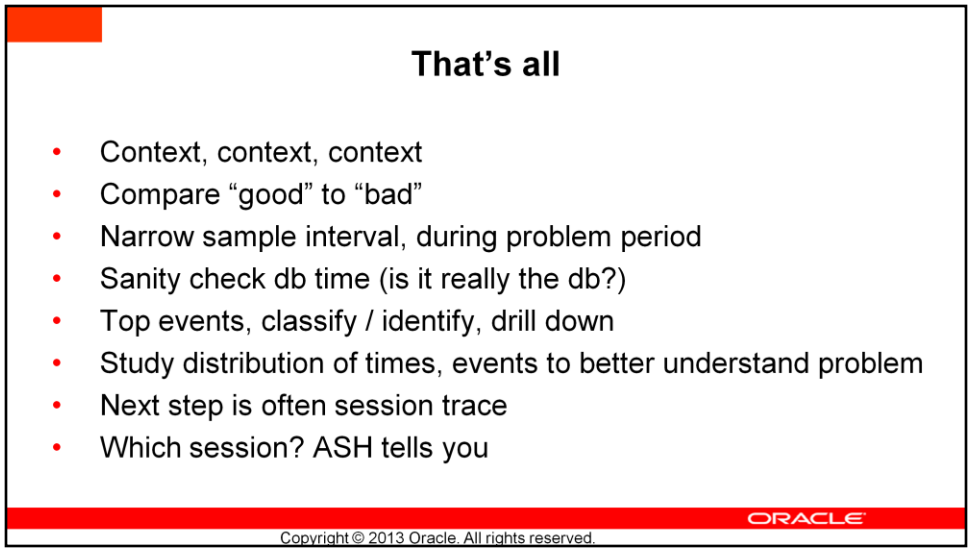


With a distribution analysis of wait events within an ASH report, we can see shifting activity within our fifteen-minute period. This is the third ASH report, with our “library cache load lock” top wait event, broken down into periods of approximately five minutes each. As you can seek, the library cache load lock event was third highest in the beginning of the fifteen-minute report, then moved to second place, and finally top place in the last part of the report. For the final twenty three seconds, the event dropped off again, suggesting it happened as a fairly sudden and intense onset. Also note, for the third period, the event went from 4.51% of all events (by count, not time) to 9.15%, so the spike was fairly high.

Now What?

- For SQL issues, trace session by sqlid or sid/serial from top sessions
- Use SQLT to tune, and / or SQL Plan Management to stabilize
- In our case, find Top Blocking Sessions for “library cache load lock” during problem period. Get session/serial and obtain trace and errorstack, level 12. We want to know what blocking session is doing.
- Could well be an application issue, so application source may tell the story. See how code handles parse failures, or whatever is going on in there.

ORACLE



We are finished. Good reading material includes the Concepts Manual and the Performance Guide.

Please do follow a coherent methodology, at least considering good evidence before taking action, and then verify your actions and outcomes. That is, make sure you have actually made the changes you planned, and that they had the effect you wanted.

For upgrade planning, consider SQL Plan Manager and the Real Application Testing tools. Great resources.

Having analyzed AWR, use ASH, EM or custom queries to find and trace specific sessions. The trace output can be used with tkprof to further analyze and tune SQL or determine other actions for bottleneck relief.

Reference Notes

- How To Collect 10046 Trace (SQL_TRACE) Diagnostics for Performance Issues [ID 376442.1]
- FAQ: How to Use AWR Reports to Diagnose Database Performance Issues [ID 1359094.1]
- Master Note: Database Performance Overview [ID 402983.1]
- FAQ: SQLT (SQLTXPLAIN) Frequently Asked Questions [ID 1454160.1]
- How to Use SQL Plan Management (SPM) - Example Usage [ID 456518.1]
- FAQ: SQL Plan Management (SPM) Frequently Asked Questions [ID 1524658.1]

ORACLE

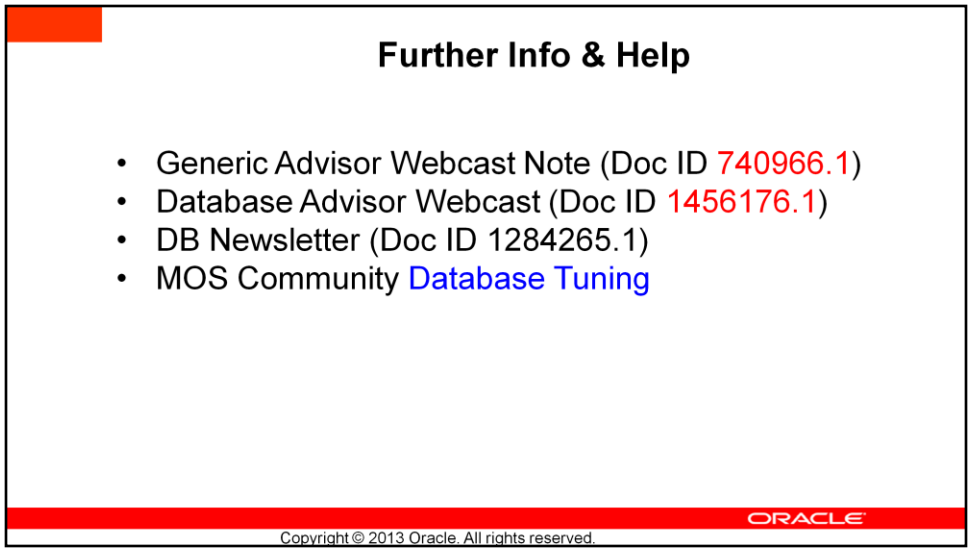
Summary

What we covered today

- Compare “good” to “bad”
- Sanity check db time
- Top events, classify / identify
- Study distribution of event times
- Characterize symptoms
- Most often, trace a session
- Which session? ASH tells you

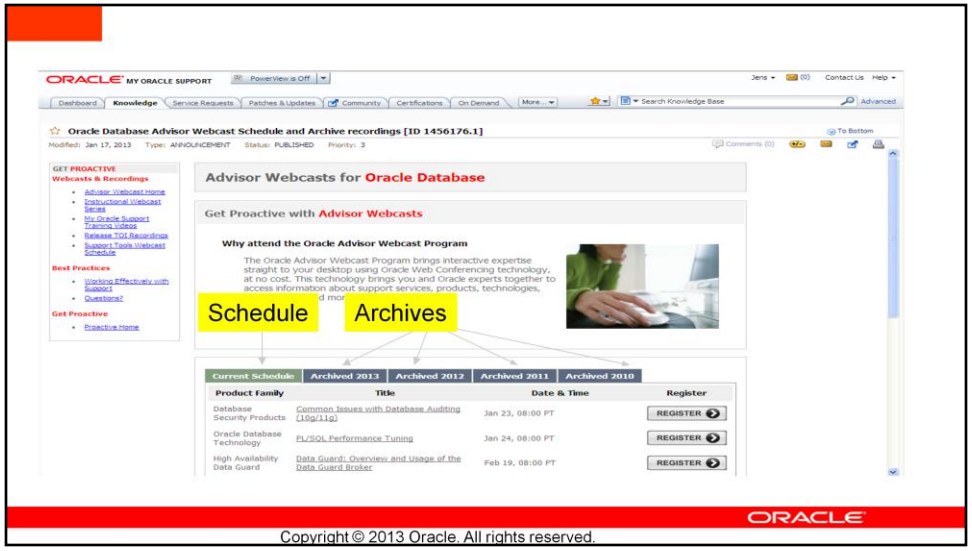
ORACLE

AW template – content slide

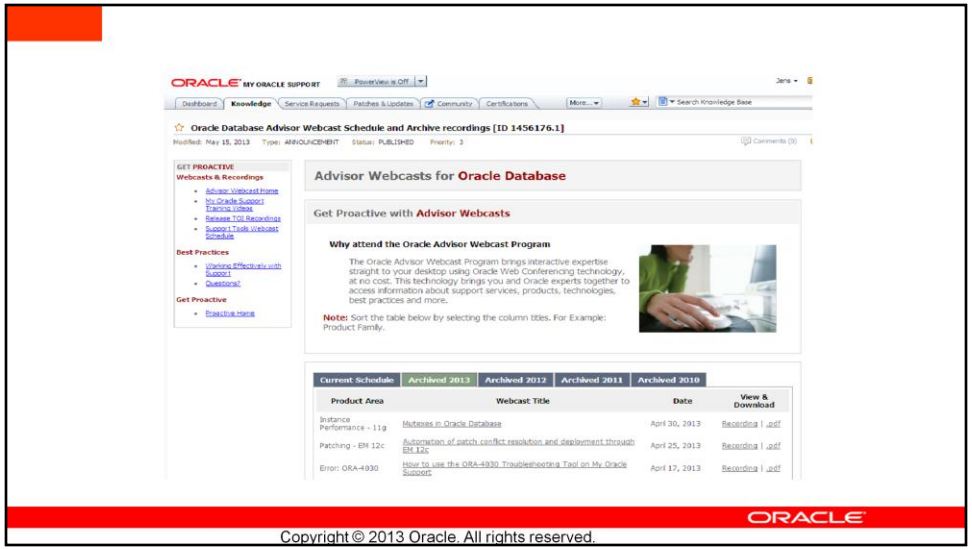


AW template - post event advertising

AW template - post event advertising



AW template - post event advertising



AW template - post event advertising

Learn More

Available References and Resources to Get Proactive

About Oracle Support Best Practices
www.oracle.com/goto/proactivesupport

Get Proactive in My Oracle Support
https://support. oracle.com | Doc ID: 432.1

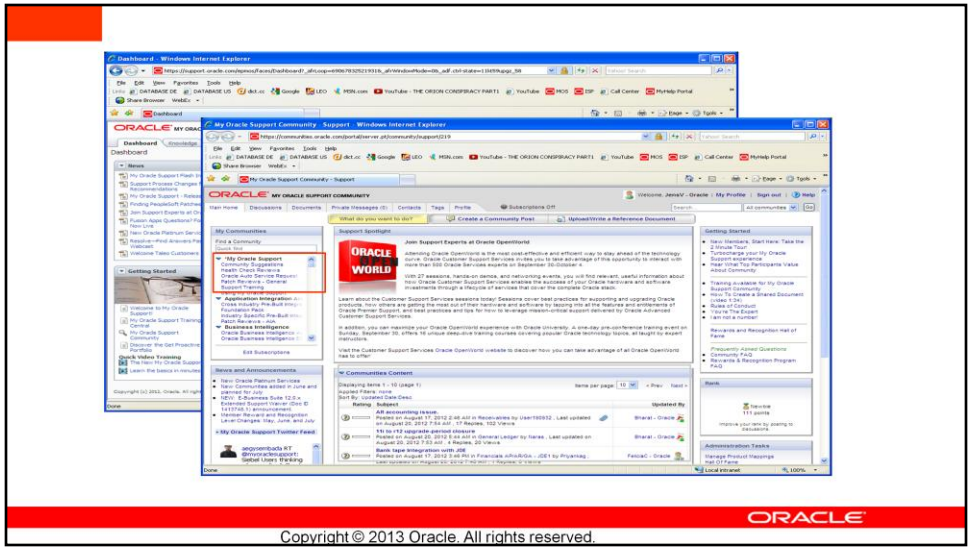
Get Proactive Blog
https://blogs.oracle.com/getproactive/

Ask the Get Proactive Team
get-proactive_ww@oracle.com

ORACLE

AW template - post event advertising

This slide provide links to proactive pages including proactive Blog and email contact. As you can download presentation, this information is available for you at any time



AW template - post event advertising

AW template – Thanks slide

End Webconference

AW template - End