GETTING FAST RESULTS FROM STATSPACK

Bjørn Engsig
Miracle A/S

Abstract

Database administrators and consultants doing Oracle database tuning have been focusing on figures like *buffer cache hit ratio* and *latch free wait time* to assist them in optimizing database performance. Several tools, including the simple script-based Oracle tools like statspack (or utlbstat/utlestat in previous Oracle releases) have been calculating such figures among many others, but with the large amount of data produced you were really not able to answer the simple question: "Where should I start my tuning effort?" Based on a paper by Anjo Kolk et. al., we show you how you can quickly identify exactly those figures from the statspack report, that tell you in which order you should do your various tuning steps. As a result, you are certain not to look at I/O, if you don't have an I/O problem, and you are certain not to attempt tuning the database if the real problem is with the application.

Recap of "YAPP"

In [1], Anjo Kolk et. al. presents a different way to look at database performance diagnostics and tuning. In stead of using ratios, such as the buffer cache hit ratio, or single figures taken out of context, his general idea is based the equation:

response time = service time + wait time

which basically says that the response time perceived by the user consist of service time and wait time. The service time is the time spent by the CPU actively working on your request, and the wait time is the time you spend waiting for some resource to respond or become available. When you e.g. execute a SQL statement that is doing some index lookup, the CPU time involved may be in processing blocks in the buffer cache, scanning an index block for a certain value and getting your requested row out of the data block. To do this, Oracle may have to read the data block from the disk, which incurs a wait time until the disk responds. In more complex cases, you may spend CPU processing PL/SQL and you may wait for a lock or for Oracle to write data to the redo log file when you do a commit.

The general idea behind the YAPP method is to identify in some detail what the components of the service time and the wait time are and simply order these. The component at the top is the one that should be the first one to tune. As a result, you will not make conclusions like "My buffer cache hit ratio is too low, so I better increase the cache", if I/O is not causing any trouble. And you will not say, "I must reduce my 20 second latch wait time", if you are using 20 minutes of CPU processing SQL. A second observation in the method is that tuning something that is taking long time can be done both by reducing the time (such as using faster disks) or reducing the number of times (such as making fewer disk reads). Hence, the steps involved in this method that we will refer to as *time based tuning*, are simply:

1. Identify the service time and the wait time and the components of these
2. Order all time components
3. Start your tuning effort from the top of this list
4. For each entry in the list, either reduce the cost per execution, or the number of executions

The data produced in a Statspack [2] and to some extent in a utlbstat/utlestat [5] report are sufficient to make tuning based on timing rather than tuning based on ratios.

There can, however, be some difficulty in actually identifying the components of the time spent, and when you investigate the details, it will also be realized, that the terms 'service' and 'wait' time in fact are inaccurate. As an example, when you "wait" for a disk I/O request, and the actual request is fulfilled from the operating systems buffer cache, it really is service (i.e. CPU) time.

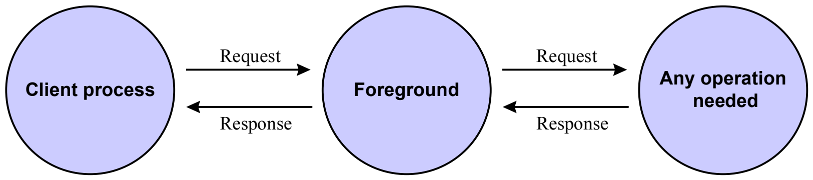
A better formula for calculating the response time is

$$\text{response time} = \sum \text{time component}_i$$

which merely say, that the total response time seen from the users perspective is a sum of different time components. Performance tuning is simply a matter of identifying as many, if not all, of these, and start tuning the largest ones first.

Let us examine the processing in some detail. Seen from the Oracle instance point of view, you typically have three parts involved: The client[1], which is the part that sends requests such as "fetch 10 rows"; the foreground process, which is the Oracle process doing things like processing blocks in the Oracle buffer cache; and the third part, which is anything the foreground needs to involve, such as the file system to read a disk block.[2]

The following picture shows the three parts and also shows how requests and responses are sent between the parts.



In the next section, we will go into details about time measurements in the foreground process.

Time measurement in Oracle

All Oracle processes, both *foreground* processes, i.e. those directly connected to the client or application server side, and *background* processes, such as the database writer or the redolog writer, register CPU time spent and time spent waiting for various events. This information is saved in the Oracle shared memory, SGA, and made available to users via the v$-interface. The information is available both for each individual process, and summed into system level totals. Using this, a person may access views such as v$system_event and v$sysstat to access this information. The Oracle tool *statspack* (and its predecessor *utlbstat/utlestat*) uses queries against these views to capture, calculate and report information. To activate time measurements in Oracle, the database administrator will turn on the *timed_statistics* parameter either during database restart via the init.ora file or using an `alter system` command. All time measurements in Oracle8i and previous releases are in units of 1/100 of a second, also named centi-second or cs for short. Starting with Oracle9i, timings are in 1/1,000,000 of a second, or a microsecond (µs).

This paper is primarily looking at the system level, but the methods shown are applicable at the individual process level as well.

The most important v$-views for the time based analysis are v$sysstat, v$system_event, v$latch, and v$sqlarea. V$sysstat (among much else) shows how CPU time is spent, v$system_event shows information about all events processes have been waiting for, v$sqlarea can be used to find resource intensive SQL statements, and v$latch provides information about latches, broken down by the different areas of the system global area protected by each latch.

[1] In three- or N-tier environment, the client will be the application server, the transaction monitor or similar.
[2] When comparing the terms used here with [1], the term "foreground" is used equivalently, whereas Anjo Kolk makes some distinction between background processes (such as the redolog writer process) and wait time (such as waiting for a read from disk). As we shall see, this distinction is subtle and actually unnecessary.

V$system_event

This view has a number of columns; for this discussion the two important ones are:

EVENT	Name of an event, that Oracle has been waiting for such as 'db file sequential read' for reading from a database file or 'buffer busy wait' when waiting for a database buffer to become available.
TIME_WAITED	Total time waited for this event since the start of the database measured in cs (independent on Oracle version)

Each row in the v$system_event view refer to an individual wait event, and all rows are potentially interesting. However, there are events that are far more common than others, and some events are *idle events*, meaning, that they do not actually identify active waiting within a user call.

V$sysstat

This view has a number of columns; for this discussion the two important ones are:

NAME	Name of an internal Oracle statistics such as the 'CPU used by this session'
VALUE	Value of the statistics

Each row refer to a specific statistical value, but only few of these are important here.

V$latch

Latches are internal Oracle structures, similar to locks, that control access to data in the system global area. Latches are held while data is being modified to prevent concurrent modification from several processes at the same time. Important columns in the v$latch view are:

NAME	Resource protected by the latch, e.g. shared_pool
SLEEPS	Number of times the latch request could not be fulfilled and the process was sleeping until it could attempt another request.

All rows are potentially relevant, but few are particularly important and common.

V$sqlarea

All SQL statements executed by Oracle are stored in the shared pool part of the system global area. As long as space permits, SQL statements that have recently been executed are also saved, but eventually, new SQL statements will replace old ones. The v$sqlarea view lists all SQL statements found in the shared pool, both currently executing, and those that have not yet been aged out. Important columns of this view are:

SQL_TEXT	Actual text of the SQL statement (shortened to a maximum of 1000 characters)
BUFFER_GETS	Number of buffer gets, i.e. logical block reads, by this SQL statement. This is the most important factor for CPU usage of SQL statements.
DISK_READS	Number of physical read calls by this SQL statement.
CPU_TIME	CPU time spent processing this SQL statement. This column is available since Oracle9i

Using Statspack

Generally, the v$-views mentioned in the previous section display cumulative values since the start of the Oracle instance, but this absolute figure is rarely of interest; the suggested approach is to take snapshots of these views at regular intervals and calculate the difference between to successive snapshots. The Oracle tool *Statspack* is able to do exactly this; the typical setup is to take snapshots at regular intervals using the Oracle job queue mechanism. Once data are collected, the database administrator can run the reporting script that is shipped with statspack, and it will make all calculations based on the difference between two selected snapshots. Statspack is fully documented in [3].

The statspack report includes several sections; among these are sections containing information from the four views mentioned above.

Identifying time components

The whole purpose of time based tuning is to find areas that show a high potential benefit tuning. In the equation from [1]

response time = service time + wait time

which we have modified to

$$\text{response time} = \sum \text{time component}_i$$

you need to identify the time components which are either the service time spent inside the foreground, or the wait time being time spent waiting for an event. The information necessary can be found from v$sysstat data, found under the heading 'Instance Activity Stats for DB', and from v$system_event, found under the heading 'Wait Events for DB'. You need to identify the following three v$sysstat values:

CPU used by this session	Total CPU time spent.
Recursive cpu usage	Time spent doing recursive work in the foreground. This includes data dictionary lookup and any PL/SQL work, including time spent by SQL inside PL/SQL.
parse time cpu	CPU time spent parsing SQL statements.

The total time spent is the 'CPU used by this session' statistics, and two of the components are 'recursive cpu usage' and 'parse time cpu' respectively. The CPU component, 'recursive cpu usage' is made up from three different sources: Data dictionary lookup; processing PL/SQL code; and executing SQL statements from PL/SQL code. It is currently not possible to distinguish these three. Therefore, in most cases, you would only be able to identify two components of CPU: CPU spent parsing, and any other CPU spent. This will subsequently be referred to as 'other cpu', and it can be calculated using

other cpu = CPU used by this session - (parse time cpu)

If it is known, that there is little to no SQL processing done within PL/SQL, you should also subtract 'recursive cpu usage' from 'CPU used by this session' to get the 'other cpu' component. In such a scenario, 'resursive cpu usage' is a time component by itself, and the formula for calculating other cpu is

other cpu = CPU used by this session - (recursive cpu usage + parse time cpu)

Next, you need to find the components of the wait time. The simplest approach is to find the top five wait events under the heading 'Top 5 Wait Events'; alternatively, you can find the largest of the wait events under the heading 'Wait events for DB'

The following is a step-by-step set of operations that will quickly lead you to your most important place of tuning.

1. Identify the time components 'parse time cpu'.
2. Identify the value 'CPU used by this session' and subtract 'parse time cpu' from this. This gives the second time component 'other cpu'.
3. Identify the most time-consuming wait events[3].
4. Sort the values from step 1 through 3 by descending times and start your tuning effort from the top.
5. If the most time consuming operation is any other wait event than 'latch free', look at the wait event tuning recommendations below.

[3] Some events are 'idle events', which should not be included, and they are in fact normally omitted in the Statspack report. See the table included in this paper or [1] for a complete list.

6. If the most time consuming operation is the 'latch free' event, look at the latch tuning recommendations below.
7. If the most time consuming operation is any CPU component, follow the CPU tuning recommendations below.

An important aspect of this approach is that you should *not* attempt any tuning of a component that is not on the top of the list of times spent identified in step 4. Hence, even if you seem to have a low buffer cache hit ratio, you should only tune I/O or modify the buffer cache size, if step 4 above identifies I/O as being a problem. The fact is, that most such ratios have little or no importance with respect to performance diagnostics. More details about this topic is found in [6] and [7].

The following sections detail the tuning possibilities, that should be followed based on the outcome of the above analysis. For each time component, description and proposed actions are described.

Tuning possibilities for CPU

recursive cpu usage	This component can be high if large amounts of PL/SQL are being processed. It is outside the scope of this document to go into detail with this, but you will need to identify your complete set of PL/SQL, including stored procedures, finding the ones with the highest CPU load and optimize these. If most work done in PL/SQL is procedural processing (rather than executing SQL), a high recursive cpu usage *can* actually indicate a potential tuning effort.
parse time cpu	Parsing SQL statements is a heavy operation, that should be avoided by reusing SQL statements as much as possible. In precompiler programs, unnecessary parting of implicit SQL statements can be avoided by increasing the cursor cache (MAXOPENCURSORS parameter) and by reusing cursors. In programs using Oracle Call Interface, you need to write the code, so that it re-executes (in stead of re-parse) cursors with frequently executed SQL statements. The v$sql view contains PARSE_CALLS and EXECUTIONS columns, that can be used to identify SQL, that is parsed often or is only executed once per parse.
other cpu	The source of other cpu is primarily handling of buffers in the buffer cache. It can generally be assumed, that the CPU time spent by a SQL statement is approximately proportional to the number of buffer gets for that SQL statements, hence, you should identify and sort SQL statements by buffer gets in v$sql. In your statspack report, look at the part 'SQL ordered by Gets for DB'. Start tuning SQL statements from the top of this list. When using utlbstat/utlstat (in release 8.1.5 and earlier), the report produced does not include statistics on SQL statements; hence you actually need to query the v$sqlarea view at intervals and observe which SQL statements are growing most rapidly in buffer gets. In Oracle9i, the v$sql view contain a column, CPU_TIME, which directly shows the cpu time associated with executing the SQL statement.

Tuning possibilities for wait events

db file scattered read	The scattered read event is used when Oracle needs to read multiple blocks at a time during a full table scan. The init.ora parameter db_file_multiblock_read_count specifies the maximum numbers of blocks read in that way. Typically, this parameter should have values of 4-16 independent of the size of the database but with higher values needed with smaller Oracle block sizes. If you have a high wait time for this event, you either need to reduce the cost of I/O, e.g. by getting faster disks or by distributing your I/O load better, or you need to reduce the amount of full table scans by tuning SQL statements. See below for more detail about I/O tuning.
db file sequential read	The sequential read event identifies Oracle reading blocks sequentially, i.e. one after

	each other. This happens during normal (index based) operations. If you have a high wait time for this event, you either need to reduce the cost of I/O, e.g. by getting faster disks or by distributing your I/O load better, or you need to reduce the amount of I/O by increasing the buffer cache or by tuning SQL statements. See below for more detail about I/O tuning.
buffer busy waits	A buffer busy wait happens when multiple processes concurrently want to modify the same block in the buffer cache. This typically happens during massive parallel inserts if your tables do not have free lists and it can happen if you have too few rollback segments. The view v$waitstat, which is also part of the Statspack report can assist you in identifying the actual cause.
latch free	Please see the subsequent section on latch tuning possibilities.
enqueue	Enqueues are generally locks used by the application, e.g. when a select for update is executed. If you see high wait time for the enqueue event, you need to look at the application and in particular look for code, that holds locks for long periods of time. It is not possible to identify the time spent waiting for each individual lock, although the v$lock view tells the number of waits for each type of lock.
log file sync	Whenever a session does a commit, it needs to post the redolog writer process to flush the log buffer. If this event has a high wait time, or you need to reduce the number of commits by committing larger transactions, you need to optimize the I/O subsystem for better log file performance. The associated event, 'log buffer parallel write' is used by the redo log writer process, and it will indicate if your actual problem is with the log file I/O. Large wait times for this event can also be caused by having too few CPU resources available for the redolog writer process.
free buffer wait	When a session needs a free buffer and cannot find one, it will post the database writer process asking it to flush dirty blocks. You need to investigate the 'db file parallel write' event in the database writer process to identify if you need to modify your I/O system; alternatively, the database writer process is not sufficiently active. If the free buffer wait event has high waiting time, which is not caused by poor I/O write capacity, you can tune your instance by increasing the buffer cache.
rdbms ipc message pmon timer smon timer SQL*Net message from client	These events are idle events and are expected to have high values. Statspack actually excludes these events in most of the report, but they are included in the utlbstat/utlestat report. For a full list of idle events, please see [1].

Tuning possibilities for latches

shared pool	The shared pool latch is heavily used during parsing, in particular during hard parse. If your application is written so that it generally uses literals in stead of bind variables, you will have high contention on this latch. In release 8.1.6 and later, you can set the cursor_sharing parameter in init.ora to the value 'force' to reduce the hard parsing and reduce some of the contention for the shared pool latch. Applications that are coded to only parse once per cursor and execute multiple times will almost completely avoid contention for the shared pool latch.[4]
library cache	The library cache latch is heavily used during both hard and soft parsing. If you have high contention for this latch, your application should be modified to avoid parsing if

[4] A complete discussion of the usage of this latch and the associated startup parameters is found in reference [2].

	at all possible. Setting the cursor_sharing parameter in init.ora to the value 'force' provides some reduction in the library cache latch needs for hard parses, and setting the session_cached_cursors sufficiently high provides some reduction in the library cache latch needs for repeated soft parsing within a single session. There is minor contention for this latch involved in executing SQL statements, which can be reduced further by setting cursor_space_for_time=true, if the application is properly written to parse statements once and execute multiple times.[4]
row cache	The row cache latch protects the data dictionary information, such as information about tables and columns. During hard parsing, this latch is used extensively. In release 8.1.6 and above, the cursor_sharing parameter can be used to completely avoid the row cache latch lookup during parsing. [4]
cache buffer chain	The cache buffer chain latch protects the hash chain of cache buffers, and is used for each access to cache buffers. Contention for this latch can often only be reduced by reducing the amount of access to cache buffers. Using the X$BH fixed table can identify if some hash chains have many buffers associated with them. Often, a single hot block, such as an index root block, can cause contention for this latch. In Oracle9i, this is a shared latch, which minimizes contention for blocks being read only.
cache buffer lru chain	The buffer cache has a set of chains of LRU block, each protected by one of these latches. Contention for this latch can often be reduced by increasing the db_block_lru_latches parameter or by reducing the amount of access to cache buffers.

Tuning possibilities for I/O

If it turns out that I/O is where you spend most of the time, either directly via the db file scattered or sequential read events or indirectly via the file write events in the database writer or redolog writer, you need to get information about your I/O rates. The statspack report includes a section titled 'Tablespace IO Summary for DB', which lists all tablespaces and their I/O rates, and the section titled 'File IO Statistics for DB' lists I/O rates for each individual data file. Initially, you should do some verification of the expected rates, and secondly you should look at the distribution of I/O rates. If your I/O rates are reasonable (e.g. 2-10 ms per I/O for cached file systems or disk systems with cache or 5-20 ms per I/O for raw disk devices), and if I/O on all data files have similar rates, you can conclude, that your I/O subsystem is providing adequate performance. In this case, it is unlikely that you can reduce the cost of I/O, hence, you need to reduce the amount of I/O either by having a larger buffer cache or by modifying SQL statements. If, however, I/O rates are much larger than expected or are poorly distributed, you will be able to reduce the cost of I/O by changing the I/O subsystem. This may involve acquiring more disk drives, modifying architecture (e.g. avoiding RAID 5) or simpler redistribution of I/O.

Caveats and boundary effects

Although the Oracle statistics and wait event data give very good data for you to identify bottlenecks in your Oracle system, there are areas, where you may be mislead by the data provided:

- In Oracle8i and previous versions time granularity is one centisecond, but on very fast systems, events may overlap timing intervals. Therefore, some events that did happen are never registered and some short running events are registered with a much longer time than the actual time. It is generally assumed, that these effects average out, but this cannot be relied upon. This problem does not exist in Oracle9i, where the time granularity is one microsecond.
- CPU time spent in the Oracle foreground processes is only coarsely instrumented. The CPU used by this session statistics is generally by far the largest, and the assumption that the CPU usage of SQL statements is proportional to the number of buffer gets is only an approximation. In particular cases where large amounts of PL/SQL is involved, or where complex expressions, join operations, or predicates are being

evaluated may be inaccurate. Generally, OLTP type applications are the ones, where the approximation is most valid. For decision support type applications it is more likely, that the approximation is not correct.

- The v$sysstat view contain summed information from both foreground and background processes. However, the CPU time components that are of interest are only those from the foreground processes. As some of the background processes (in particular the database and redolog writers) use non-negligible amounts of CPU, this can cause perceived overhead in CPU usage.
- Some time is not at all accounted for. For example, time spent in SQL*Net or time spent waiting for CPU or doing context switching on a heavily loaded server with many processes or threads is not covered.
- The method is working from the perspective of the Oracle server, or more precisely from the Oracle foreground processes. Effects of network latencies or bottlenecks on the application server are not included. This can be elaborated by inspecting the formula for response time again:

$$\text{response time} = \sum \text{time component}_i$$

If time spent in Oracle is only a small part of the total time, tuning Oracle is not likely to improve response time much!

Using utlbstat/utlestat

Statspack is available in Oracle since release 8.1.6, and should be your preferred tool to get information from the v$ views. If you are running an earlier version of Oracle8, there is a suitable version of statspack available for download from http://technet.oracle.com. In previous releases, the statspack predecessor utlbstat/utlestat is available. The major differences between the two tools are:

- Utlbstat/utlestat is run directly by the database administrator rather than being configured and run via the Oracle job queue mechanism. Utlbstat/utlestat only provides data from one time interval at a time.
- Utlbstat/utlestat does not display any detailed information about SQL statements, and you are therefore not able to find the most active SQL statements in the utlbstat/utlestat report. If your conclusion is that 'other cpu' is the largest time component, you need to query the v$sql view directly to identify resource intensive SQL.
- There is no top-five list of wait events with utlbstat/utlestat. In stead, you need to find the largest wait event by inspecting the table of non-background wait events listed.
- The file produced by utlbstat/utlestat is always 'report.txt'. Hence, in order to save results from different executions of utlbstat/utlestat, you need to rename the file after each execution.

Using the method at the session level

It is outside the scope of this document to discuss using time based performance analysis at the level of individual sessions. However, the method is fully applicable at that level as well:

- You will find session level views similar to the system level ones listed above: V$sesstat contains statistics and v$session_event contains wait event information for all currently running sessions. While a session is running, you can make queries against these views and make conclusions at the session level similar to the system level conclusions discussed in this paper.
- The statistics 'CPU used by this session' is not updated while a PL/SQL block or stored procedure is executing. Therefore, it can be difficult to correctly identify CPU time components of sessions that execute long running PL/SQL.
- v$session_wait contain a snapshot of events, that sessions are waiting for at the time of the query. In order to identify exactly which disk block is being read via the 'db file sequential read' event, you can continuously execute a SQL statement like `select * from v$session_wait where event='db file sequential read'` and occasionally, it will display full information about the file number and block number that is being read. Subsequently, you can use the data dictionary view (e.g. DBA_EXTENTS) to identify the object that is physically found at that place.

The *hotsos profiler* [9] uses this approach extensively.

Acknowledgments

This paper is a result of input from several sources. Anjo Kolk, who has been more than 15 years with Oracle, wrote his initial YAPP paper around the Oracle version 7.0 time frame, and the idea of using the "wait interface" has since evolved among several groups worldwide. Cary Millsap and Mogens Nørgaard, who both have been in the Oracle community for more than a decade have been promoting this idea of basing performance diagnostics on understanding of where you actually spend your waiting time rather than on more or less meaningless ratios. Many late night talks to these people have inspired this paper, which I hope will be able to really explain how simple the idea is, and how simple you can use the available data, e.g. from statspack.

The pun on the word *fast* in the title of this paper is clearly intentional. You should get results fast by quickly looking at the important numbers and avoiding those that have little impact on *your* database, and after doing the necessary tuning, you should get fast results from your database system.

References

[1] Anjo Kolk, Shari Yamaguchi, Jim Viscusi, 1999: Yet Another Performance Profiling method, YAPP. Available from http://technet.oracle.com/deploy/performance

[2] Graham Wood and Connie Dialeris Green, 2000: Diagnosing Performance with Statspack, Part I. Available from http://technet.oracle.com/deploy/performance

[3] Graham Wood and Connie Dialeris Green, 2000: Diagnosing Performance with Statspack, Part II. Available from http://technet.oracle.com/deploy/performance

[4] Oracle Corporation, 1999: Oracle8i Reference, Chapters on dynamic performance views, wait events and statistics.

[5] Oracle Corporation, 1999: Utlbstat.sql and utlestat.sql, available as text files with Oracle versions 8.1.5 and previously. The two files, including instructions in the first one of these, are found in the rdbms/admin directory under ORACLE_HOME.

[6] Mogens Nørgaard, 2000: Introducing Oracle's Wait Interface. Available from http://www.hotsos.com

[7] Cary Millsap, 2001: Why a 99%+ Database Buffer Cache Hit Ratio is NOT Ok. Available from http://www.hotsos.com

[8] Bjørn Engsig, 2001: Efficient use of cursor_sharing and related startup parameters. Available from http://technet.oracle.com/deploy/performance

[9] Hotsos profiler, available from http://www.hotsos.com/products/profiler.html.

Revision history

15-sep-2001	Submitted to EOUG forum in Rome
2-jan-2002	Published at Miracle A/S homepage, www.miracleas.dk