Interpreting AWR reports – straight to the Goal

Franck Pachot
dbi services
Switzerland

Keywords:

AWR, Statspack, Oracle, Tuning, DB Time.

Introduction

A Statspack/AWR report is not something to be read from the beginning to the end. It collects a lot of statistics and we need to catch only the relevant ones. Except for the few first sections, information is grouped by the origin (wait events, time model, statements from V$SQL, instance statistics from V$SYSSTAT, segments statistics, etc). But we don't need to analyse all of them. Our goal is to see where the database is spending most of the time because this is where we can try to decrease the user response time.

Let's learn by example. I've taken an AWR that show the most commonly encountered issues. Note that I use only the sections that are available from Statspack as well, so you can do the same analysis without Diagnostic Pack, and even in Standard Edition. You only have to get Statspack snapshots in level 7 (which is not the default when you instal Statspack) because this is the level that starts to gather Segment Statistics.

Elapsed time and DB Time

The main dimension here in performance tuning is the time. The end-user measures performance as time. And we often measure resource utilization in time.

Report duration

Oracle is heavily instrumented and gathers statistics in real time about the database activity. The numbers are cumulated since the instance startup. In order to see what happens during specific time window, those statistics are stored in snapshots and the AWR/Statspack report is an easy way to view de delta values between two snapshots.

An AWR report shows the activity during a time window. Most statistics do not show the details on the time dimension (exception is the wait event histograms) but only the average value during that elapsed time. As a consequence, if the window is too wide, then averages will hide interesting details. Let's take an example. If your system is 100% busy during one hour and then idle for the 4 remaining hours, then a report covering the whole 5 hours will show a 20% busy system. And you will never be able to address the busy activity properly.

There is another reason to avoid reports that cover a long period. Some information is coming from what was still in the shared pool at the time of the end snapshot. If the duration is too large and you have a lot heterogeneous of activity, then you will probably miss a lot of SQL statements.

Let's check that from the first section:

	Snap Id	Snap Time	Sessions	Curs/Sess
Begin Snap:	330	11-Mar-14 15:18:53	40	5.4
End Snap:	336	11-Mar-14 15:34:32	55	4.7
Elapsed:		**15.64 (mins)**		
DB Time:		**76.48 (mins)**		

Here my report duration is 15 minutes. It's the most important information. And all the remaining will be about analysing what the database was doing during that time.
To be more precise, we will analyse:

- What the users were doing on the database: Users are probably running some SQL or PL/SQL, and then parse it, execute it, or fetch from it. Or running java, etc. We need to get information about the user calls that happened during the elapsed time.
- Which resources the database has consumed to serve those user calls. Of course the Oracle software run in CPU, but can also wait on some system calls (I/O for example).

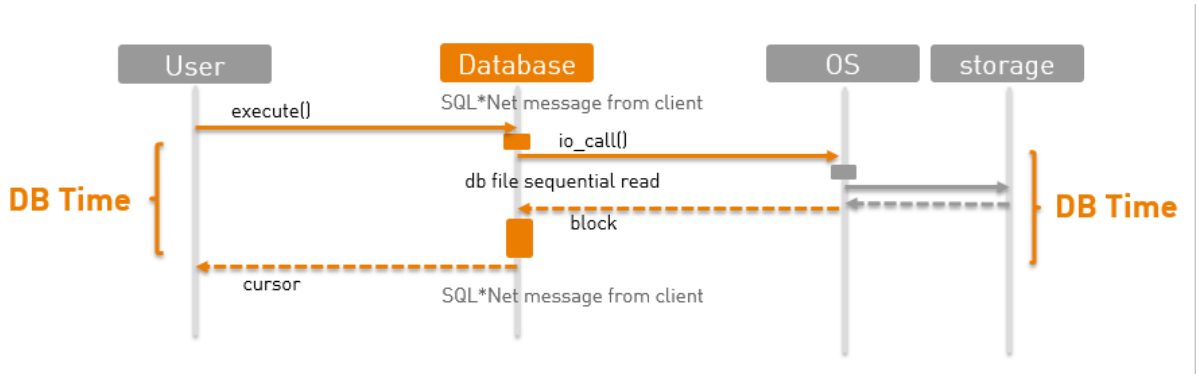
DB time

Most of the time, a user is doing nothing with the database: he is reading the last result that was fetched, or the application server is computing some stuff before calling back to the database, or the user is still connected but is at the coffee machine. All that is idle time for the database. So the DB time – which is the time spend in the database – can be lower than the elapsed time.

On the other hand, the database can be accessed concurrently by several users (sessions) and can process many calls at the same time. For one session, the DB Time is between 0 (totally idle) and elapsed time (totally busy). But for the whole system, the DB Time can be larger than the elapsed time because it is a multiuser software.

From the user point of view, the DB Time is the sum of all user calls to the database, from the end of 'SQL*Net message from client' to the next 'SQL*Net message to client'.

From the system view, the DB Time is the sum of system usage, which is the CPU usage plus any system call to lower systems (storage, network, etc). Application waits (latches, locks, dbms_lock.sleep,…) are implemented as system calls as well.



Our tuning job is all about analysing the database activity. The main unit is the time, which is what we want to reduce. And the AWR report is the balance sheet among resources and their use, that will help to understand where we can improve the user response time.

Database load (aka AAS)

When I read an AWR report I often have to do some simple arithmetic to understand better the reported statistics, and here is the first one – and the most important one:

DB time / Elapsed = 76.48 / 15.64 = 4.9

On average I have 4.9 active sessions. It is 4.9 users running something on the database if I'm in client/server, or it is 4.9 active connections from the connection pool. It's an average. I may have 49 end-users that are spending 10% of their time only on database calls.

That's my database load, telling me whether my database is busy or not. Busy means running or waiting on something that must be completed before being able to run. If DB time is much lower than the sum of the user experienced response time, then the performance issue is probably not at database level. That is our first analysis: when somebody come to us telling that the database is slow we need to check if that 'slow' activity is actually in the database.

It's exactly the same idea as the Unix CPU load which show the average number of processes running or willing to run on CPU. But here it concerns all database resources: not only CPU but also storage, network, or even application waits (locks).

By the way, OS CPU load must be checked before any further wait event analysis, because processes in runqueue can be accounted within the system call rather than with CPU usage. The reason is simply that the statistics are updated by CPU instructions – so they can't be updated while the process is waiting for CPU. What you need to remember is that when all cores are busy, then the wait events are inflated, thus becoming meaningless. We will see later how to check the host CPU activity.

Time model

In order to qualify the DB Time from the user point of view, we go to the time model section:

Time Model Statistics DB/Inst: ORCL/orcl Snaps: 330-336

Statistic Name	Time (s)	% of DB Time
sql execute elapsed time	**4,358.6**	**95.0**
DB CPU	608.4	13.3
parse time elapsed	18.8	.4
hard parse elapsed time	15.7	.3
PL/SQL execution elapsed time	13.6	.3
connection management call elapsed time	8.3	.2
hard parse (sharing criteria) elapsed time	2.5	.1
hard parse (bind mismatch) elapsed time	2.1	.0
PL/SQL compilation elapsed time	1.7	.0
repeated bind elapsed time	0.0	.0
sequence load elapsed time	0.0	.0
DB time	4,588.7	

Here 95% of the time is SQL processing. Note that the total is not expected to be 100%: here 'sql execute elapsed time' includes some DB CPU. You may have also SQL statements calling PL/SQL functions and the PL/SQL execution time will count in both. Most of the timed event items can be overlapping.

Then, it's a good idea to check if most of the statements are identified in the SQL sections:

```
SQL ordered by Elapsed Time               DB/Inst: ORCL/orcl  Snaps: 330-336
…
-> Captured SQL account for   94.6% of Total DB Time (s):           4,589
-> Captured PL/SQL account for   94.5% of Total DB Time (s):           4,589
```

If only a low percentage has been capture, that usually means that the report cover a period where the database had an heterogeneous activity, either the duration is too long or there is too many unshared SQL (not using bind variables). Here I know I'll have detail about 94% of the SQL activity, which is good.

Foreground Events

From the user point of view we have the time model that qualify the user call.
then let's get the system point of view, which will be the start point for our performance analysis:

```
Top 10 Foreground Events by Total Wait Time
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                           Total Wait    Wait   % DB Wait
Event                                Waits Time (sec) Avg(ms)   time Class
------------------------------ ----------- ---------- ------- ------ ----------
db file sequential read             31,762      3,497     110   76.2 User I/O
DB CPU                                             608           13.3
log file sync                        5,676        189      33    4.1 Commit
enq: TX - row lock contention          139        171    1229    3.7 Applicatio
control file sequential read         1,105          9       8     .2 System I/O
```

That shows what we will have to tune on our system if we want to decrease the response time (which the DB Time is responsible for). We will have to address the I/O first as it is the major component here.

A 'Foreground event' which was call 'Timed Event' until 12c is the part of DB Time where the session process is either:

- Running in CPU (which does not include waiting in the host runqueue)
- Waiting for a system call – known from oracle as wait events (which may include the time in runqueue at the end of the call).

Except some bugs where some system calls are not instrumented properly, the time that is not accounted here is the time waiting in the host runqueue in the middle of CPU operations. Then if the total is far less than 100% then it may be a sign of CPU starvation.

And because CPU starvation inflates the wait events, making their duration irrelevant, we need to check the host CPU utilization before continuing with wait events. Fortunately, we have that information in the report as well.

Host CPU

```
Host CPU
~~~~~~~~                   Load Average
 CPUs Cores Sockets      Begin       End      %User   %System      %WIO     %Idle
----- ----- ------- --------- --------- --------- --------- --------- ---------
    8     4       1       1.5       1.1      11.2       3.5      49.2      85.2
```

We need to know if the host suffers CPU starvation or not. When processes are going to runqueue after a system call, the time waiting for CPU is accounted in the wait event. And that means that the wait event analysis is totally biased (inflated). In that case we must address the host CPU issue first.

One thing to consider here is that the CPU utilization reported here is related to the 8 CPUs, but we have physically 4 cores hyper-threaded. Hyper-threading will never double the CPU time available, so it's safer to convert the percentages to be related to the number of cores.

I never consider CPU utilization percentages as reported by the OS without converting them to be related to the number of cores. I've seen people thinking their system was mainly idle because they has 25% utilization. But that was on a 4-way SMT, meaning that all the cores were 100% busy.

In my example, my load average is much lower than the number of cores so I'm quite sure that the processes do not wait in the runqueue when they return from the system call. Good, I can continue on wait event analysis.

Where to start

So let's get back to our start point: the 'Top 10 Foreground Events'

Our goal is to reduce response time. And DB Time is the database part of the response time. So the improvement we can expect is directly related to the percentage of DB Time we address. For that reason we will probably start from the higher percentage we see in the 'Top 10' section.

Addressing a timed event that is just a small percentage of the DB Time is probably a waste of time. There is an exception however for events that are scalability issues (latches, locks, etc). Even if the gain on the response time is minimal, it is good to be sure that we don't have scalability bottlenecks. The reason is that when the number of users will increase, then those wait events will become quickly a major problem. And addressing those wait class is a pro-active tuning.

DB CPU will always be there. The software run on CPU. And CPU resource is scalable as long as the server load don't reach the number of available cores. However, tuning CPU consumption can be interesting because the software license is charged by the number of host cores (or sockets when in Standard Edition). That means that lowering CPU ensures you to be able to follow the load growth without buying more licenses.

'User I/O' wait event will also always be there. It's not an in-memory database. You have to read data from disk. But because disk access has a big latency, we need to lower that. If you have I/O contention, the I/O calls will be queued somewhere and you will see increasing average waits.

'Log file sync' is about writing the redo to disk. It can be written asynchronously but it is mandatory to have you redo written to persistent storage when you commit.

Obviously we will start with those User I/O waits as they are responsible for 76% of the DB Time.

User I/O
You must know the wait events definition, or search in the documentation (Reference Manual). 'db file sequential read' is a single block i/o call. The names can be misleading, for example 'sequential' here is not related to the read pattern (it is not doing large contiguous I/O) and it is rather what is called random read at storage level.

```
Top 10 Foreground Events by Total Wait Time
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                   Total Wait    Wait   % DB Wait
Event                                        Waits Time (sec) Avg(ms)    time Class
------------------------------ ----------- ---------- ------- ------ ----------
db file sequential read             31,762      3,497     110   76.2 User I/O
```

Average time
With I/O, the first thing I check is the average time. Because the expected average I/O service time should be known. For example on a 7200RPM you can't expect less than 8ms. On 15000RPM it can be 4ms. Single block reads are short (default block size is 8k) so the transfer time is minimal.
Of course because of caching you may have shorter I/O calls when they don't have to read from the spinning disk. Or you have SSD with smaller latency.

Well here we see that the average wait time for 'db file sequential read' is 110ms and that's too high.

I/O detail
When digging to the root cause we go to the sections that gives more detail about the wait event.
There is a detail per tablespace and a detail per datafile:

```
File IO Stats                               DB/Inst: ORCL/orcl  Snaps: 330-336
-> ordered by Tablespace, File
Tablespace               Filename
------------------------ ----------------------------------------------------
                  Av       Av     Av                       Av     Buffer  Av Buf
         Reads Reads/s  Rd(ms) Blks/Rd       Writes Writes/s      Waits  Wt(ms)
-------------- ------- ------- ------- ------------ -------- ---------- -------
SOE                      C:\ORA\ORADATA\ORCL\SOE01.DBF
        14,679      16    54.5     1.0        3,434        4         48     1.3
SOE                      D:\ORA\ORADATA\ORCL\SOE02.DBF
        13,689      15   184.6     1.0        2,010        2         31     0.0
SYSAUX                   C:\ORA\ORADATA\ORCL\SYSAUX01.DBF
           506       1    17.5     1.1          959        1          0     0.0
```

In my case it shows that I have a problem that is higher on one disk. At that time I know that I will have to check with the storage team. The latency times are greater than what I can expect. Or maybe that disk is shared by other application that are doing a lot of disk access (such as backups) and I have a contention in the disk queues.
It's not yet in the root cause but at least I know that I need to involve the storage team. This is external to my database: the I/O delivered by the system is not what is expected.

Wait event histograms
In order to get a more detailed picture I check the wait event histograms:

```
Wait Event Histogram                        DB/Inst: ORCL/orcl  Snaps: 330-336

% of Waits
-----------------------------------------------
                           Total
Event                      Waits  <1ms  <2ms  <4ms  <8ms <16ms <32ms  <=1s   >1s
-------------------------- ----- ----- ----- ----- ----- ----- ----- ----- -----
...
db file sequential read    31.8K   2.3    .3    .5   6.2  19.8  22.6  46.8   1.6
enq: TX - row lock content   139    .7                      .7   1.4  45.3  51.8
log file sync               5689  39.9  14.6  15.3   6.1   2.5   3.4  18.2    .0
          ------------------------------------------------------------
```

The goal is to see if all I/O alls are long, or if we have only few outliers that moves the average higher. Here, no doubt, 90% of I/O are between 8ms and 1 second. Wait event histograms are a great way to get beyond the fact that averages hides the details.

Wait count
In the Top Foreground Events above, after having checked the average time I check the number of waits because they are the number of I/O calls. In the case of 'db file sequential read' – single block reads - it's also the number of blocks. And they are are exposed in other statistic: 'physical reads'.
The physical read are detailed by segments (index, tables) in 'Segments by physical reads' and by statements in 'SQL ordered by Reads'

```
Segments by Physical Reads                  DB/Inst: ORCL/orcl  Snaps: 330-336
 -> Total Physical Reads:          31,947
 -> Captured Segments account for   95.5% of Total

            Tablespace                      Subobject  Obj.      Physical
 Owner         Name    Object Name           Name     Type         Reads  %Total
 ---------- ---------- -------------------- ---------- ----- ------------ -------
 SOE        SOE        CUSTOMERS                       TABLE       10,352   32.40
 SOE        SOE        CUSTOMERS_PK                    INDEX        9,620   30.11
 SOE        SOE        ORDERS                          TABLE        2,145    6.71
 SOE        USERS      ORD_WAREHOUSE_IX                INDEX        2,145    6.71

SQL ordered by Reads                        DB/Inst: ORCL/orcl  Snaps: 330-336

    Physical              Reads              Elapsed
       Reads  Executions per Exec   %Total   Time (s)   %CPU    %IO    SQL Id
 ----------- ---------- ---------- ------ ---------- ------ ------ -------------
      15,798     10,640        1.5   49.5    1,992.0     .2   99.9 8dq0v1mjngj7t
 Module: New Order
 SELECT CUSTOMER_ID, CUST_FIRST_NAME, CUST_LAST_NAME, NLS_LANGUAGE, NLS_TERRITORY
 , CREDIT_LIMIT, CUST_EMAIL, ACCOUNT_MGR_ID FROM CUSTOMERS WHERE CUSTOMER_ID = :B
 2 AND ROWNUM < :B1
```

I had 31762 'db file sequential read' and I have the details about a large part of them: 10352 + 9620 are about CUSTOMER table and CUSTOMER_PK index. That is probably customer access by primary key. And the SQL section shows also the main SQL statement responsible for those I/O.

My point here is to know if there is something to tune here or not. On average we have 1.5 physical reads per execution. If I have a large customer database, and not a huge SGA, they obviously don't fit

all in memory. Then it is quite expected that I have to get 1 or 2 blocks (table block plus index leaf block) from disk. I can't expect to reduce that here. This is why on the first sight I don't think I have an application problem here.

So my conclusion at that point, about the User I/O part, is that I have something to tune at system level. If I'm able to get 8ms I/O calls than I can expect a great improvement here.

DB CPU

```
Top 10 Foreground Events by Total Wait Time
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                           Total Wait    Wait   % DB Wait
Event                                Waits Time (sec) Avg(ms)   time Class
------------------------------ ----------- ---------- ------- ------ ----------
db file sequential read             31,762      3,497     110   76.2 User I/O
DB CPU                                             608           13.3
```

In the Top 10 events, the DB CPU is in the second place with 13% of the DB time. It's 608 seconds and, in the same way I had calculated the database load before, I can calculate the CPU load for my database: 608 / (15.64*60)=0.65

This is to be compared to the available number of cores and once again, be careful with the number of CPU reported on hyper-threading or SMT. Here my database is using only 0.65 core on my 4 cores server.

If we want to tune DB CPU we will probably go to the 'SQL ordered by CPU' section. And because we know from the time model that most of the CPU comes from SQL execution, I check the 'SQL ordered by Gets' as well as 'Segments by logical reads'. The reason is that logical reads is often the main cause fir CPU consumption, and I can get the SQL statements and the table/indexes detail from that 'logical reads' statistics in the same way as we did with the 'physical reads' above.

You can get all information about one statement with the awrsqrpt.sql and only one thing is missing: the predicates in the execution plan. So we may have to do an execution plan ourselves in order to have the predicate section.

Log file sync

```
Top 10 Foreground Events by Total Wait Time
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                           Total Wait    Wait   % DB Wait
Event                                Waits Time (sec) Avg(ms)   time Class
------------------------------ ----------- ---------- ------- ------ ----------
db file sequential read             31,762      3,497     110   76.2 User I/O
DB CPU                                             608           13.3
log file sync                        5,676        189      33    4.1 Commit
```

Log file sync is related to commit activity. When you commit and you must be sure that the redo generated by your transaction is written to a persistent storage, you wait on 'log file sync'. In our Top 10 Events we see that we had 5676 commits that were waiting on average 30 seconds for log writer.

There are two things to check then.

First the commit activity, redo size and commit frequency is in the load profile:

```
Load Profile              Per Second    Per Transaction   Per Exec   Per Call
 ~~~~~~~~~~~~         ---------------    --------------- ---------- ----------
       DB Time(s):               4.9               0.3       0.05       0.15
        DB CPU(s):               0.7               0.0       0.01       0.02
        Redo size:          36,961.6          2,128.9
    Logical reads:         121,370.5          6,990.8
    Block changes:             346.9             20.0
       User calls:              32.3              1.9
          Executes:            105.8              6.1
         Rollbacks:              0.0              0.0
     Transactions:              17.4
```

That's about 17 commits per second which is not very high.

Then we can check the log writer I/O activity (through 'log file parallel write' wait event histogram)

```
Background Wait Events                      DB/Inst: ORCL/orcl  Snaps: 330-336

                                                          Avg
                                          %Time Total Wait    wait    Waits   % bg
 Event                             Waits -outs   Time (s)    (ms)     /txn   time
 -------------------------- ------------ ----- ---------- ------- -------- ------
 log file parallel write          14,825     0        195      13      0.9   30.9
```

On that example, my conclusion is that 13 milliseconds is not too much and will probably improve anyway if we address the I/O issue we have seen above. So there is no time to waste going further on that point for the moment. And it's only 13% of the response time anyway. The previous points have a higher priority to address.

That does not mean that we don't have an issue. It just means that we have other issues to address first.

Row lock

```
Top 10 Foreground Events by Total Wait Time
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                          Total Wait    Wait   % DB Wait
Event                               Waits Time (sec) Avg(ms)   time Class
------------------------------ ----------- ---------- ------- ------ ----------
db file sequential read             31,762      3,497     110   76.2 User I/O
DB CPU                                             608           13.3
log file sync                        5,676        189      33    4.1 Commit
enq: TX - row lock contention          139        171    1229    3.7 Applicatio
```

I'll analyse the enqueue contention not because of its %DB time which is small but because it may become a scalability issue when the load increase.

Scalability vs time improvement

Even if the consequence is low, some events can become a scalability issue. That concerns the events that are waiting for a non-scalable resource.

CPU consumption is scalable: you can use more CPU. The exception is when a job is, by design, bound to only one session of course. I/O consumption is scalable, especially if you use asynchronous I/O. you can add disks, or stripe current data into more controllers, more disks.

Locks (as well as latches) are not scalable: all sessions are waiting on one resource only. Having more session means that the waits will increase exponentially.

It's quite easy to find which table is concerned from the segment statistics:

Segments by Row Lock Waits DB/Inst: ORCL/orcl Snaps: 330-336

Owner	Tablespace Name	Object Name	Subobject Name	Obj. Type	Row Lock Waits	% of Capture
SOE	SOE	PRODUCT_INFORMATION		TABLE	138	77.53
** UNAVAIL	** UNAVAIL	** UNAVAILABLE **	AILABLE **	UNDEF	7	3.93

And then we have to match that with the SQL statements that may update (or select for update) that PRODUCT_INFORMATION table. In my example it is obvious from the SQL section:

```
Module: New Order
 SELECT QUANTITY_ON_HAND FROM PRODUCT_INFORMATION P, INVENTORIES I WHERE I.PRODUC
 T_ID = :B2 AND I.PRODUCT_ID = P.PRODUCT_ID AND I.WAREHOUSE_ID = :B1 FOR UPDATE
```

We can understand that we have to lock the inventories table but maybe we can rewrite that query to avoid locking all tables. After checking with the developers we may suggest a FOR UPDATE OF QUANTITY_ON_HAND so that only the inventories table rows are locked.

Anyway, that's for scalability only. We don't expect immediate improvement in response time because it is less than 4% of the total time.

Note that my example is a simple one here because the statement was obvious to find. Locks and Deadlocks may be tricky to understand without having a good picture about how the application behaves. The reason is that only the waits are instrumented, but we may wait on a transaction that acquired a lock a long time before. And oracle cannot trace or log all locks that are acquired or requested.

Quantifying the expectation

What we did above was about findings and recommendation. We have to summarize them at the end of our tuning job. In this document, I focused only on the analysis: see where the issues are, know which team have to be involved to go further. Solutions and recommendations may take more time to get, and we often have several alternatives from workarounds to long-term fixes.
Each alternative will come with a quantified effort to implement. It can be just a setting, or a complete redesign. Besides the estimated effort we need to give an estimated gain so that the management can decide on an implementation.

The great benefit of the DB Time approach is that we can estimate the improvement we can expect from our recommendations. Depending on the context, I can even give precise figures about the expected gain.

Let's see how we can estimate the gain in time. I get back to the 'Top 10 Foreground Events' and change the values with the expected ones:

```
Top 10 Foreground Events by Total Wait Time
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

                                                           Avg
                                                          wait      % DB
Event                                 Waits     Time(s)   (ms)      time Wait Class
------------------------------ ------------ ----------- ------ ------ ----------
db file sequential read             31,762       3,497 317    110 10   76.2 User I/O
DB CPU                                              608  38            13.3
log file sync                        5,676         189          33     4.1 Commit
enq: TX - row lock contention          139         171        1229     3.7 Applicatio
control file sequential read         1,105           9           8      .2 System I/O
```

If we can achieve 10ms I/O instead of 110ms then the first line ('db file sequential read') will be 317 seconds instead of 3497. The response time can be divided by 3.

If in addition to that we are able to decrease the CPU time by 571 seconds (which is the time consumed by our full table scan query) then the response time will be divided by 5.

It's also interesting to see that if we address only the second point the improvement is minimal.

It his example, I can give my recommendations with the expected improvement. Depending on how I know the customer, I can take the risk to give the expectation as 'response time can be divided by 5 if there is no other bottleneck'. Or I can just estimate the gain as low/medium/high. But in both cases, that will help to assign priorities to the implementation of the recommendations.

Core message

Focus only where potential improvement is significant
Methodically analyse from the DB Time to the root cause
Consider only relevant statistics
Estimate the improvement
Document each point with:
- observation
- explanation
- solutions
- expected result

Once implemented compare new report with expectations

References:

- Wait events, statistics, and time model is documented in the Oracle Database Reference book: http://docs.oracle.com/database/121/REFRN
- An example of bad cpu utilization interpretation with hyper-threading: http://www.dbi-services.com/index.php/blog/entry/sockets-cores-virtual-cpu-logical-cpu-hyper-threading-what-is-a-cpu-nowadays-1
- The reference book about Troubleshooting Oracle Performance, by Chris Antognini: http://antognini.ch/top/

Contact address:

Franck Pachot, dbi services, Chemin de Maillefer 36 | CH-1052 Le Mont-sur-Lausanne
Phone: +41(0)79-9632722, Email : franck.pachot@dbi.services.com, Twitter : @FranckPachot