

# DIAGNOSING PERFORMANCE BOTTLENECKS USING STATSPACK AND THE ORACLE PERFORMANCE METHOD

*Connie Dialeris Green, Oracle*

## **INTRODUCING THE PERFORMANCE METHOD AND STATSPACK**

This paper provides an overview of the Oracle Performance Method<sup>1</sup>, with a focus on Oracle instance performance diagnosis using Statspack within the method's framework. An example of interpreting a Statspack report is also included to illustrate the method.

The performance method is a simple approach for developing and maintaining a well performing system. It is comprised of two parts. The first part is proactive, and encompasses the analysis, design, development and system test phases of application deployment. The second, the reactive component, is a simple technique which pinpoints performance problems in your Oracle system by identifying the primary bottleneck. This paper will focus on the reactive component.

Statspack is a performance data gathering tool which first shipped with Oracle<sup>8i</sup> release 8.1.6. Statspack gathers data from the memory-resident v\$ views, and stores that data in Oracle tables for later analysis. Although similar to its predecessor (BSTAT/ESTAT), Statspack simplifies performance diagnosis by presenting the performance data in a manner which is effective for pinpointing bottlenecks.

## **THE PERFORMANCE METHOD - PROACTIVE**

The proactive component of the performance method provides a focus on scalability and throughput issues. It documents Oracle-specific considerations for sound development and implementation strategies for OLTP systems, including:

- simple schema and application design
- instrumenting application performance measures into the application
- avoiding common oversights in application architecture
- caching frequently accessed data
- workload testing, modeling and implementation
- SQL plan capture
- proactive performance statistics capture (i.e. performance baselines), with timing data (timed\_statistics = true)

The main premise is to design your application and make implementation decisions, with scalability in mind from the start. The proactive phase of the performance method emphasizes the importance of gathering baseline statistics for future performance and capacity management.

### *THE IMPORTANCE OF BASELINES AND STATISTICS*

One of the biggest challenges for performance engineers is determining what changed in the system to cause a satisfactory application to start having performance problems. The list of possibilities on a modern complex system is extensive.

Historical performance data is crucial in eliminating as many variables as possible. This means that you should collect operating system, database, and application statistics starting from the system's testing phase onwards, or minimally from the first day an application is rolled out into production. This applies even if the performance is unsatisfactory.

---

<sup>1</sup> For a full description of the Oracle Performance Method, please see the Oracle<sup>9i</sup> Performance Methods manual.

As the application stabilises and the performance characteristics are better understood, a set of statistics become the baseline for future reference. These statistics can be used to correlate against a day when performance is not satisfactory, and can assist in quantifying subsequent improvements made. They are also essential for future capacity and growth planning.

Oracle statistics are queried from the v\$ views using a snapshot method such as Statspack. Statistics which should be gathered include:

- Wait events
- SQL statistics and SQL Plans
- Overall systems statistics (shared pool, buffer cache, resource such as latches, locks, file IO)

## **THE PERFORMANCE METHOD - REACTIVE**

The reactive component of the Oracle performance method is often referred to as the Performance Improvement Method. This method is markedly different when compared to many commonly described performance approaches. Its focus is to identify and characterise the main performance bottleneck using the Oracle wait event interface, devise a sound theory based on additional evidence gathered, then propose remedial action.

To contrast, the current most widely known method<sup>2</sup> advocates tuning by a rigorous but statically ordered checklist of items, with the decision making process based solely on the outcome of computed ratios. Due to the orderly nature of this method and its lack of attention to the major waits in the system, it may take some time to uncover the real bottleneck.

## **STEPS IN THE ORACLE PERFORMANCE IMPROVEMENT METHOD**

This method describes a technique which will allow you to develop a feel for the problem by examining related evidence, noting irregularities, and making observations. It encourages saving judgments as to the cause of the performance problem until significant evidence has been gathered. Most times, a reliable instinctive feel is developed over time, by encountering and solving performance problems, and most importantly, by being aware of *what is normal at your site*.

1. Get an accurate description of the problem
2. Check the OS for exhausted resources
3. Gather database statistics
4. Check for the top ten most common pitfalls
5. Analyze the data gathered, focus on wait events, theorize the cause of the problem
6. Propose a series of remedial actions, then implement them
7. Measure the effect of the changes
8. Repeat any steps, as required, until performance goals are met

## **DESCRIPTION OF THE STEPS IN THE ORACLE PERFORMANCE IMPROVEMENT METHOD**

Below are the steps to take when solving a performance problem.

1. Get an accurate description of the problem.

Proposing or making changes before identifying the scope of the problem can be dangerous, and will most likely not resolve the problem. The information to collect includes:

- The scope of the problem. Is the problem instance wide, or local to a user or group of users, or a program? When did the system last perform well? If the problem is local to a specific program, look at the program

---

<sup>2</sup> A reason why the most commonly known performance method is popular, is it has been around for a long time, devised in Oracle V6 days, when this was the only data Oracle provided. The wait event views which were introduced in Oracle Version 7, greatly reduced the importance of the statistics described in the Version 6 method.

code, use `sql_trace` with `explain plan`<sup>3</sup> to identify why the code is behaving sub-optimally. Application specific tuning is not covered in this document. For more information, see the Oracle9i Performance Guide and Reference.

- The expected response time or throughput. This can also be considered the critical success factor. This measure will help in determining the effect of any changes, and also determine when to stop tuning.
- The time frame. When does the problem occur?
- Any recent changes in data volume, configuration changes (even seemingly inconsequential ones such as index rebuilds, new Optimizer statistics, minor application modifications), software/hardware upgrades.

## 2. Check the OS for exhausted resources.

Look at the operating system to see if any of the resources are severely limited. This data is integral to support or refute theories on why performance degraded, and sometimes helps narrow down the scope of the problem.

However, making assumptions based purely on OS statistics may misrepresent the problem. For example, there may be no free CPU available (i.e. 100% used), leading one to believe more CPU is required, when actually the excess CPU usage may be caused by unscalable SQL, execution plan changes, or run away processes. [See Sanity Check the OS in the Reference section for details on what to check]

## 3. Gather database statistics.

When tuning an Oracle system, the Oracle database statistics indicate the primary bottleneck, and so the first place to start tuning in earnest. For example, if the OS statistics indicate there is a disk with an IO bottleneck, but the Oracle statistics indicate the primary problem is latch related, then the primary bottleneck is latch related; this is the first problem to investigate.

Optimally, the site has been proactive and has gathered both database and OS statistics from when the system was performing acceptably well. If not, start gathering statistics now. [See `spdoc.txt` in the `rdbms/admin` directory for information on how to gather Oracle statistics, and your Operating System documentation for how to gather OS statistics]

## 4. Check for the top ten most common pitfalls.

The list of pitfalls represent the most commonly encountered tuning problems affecting an application's performance. Determine if any of these are likely to be the problem, and list these as symptoms for analysis later. [See Top Ten Pitfalls to Avoid in the Reference section]

## 5. Analyze the data gathered.

Find patterns. Build a conceptual model of what is happening on the system using the symptoms as clues to understand what caused the performance problems. [A example for doing this is covered in Using Statspack to Identify Bottlenecks in the next section]

## 6. Propose a series of remedial actions, the anticipated result in the system, then implement them.

Apply the remedial actions in the order that will benefit the application the most. A golden rule in performance work is that you only change one thing at a time, and then measure the difference. Unfortunately, system downtime requirements might prohibit such a rigorous investigation method. If multiple changes are applied at the same time, then try to ensure that they do not overlap in the same area.

## 7. Measure the effect of the changes.

Check the changes made have had the desired effect using statistics, and most importantly the user's perception of the performance improvement (if any). If performance is not acceptable and the implemented change has not had the desired effect, then the change implemented may not have been the complete solution. Re-examine the statistics, reconsider your understanding of the problem, attempt alternative methods to build a better conceptual model of how the application works, and the problem at hand. If the implemented change did decrease the wait

---

<sup>3</sup> Note in Oracle9i, the actual Optimizer plan used can be queried from the `v$sql_plan` view; this data is captured in level 6 and above snapshots by Statspack, and is visible in the new SQL report `sprepsql.sql`

time for the specified events but performance is still not as desired, cross-check the problem definition to see whether it has changed, then find the next bottleneck; continue refining the conceptual model of the application until your understanding of the application becomes more accurate, and performance is improved.

#### 8. Iteration.

It may be necessary to repeat some of these steps while looking for the culprits. The steps should be repeated until performance goals are met or become impossible due to other constraints. This process is iterative, and it is possible that some investigations will be made that have little impact on the performance of the system.

### USING STATSPACK TO IDENTIFY BOTTLENECKS

Statspack provides a simple way of collecting Oracle performance data, and of identifying bottlenecks. The reports pre-compute many useful statistics, and eliminates misleading statistics. There are two Statspack reports.

The first, `spreport.sql`, is an overall instance performance report. The first page of this report contains an instance performance summary, which concentrates a complete view of instance health. Subsequent pages include sections which report detailed statistics on the various tuning areas. The instance performance report is used when investigating an instance-wide performance problem.

The instance performance summary page always indicates the biggest bottleneck to investigate, and hence *the place* to start when examining Oracle performance data. The summary page is broken down into these areas (in order of importance):

- Top 5 Wait Events
- Load Profile
- Instance Efficiency

The remaining sections of the Statspack instance report are used to gather additional data. The high-load SQL sections are always scanned irrespective of the problem, as they provide insight into the SQL executing at the time the problem occurred, and the application in general.

If a level 5 or above snapshot is taken, the SQL report (`sprepsql.sql`) which is new in Oracle9i, provides in-depth information for a single SQL statement. The SQL report includes all of the SQL statistics for a particular hash value, the complete text of the SQL statement. Additionally, if a level 6 snapshot or above was taken the SQL execution plans are also included. This report is frequently used to tune problems identified as local to a specific program.

This remainder of this paper focuses on how to use the instance report (`spreport.sql`).

### STATSPACK STRATEGY

- Use the *Top 5 Wait Events* on page 1 to identify the events with most wait time by percentage<sup>4</sup>.

These events are preventing the majority of server processes from being productive, and so are likely<sup>5</sup> the bottleneck. Check whether the top events are related. Are the events consistent with any OS statistics?

There may be one event which greatly outranks the others, in which case this should be considered the bottleneck; focus on this event. Alternatively, there may be a set of related events which again indicate one primary area of contention (for example, the top three events may all be IO related, which may indicate a SQL tuning issue or IO bottleneck). A third possibility is there are a set of disjoint events which may rank closely for the greatest wait time. In this case, you may want to look at each one in turn, starting with the top event.

---

<sup>4</sup> Idle events are omitted from this list. Also, if `timed_statistics` is true, the events are ordered by the amount of time each event was waited for; this ordering gives the best indication of where most of the time was lost, and therefore where the biggest benefits can be gained. If `timed_statistics` is false, the order is by the number of waits. Oracle recommends setting `timed_statistics` to true for best performance diagnosis.

<sup>5</sup> Note that in a healthy, well performing system, the top wait events are usually IO related. This is an example of a case where the statistics alone do not accurately indicate whether there is a problem, which is why the most important indicator of performance is user perception.

Ignore events in the Top 5 listing which do not comprise a significant portion of the wait time.

- Gather additional data.

The purpose of gathering additional data is to help build up an understanding of the characteristics of the instance and to identify the application code executing, at the time the problem occurred. This information provides context to the problem, and gives you a good feeling for what was occurring on the system as a whole at that time. Gathering additional data usually requires skipping backwards and forwards through the report to check statistics which may be of interest. Initially, additional data is gathered up-front from the summary page. This context helps you better identify the importance of additional drill-down data which is examined in the next step. The data gathered may portray a consistent picture of one bottleneck, or a series of bottlenecks. It is likely you will find interesting data along the way which is not related to the primary bottleneck in the system; in this case note the data for future attention, but ignore it, as it will not help in significantly in this tuning effort.

- Begin gathering additional data. Look at:
  - the *Wait Events* and *Background Wait Events* sections for the average wait time for the highest ranking events (this column is identified by the heading *Ave wait(ms)*). This data can sometimes provide insight into the scale of the wait. If it is relevant to do so, also cross-check the event times with any other applicable Statspack or OS data. For example, if the events are IO related, what are the average read times, and are they acceptable? Is the Oracle data consistent with the OS read times, or does the OS data indicate the disks containing the datafiles are overly busy?
  - the *Load Profile* and *Instance Efficiency* sections on page 1, specifically at any statistics or ratios which are related to the top wait events. Is there a single consistent picture? If not, note other potential issues to investigate while looking at the top events, but don't be redirected away from the top wait events. Scan the other statistics. Are there any statistics in the *Load Profile* which are unusually high for this site, or any ratios in the *Instance Efficiency* section which are atypical for this site (when compared to a baseline)?
  - Also note that it is vital to examine the SQL sections of the Statspack report, to identify what the application was requesting of the instance which caused this performance regression. The SQL sections also sometimes identify tunable high-load SQL, or SQL statements which are avoidable<sup>6</sup>.
- Drill-down for additional data to the appropriate section in the Statspack report.

The relevant sections to examine are indicated by the top wait event. Some examples: if the top events are IO related (e.g. *db file sequential read* and *db file scattered read*), look at the *SQL ordered by Reads*, and the *Tablespace IO Stats*, and *File IO Stats* sections. If the top event is latch related (*latch free*), then look at the *Latch Activity*, *Latch Sleep Breakdown* and *Latch Miss Sources* sections of the report to identify which latch or latches are contended for.

While drilling down, determine whether the data in these sections is consistent with the wait events? If not, identify related areas in the report for an alternative data source. Extract other information from the drill-down data (questions to ask include: Are the number of times a resource was used high or low? Are there any related resources which when pieced together form a pattern to indicate a specific problem?)

- Note that in some situations there may not be sufficient data within the Statspack report to fully diagnose the problem; in this case, it is necessary to gather additional statistics manually.

By this stage, candidate problems and contended-for resources have been identified (with the highest priority issues dictated by the top-5 wait events). This is when the data should be analyzed. Consider whether there is there sufficient data to build a sound theory for the cause and resolution of the problem.

Use the Oracle9i Performance Guide and Reference as a resource to assist identifying the cause, and the resolution of the contention. The manual includes detailed descriptions of how to:

---

<sup>6</sup> An avoidable SQL statement is one that does not need to be executed, or does not need to be executed as frequently (e.g. one report indicated the application ran 'select sysdate from sys.dual' 40 times for each transaction).

- diagnose causes and solutions of Wait Events
- optimally configure and use Oracle to avoid the problems discovered

Even while following the strategy outlined above to analyze the problem, it is still possible to fall into the traps outlined in Performance Tuning Wisdom.

### PERFORMANCE TUNING WISDOM

Below are a list of traps, and some wisdom which may help you find a faster, or more accurate diagnosis.

- Don't confuse the symptom with the problem  
(e.g. `latch free` wait event is a symptom, not the problem)
- No statistic is an island i.e. don't rely on a single piece of evidence in isolation to make a diagnosis  
(e.g. the buffer cache hit ratio can often be misleading; similarly with other rolled-up statistics)
- Don't jump to conclusions
- Don't be sidetracked by irrelevant statistics (there are lots of them)  
(out of the 255 `V$SYSSTAT` statistics, there are approximately 15 being useful for 99% of issues)
- If it isn't broken, think twice before fixing it (or don't fix it at all)
- Don't be predisposed to finding problems you know the solutions to - this is known as *the old favourite*.  
(i.e. The problem identified and evidence sought is related to a favourite issue encountered previously, for which there is a well known solution, rather than looking for the bottleneck. Usually, the required evidence will be found to support the preconception)
- Be wary of solutions that involve fixing a serious performance problem by applying a single, simple change.  
(This usually involves setting an `init.ora` parameter, with `_underscore` parameters being popular. This type of solution is known as a *silver bullet*)
- Make changes to a system only after you are certain of the cause of the bottleneck  
(If you do make changes hastily, in the worst case performance will degrade)
- Many times, modifying the application results in significantly larger and longer term performance gains, when compared to solutions based on tweaking `init.ora` parameters<sup>7</sup> (rejection of this actuality is often accompanied by the search for a silver bullet solution)
- Removing one bottleneck may result in the dynamics of the instance changing (a good reason not to implement multiple changes at once), and hence the next bottleneck to be solved may *not* be the current *second in the list*

In summary, solving a performance problem similar to playing a game which is a cross between solving a whodunnit murder mystery and snakes and ladders. Be prepared to:

- be redirected by the evidence to the relevant statistics
- be surprised, and discover something new
- gather evidence along the way, without making hasty judgments

You might follow some blind alleys and find yourself looking at data you may not have initially expected to use.

---

<sup>7</sup> Modifying `init.ora` parameters is the fastest and easiest way to *try* to fix a performance problem. Unfortunately, it is rarely the solution. Typically, modifying `init.ora` parameters only helps in cases where the instance has been severely under or over-configured. Only in these cases will modifying initialization parameters cause large performance gains. Comparatively speaking, it may be possible to improve instance performance by 40% by modifying `init.ora` parameters, and 400% by modifying the application.

## EXAMPLE OF USING STATSPACK REPORT

The following example comes from a scalability benchmark for a new OLTP application. Excerpts of the Statspack report have been included, rather than the complete report. The requirement was to identify how the application could be modified to better support a greater concurrent user load. The instance was running Oracle8i release 8.1.7, and the Statspack snapshot duration was 50 minutes.

The intent of using a specific Statspack report is not to identify how to fix these specific problems encountered for this application, rather the aim is to provide an example of the technique described above.

### IDENTIFY THE LARGEST WAIT EVENTS, BY TIME STARTING IN THE TOP-5

#### Top 5 Wait Events

Event	Waits	Wait Time (cs)	% Total Wt Time
enqueue	482,210	1,333,260	36.53
latch free	1,000,676	985,646	27.01
buffer busy waits	736,524	745,857	20.44
log file sync	849,791	418,009	11.45
log file parallel write	533,563	132,524	3.63

#### Observations:

In this case, the most significant wait events are distributed over *enqueue*, *latch free*, and *buffer busy waits*, with *enqueue* being the most prominent, as they account for the majority of the high-wait time events. There is no obvious correlation between these events, which may mean there are three separate problems to investigate.

The most important wait event to investigate is *enqueue*.

- Problem 1: *enqueue*: Drill down to the *Enqueue activity* data.

Although the only focus should be on the primary bottleneck, for this example, we will go through each of the top-3 events to illustrate the method. In real-life situations it is sometimes necessary to investigate the next most important issues, *while the solution for the first bottleneck is being addressed*. In this example, the next major events are:

- Problem 2: *latch free*: Check the *Latch Activity* and *Latch Misses* sections to determine which latch, then determine why it is being used.
- Problem 3: *buffer busy waits*: Check the *Buffer Waits* section and the *Tablespace IO* and *File IO* sections to identify which files and buffers? Then determine why?

Before jumping to the drill-down sections, gather additional background data which will provide context for the problem, and details on the application.

### ADDITIONAL INFORMATION - WAIT EVENTS DETAIL PAGES

Check the *Wait Events* and *Background Wait Events* detail sections for the average time waited (*Avg wait (ms)*) for the top events. Identify the magnitude of the wait, and if possible, compare Oracle statistics to any relevant OS data (e.g. check whether the average read time for the event *db file sequential read*, and the OS read time for the disk containing that datafile are consistent, and as expected).

Event	Waits	Timeouts	Total Wait Time (cs)	Avg wait (ms)	Waits /txn
enqueue	482,210	50	1,333,260	28	0.8
latch free	1,000,676	751,197	985,646	10	1.6
buffer busy waits	736,524	3,545	745,857	10	1.2
log file sync	849,791	13	418,009	5	1.4
log file parallel write	533,563	0	132,524	2	0.9

SQL*Net break/reset to clien	535,407	0	20,415	0	0.9
db file sequential read	22,125	0	6,330	3	0.0
...					

Observations:

- None of the wait times are unusual, and in this case there are no relevant OS statistics to compare against.

**ADDITIONAL INFORMATION - LOAD PROFILE**

Examine other statistics on the summary page, beginning with the load profile.

Load Profile

	Per Second	Per Transaction
Redo size:	1,316,849.03	6,469.71
Logical reads:	16,868.21	82.87
Block changes:	5,961.36	29.29
Physical reads:	7.51	0.04
Physical writes:	1,044.74	5.13
User calls:	8,432.99	41.43
Parses:	1,952.99	9.60
Hard parses:	0.01	0.00
Sorts:	1.44	0.01
Logons:	0.05	0.00
Executes:	1,954.97	9.60
Transactions:	203.54	
% Blocks changed per Read:	35.34	Recursive Call %: 25.90
Rollback per transaction %:	9.55	Rows per Sort: 137.38

Observations:

- This system is generating a lot of redo (1mb/s), with 35% of all blocks read being updated.
- Comparing the number of *Physical reads per second* to the number of *Physical writes per second* shows the physical read to physical write ratio is very low (1:49). Typical OLTP systems have a read-to-write ratio of 10:1 or 5:1 - this ratio (at 1:49) is quite unusual.
- This system is quite busy, with 8,432 *User calls* per second.
- The total parse rate (*Parses per second*) seems to be high, with the *Hard parse* rate very low, which implies the majority of the parses are soft parses. The high parse rate may tie in with the latch free event, if the latch contended for is the library cache latch, however no assumptions should be made.

On the whole, this seems to be a heavy workload, with many parses, and writes.

**ADDITIONAL INFORMATION - INSTANCE EFFICIENCY**

Instance Efficiency Percentages (Target 100%)

Buffer Nowait %:	98.56	Redo NoWait %:	100.00
Buffer Hit %:	99.96	In-memory Sort %:	99.84
Library Hit %:	99.99	Soft Parse %:	100.00
Execute to Parse %:	0.10	Latch Hit %:	99.37
Parse CPU to Parse Elapsed %:	58.19	% Non-Parse CPU:	99.84

Shared Pool Statistics	Begin	End
Memory Usage %:	28.80	29.04
% SQL with executions>1:	75.91	76.03

% Memory for SQL w/exec>1: 83.65 84.09

Observations:

- The 100% soft parse<sup>8</sup> ratio indicates the system is not hard-parsing. However the system is soft parsing a lot, rather than only re-binding and re-executing the same cursors, as the *Execute to Parse %* is very low. Also, the CPU time used for parsing is only 58% of the total elapsed parse time (see *Parse CPU to Parse Elapsed*). This may also imply some resource contention during parsing (possibly related to the *latch free* event?).
- There seems to be a lot of unused memory in the shared pool (only 29% is used). If there is insufficient memory allocated to other areas of the database (or OS), this memory could be redeployed.

**ADDITIONAL INFORMATION - SQL SECTIONS**

It is always a good idea to glance through the SQL sections of the Statspack report. This often provides insight into the bottleneck at hand, and may also yield other (less urgent) tuning opportunities.

SQL ordered by Gets for DB: XXX Instance: XXX Snaps: 46 -48  
 -> End Buffer Gets Threshold: 10000

Buffer Gets	Executions	Gets per Exec	% Total	Hash Value
8,367,163	766,718	10.9	16.4	38491801
INSERT INTO EMPLOYEES VALUES (:1, :2, :3, :4, :5, :6)				
3,695,306	798,317	4.6	7.2	1836999810
SELECT DEPARTMENT FROM PAYROLL WHERE COST_CENTER = : "SYS_B_00" AND FNO = : "SYS_B_01" AND ORG_ID != : "SYS_B_02"				
...				
2,951,100	65,580	45.0	5.8	2714675196
select file#, block# from fet\$ where ts#=:1 and file#=:2				
1,377,986	275,327		5.0	3906762535
SELECT CC_CODE.NextVal FROM dual				
1,294,248	258,595		5.0	3303454348
SELECT SLARTI.NextVal FROM dual				

Observations:

- The majority of the SQL statements seem to have good executions plans (i.e. be tuned), as they do not require many logical reads.
- Most of the activity was INSERT or SELECT, with significantly fewer updates and deletes, looking at the number of executions.
- The modification of fet\$ (this is the data dictionary Free ExTent table), implies there is dynamic space allocation. This has been executed 65,000 times during the report interval of 50.48 minutes, which is on average, 21 times per second! This can easily be avoided, and should be (note this is not the current problem, so this is noted for future reference).
- There is a lot of sequence number use.

SQL ordered by Executions for DB: XXX Instance: XXX Snaps: 46 -48  
 -> End Executions Threshold: 100

<sup>8</sup> For definitions of hard parse, and soft parse, please see the Oracle9i Performance Guide and Reference.

Executions	Rows Processed	Rows per Exec	Hash Value
766,718	536,313	0.7	38491801
INSERT INTO EMPLOYEES VALUES (:1, :2, :3, :4, :5, :6)			
275,327	275,305	1.0	3906762535
SELECT MANAG_MRD_ID_SEQ.NextVal FROM dual			
275,327	275,305	1.0	341630813
INSERT INTO MANAG_MRD_IDENTITY VALUES (:1, :2, :3, :4, :5, :6, :7, :8, :9, :10, :11, :12, :13, :14, :15, :16, :17, :18, :19, :20, :21, :22, :23, :24, :25, :26, :27, :28, :29, :30)			
258,595	258,566	1.0	3303454348
SELECT PROD_ID_SEQ.NextVal FROM dual			
258,595	258,561	1.0	4222973129
INSERT INTO PROD_IDENTITY VALUES ( :1, :2, :3, :4, :5, :6, :7, :8, :9, :10, :11, :12, :13, :14, :15, :16, :17, :18, :19, :20 )			
914	914	1.0	1425443843
update seq\$ set increment\$=:2,minvalue=:3,maxvalue=:4,cycle#=:5,order\$=:6,cache=:7,highwater=:8,audit\$=:9 where obj#=:1			

Observations:

- There was nothing of any interest in *SQL ordered by reads* section, which implies the application is well tuned to avoid unnecessary physical IO.
- A significant proportion (30%) of the INSERTs into the EMPLOYEES table are failing, which is evident by comparing the number of *Rows Processed* to the *Executions*.
- Many of the INSERTs are executed the same number of times as the SELECT from a similarly named sequence number. This implies the sequence number is used as a column value in the insert. Possibly a key value?
- The update of seq\$ (this is the SEQUENCE number data dictionary table) is performed 18 times per minute, which is once every 3 seconds. Unless this data coincides with the SQ (SeQUENCE number) enqueue being contended for, this is not the largest bottleneck. However, for additional efficiency it may be useful to consider increasing the cache size for frequently modified sequence numbers; note this for future attention.

**PROBLEM 1 - DRILL DOWN TO: ENQUEUE**

Enqueue activity for DB: XXX Instance: XXX Snaps: 46 -48  
 -> ordered by waits desc, gets desc

Enqueue	Gets	Waits
TX	1,391,927	50

Observations:

- Unfortunately, the data here does not provide any additional insight, as it is inconsistent with the wait event data. Pre-Oracle9i enqueue statistics may not always provide the information required due to the manner in which the statistics were incremented (this report is from an 8.1.7 instance. The enqueue statistics in Oracle9i have been significantly improved).
- Look for other evidence in areas such as the Dictionary cache and SQL statements.

- Check whether there is any evidence of ST (Space Transaction) enqueue contention, either due to dynamic space allocation for permanent segments, or for SQL workarea segments (such as sort, bitmap merge and hash join workareas). Check to see if all users are using the temporary tablespace, and check to see if this tablespace is really temporary (locally managed, using tempfiles).
- As we saw by scanning through the *SQL ordered by Gets* report, the space management SQL statement was executed 65,000 times, therefore it is possible that some of the enqueue gets may be attributable to dynamic space allocation (ST enqueue).
- Before making deductions as to the cause, perform online monitoring of the system while the problem is occurring. There are a number of different methods for doing this.

Check the v\$lck table to see which locks are being waited for, and identify the SQL statements which are waiting for the enqueue. This will provide a better indication of the problem. Poll v\$lck while the problem is occurring to check the types of locks contended for:

```
select * from v$lck where request != 0
```

ADDR	SID	TY	ID1	ID2	LMODE	REQUEST	CTIME
C0000000444DCA30	9	TX	1245226	14284	0	4	0
...							
C0000000444DC8E0	137	TX	1245226	14284	0	4	0

```
select ses.sql_hash_value, sql.sql_text
  from v$session ses
       , v$sql      sql
 where ses.sid IN (SELECT sid FROM v$lck WHERE request != 0)
       and ses.sql_hash_value = sql.hash_value;
```

SQL_HASH_VALUE	SQL_TEXT
38491801	INSERT INTO EMPLOYEES VALUES (:1, :2, :3, :4, :5, :6)

Alternatively, it is also possible to query v\$session\_wait to find the enqueues contended for. The p1 column in v\$session\_wait contains the enqueue type and mode<sup>9</sup>.

Analysis:

- The solution will depend on the type of enqueue waited for, and the mode requested. In this case, the enqueue was the TX (Transaction) enqueue, in mode 4 (which is Share mode). The course of action here is to identify the reason why this enqueue is required so frequently, then reduce the need to obtain the enqueue. The Oracle9i Performance Guide and Reference documents the causes and solutions of the most frequently encountered wait events.

The possible causes for waiting for a TX enqueue in Share mode are:

- there may not be sufficient ITL (Interested Transaction List) entries in the data blocks to support the required degree of concurrency. This is most probably *not* the problem, as the contended for statements are INSERTS, and they typically use new blocks (evidenced by the low physical read rate on page 1 of the report). As it is not possible for an unused block to be full, Oracle is able to dynamically add more ITLs as needed.
- waiting for a transaction PREPARED by the XA TP monitor. The transaction is expected to either commit or rollback so the sessions wait in mode 4. This is not applicable here, as the application does not use XA.

---

<sup>9</sup> For more information on using v\$session\_wait for enqueue diagnosis, please refer to Metalink

- there are sessions waiting due to shared bitmap index fragment. Bitmapped indexes are not intended for high concurrency high INSERT applications. If this is the case, determine whether a different index type is more appropriate. This is not applicable, as the system only uses B\*Tree indexes.
- there are sessions waiting for other sessions to either commit or rollback. This can happen when the sessions are both attempting to insert what would be duplicate UNIQUE index entries. The first session has an exclusive TX lock, and the other sessions attempting to insert the 'duplicate' are waiting for the first session to commit (in which case a duplicate key in index error is returned), or to rollback (in which case the session will then insert its row). This wait is performed using a TX lock in Share mode. This is potentially a cause, so the next step should be to identify whether this is a viable explanation using additional data.

Reconsidering information gathered earlier, the *SQL ordered by executions* section indicated a significant number (30% which is 230,405) of the INSERTs into the EMPLOYEES table failed - this is the same table mentioned in the INSERT statements which were waiting for the TX mode S lock. This data provides evidence that this is the cause of the problem. Check the rollback related statistics from the *Instance Activity Stats* section to see how frequently the system is rolling back, whether the rollbacks are explicit or implicit, and how much work is being performed to rollback:

```
Instance Activity Stats for DB: XXX Instance: XXX Snaps: 46 -48
```

Statistic	Total	per Second	per Trans
rollback changes - undo records a	1,008,419	332.9	1.6
transaction rollbacks	330,945	109.3	0.5
user commits	557,671	184.1	0.9
user rollbacks	58,854	19.4	0.1

The *transaction rollbacks* statistic indicates the actual number of rolled back transactions that involve undoing real changes (i.e. the number of explicit and implicit rollbacks). Contrast with *user rollbacks* statistic which only indicates the number of ROLLBACK statements executed (i.e. the number of explicit rollbacks). The value for *transaction rollbacks* per second is very high, as is the number of undo records applied (*rollback changes - undo records applied* statistic). This data provides substantial evidence that the application is generating duplicate index keys, forcing Oracle to implicitly rollback many of the inserts.

It is also interesting to note that the amount of redo generated per second will also decrease if the transaction rollbacks can be reduced. In other words, this will drop the original high redo-generation rate observed on the summary page of the report.

- From the evidence, it is almost certain the enqueue wait event is caused by the application generating duplicate unique keys. For optimal scalability and throughput, the application should be redesigned to avoid generating duplicate unique key values.

## PROBLEM 2 - DRILL DOWN TO: LATCH FREE

Look at the latch sections of the Statspack report for information on which latch or latches are being waited for under the *latch free* event.

```
Latch Activity for DB: XXX Instance: XXX Snaps: 46 -48
->"Get Requests", "Pct Get Miss" and "Avg Slps/Miss" are statistics for
    willing-to-wait latch get requests
->"NoWait Requests", "Pct NoWait Miss" are for no-wait latch get requests
->"Pct Misses" for both should be very close to 0.0
```

Latch Name	Get Requests	Pct Get Miss	Avg Slps /Miss	NoWait Requests	Pct NoWait Miss
-----	-----	-----	-----	-----	-----

cache buffers chains	142,028,625	0.3	0.4	1,193,834	0.7
cache buffers lru chain	8,379,760	0.1	0.7	414,979	0.1
dml lock allocation	3,742,520	0.4	0.3	0	
enqueue hash chains	7,417,793	4.7	0.4	0	
enqueues	5,318,393	1.4	0.2	0	
latch wait list	308,922	0.9	0.6	312,484	0.4
library cache	36,870,207	2.1	0.7	0	
multiblock read objects	4	0.0		0	
ncodef allocation latch	53	0.0		0	
redo allocation	10,478,825	0.9	0.3	0	
redo copy	336	97.9	1.5	9,410,548	0.7
redo writing	3,889,991	1.8	0.6	0	
shared pool	628,207	0.1	0.6	0	
undo global data	8,417,202	1.6	0.3	0	

Observations:

- The *cache buffers chains* latch has a large value for *Get Requests*, percentage of misses per get (*Pct Get Miss*) and an average number of sleeps per miss ratio (*Ang Slps/Miss*).
- The *enqueues* and *enqueue hash chain* latches have significant numbers for *Get Requests*, *Pct Get Miss*, and *Ang Slps/Miss*. The contention for these latches is most likely related to the *enqueue* wait event (Problem 1). When problem 1 is resolved, it is likely latch contention for these latches will also decrease correspondingly. In this case reducing the enqueue latch gets is probably not enough to remove *latch free* from the *Top 5 Wait Events*.
- The *library cache* has a significant get-miss percentage, and sleeps per miss ratio. Interestingly, the *Load Profile* data previously gathered indicated a high soft-parse rate. The combination of this evidence may indicate a parse-related problem.
- The *redo copy* latch is not a problem, even though there is a 97% miss rate. This ratio is misleading, as although the miss rate was 97%, there were only 336 gets, which makes the percentage irrelevant. This latch should be ignored. This is a good example which shows relying solely on a single derived statistic is not sufficient to lead one to a sound diagnosis of the problem.
- The *undo global data* latch has a large number of gets; this latch is obtained when attempting to roll back transactions. It is likely latch contention for this latch will disappear if the transaction rollback rate can be reduced (see problem 1).
- Checking the Latch Sleep breakdown section may provide clarity on which latch is most contended for.

Latch Sleep breakdown for DB: XXX Instance: XXX Snaps: 46 -48  
-> ordered by misses desc

Latch Name	Get Requests	Misses	Sleeps	Spin & Sleeps 1->4
library cache	36,870,207	758,365	509,064	458554/10267 3/186083/110 55/0
cache buffers chains	142,028,625	438,718	188,714	259998/16953 3/8462/725/0
enqueue hash chains	7,417,793	347,327	132,110	226711/10994 7/9895/774/0
undo global data	8,417,202	133,473	36,457	98396/33747/ 1281/49/0
redo allocation	10,478,825	89,983	29,479	61721/27089/ 1129/44/0
row cache objects	27,905,682	77,776	7,744	70162/7486/1 26/2/0
enqueues	5,318,393	72,449	14,836	57868/14327/ 253/1/0

Observations:

- In terms of number of *Misses* and *Sleeps*, the *library cache* latch is the most significant. This is based on comparing this latch's statistics to the latch with the next highest sleep count (which is the *cache buffers chains* latch): the *library cache* latch had 1/4 the number of gets, with 1.7 times the number of misses, and nearly 3 times the number of sleeps. Contention for this latch, along with the data previously gathered indicate soft parsing.
- A large number of latch requests are resulting in the sessions sleeping while attempting the latch get. Many performance engineers may look at the sleep counts, and if there is free CPU on the system, recommend increasing the *spin\_count* parameter. This is not the correct action. The value for *spin\_count* should not usually be modified from the default value. This is a parameter which is frequently misused as a *silver bullet*. Instead, the best course of action is to determine *why* the latches are being contended for, and if possible reduce the need to get the latches.
- The high soft-parse rate (page1), the *latch free* event appearing as event number two in the *Top 5 Wait Events* section (with a significant percentage), and the *library cache* latch having a high sleep and miss rate indicate the *library cache* latch is the most significant contributor to the latch free wait event. Although it may seem *the cache buffers chains* latch gets, misses and sleeps are significant, they are *not the cause of the current bottleneck*. This data should be noted for future reference, but ignored, until the current bottlenecks are resolved.
- More data is required. New in the Oracle9i Statspack report is a *SQL ordered by Parse Calls* section; this section is an appropriate section to examine for SQL statements which are being parsed frequently. As this is an 8.1.7 instance, this data was not available through the report, but could have been queried directly from the Statspack tables.
- Also check the *init.ora Parameters* section for any parse or shared pool related parameters. While paging to the end of the report for the *init.ora* parameters, also glance at the *library cache* statistics to see if there is anything unusual (in this case, there wasn't).

init.ora Parameters for DB: XXX Instance: XXX Snaps: 46 -48

Parameter Name	Begin value	End value (if different)
cursor_sharing	FORCE	
cursor_space_for_time	TRUE	
session_cached_cursors	50	
shared_pool_size	268435456	

Analysis:

The parameter *cursor\_sharing* is set to *FORCE*. If this parameter was set for the correct reason, this implies the application was issuing large amounts of unshared SQL which differed only in the literal values used. Judging by the current soft-parse rates, without this parameter the application would most likely have had a severe hard parse problem. Instead, by setting *cursor\_sharing*, the hard parse bottleneck has been eliminated in favor of a *more* scalable (but ultimately unscalable!) soft parse bottleneck.

Applications which re-use SQL, and which have a high soft parse rate optimally should not close frequently executed cursors, instead keep the cursors open and re-execute them. This *avoids* a lot of the overhead and resources required in soft-parsing the SQL statement, including *library cache* latch gets:

- *Library cache* latch gets are required during the execution phase in order for the cursor to be pinned while it is being executed, then unpinned at the end of the execute. The pin is required to avoid this cursor being aged out, while it is being executed. Note it is possible to avoid pinning/unpinning the cursor each time it is executed by keeping the cursor open, and by using the *init.ora* parameter *cursor\_space\_for\_time*.

The parameter *cursor\_space\_for\_time* avoids the need to pin a cursor before execution *if it is being held open by the session performing the execute*, as a pin is left there for a session, after the first execution.

Rechecking the parse related statistics on page 1 provides the following insight: the number of *Executes* per second is almost exactly identical to the number of *Parses* per second, which implies cursors *are being closed* and re-parsed for each execution. This means in the majority of cases, this application is *not deriving benefit* from the use of `cursor_space_for_time`. To get the benefit from setting `cursor_space_for_time`, the application should be modified to avoid closing cursors (the application should instead use new bind values and re-execute already opened cursors).

- When locating a pre-existing statement in the shared pool, Oracle must use a library cache latch. It is possible to avoid re-locating the SQL statement in the shared pool each time it is executed, simply by keeping the cursor open. No further action is required.

However, if it is not possible to modify the application to keep cursors open and so avoiding the soft-parse all together, it may be possible to reduce the overhead of searching for the SQL statement to execute in the shared pool, by using the `init.ora` parameter `session_cached_cursors`. Setting this parameter bypasses the need to find a SQL statement executed by this session in the shared pool, by keeping a reference to this statement's (shared pool) memory address, within session memory. If a statement to be executed is found in the session cursor cache, this in results in fewer *library cache* latch gets. Statements are added to a session's cursor cache when the cursor is closed by the session, and the cursor has been parsed at least three times (i.e. `v$sql.parse_calls` must be greater than or equal to 3).

This site already has this parameter set. It is possible the setting is not sufficiently high. Before suggesting increasing the size of the cursor cache, check to see the number of cursors already kept in the session cursor cache. If the majority of sessions consistently use fewer entries in the session cursor cache than the setting for `session_cached_cursors`, then there will be no value in increasing the parameter.

To determine the distribution of sessions for the different cache sizes:

```
select ss.value cursors_cached, count(*) no_of_sessions
  from v$sesstat ss
       , v$statname sn
 where sn.statistic# = ss.statistic#
       and sn.name = 'session cursor cache count'
 group by ss.value;
```

The output will look something like:

CURSORS_CACHED	NO_OF_SESSIONS
1	2
2	4
3	1
...	
47	11
48	2
49	2
50	560

This data indicates the majority of sessions (560) are using the entire cursor cache. In this case, it is likely increasing the session cursor cache count will help, by resulting in fewer library cache latch gets when finding the cursor to execute.

Analysis:

- The soft-parse rate (and associated latch resource usage) when executing the statement is impossible to significantly reduce further without modifying the application. Overall, as there are a lot of frequently executed SQL statements; it would be optimal to modify those statements to use bind variables, and to keep those cursors open in order to re-execute them.

**PROBLEM 3 - DRILL DOWN TO: BUFFER BUSY WAITS**

Identify the type of buffer waited for, and look to see whether the majority of waits occur in a specific tablespace or datafile.

Buffer wait Statistics for DB: XXX Instance: XXX Snaps: 46 -48  
 -> ordered by wait time desc, waits desc

Class	Waits	Tot Wait Time (cs)	Avg Time (cs)
data block	655,610	679,104	1
undo header	47,033	48,626	1
undo block	32,991	18,384	1

Observations:

- The most contended for buffers are data blocks.
- More data is required. Check the *Tablespace IO Stats* and *File IO Stats* to see whether there is a particular tablespace which has a higher buffer wait count.

Tablespace IO Stats for DB: XXX Instance: XXX Snaps: 46 -48  
 ->ordered by IOs (Reads + Writes) desc

Tablespace

	Av Reads	Av Reads/s	Av Rd(ms)	Av Blks/Rd	Av Writes	Av Writes/s	Av Buffer Waits	Av Buf Wt(ms)
INDX01	18,698	6	3.2	1.0	1,625,296	537	579,955	10.7
DATA01	3,434	1	1.2	1.0	491,925	162	76,491	8.1
...								

Observations:

The majority of the waits seem to be on the tablespace INDX01 (579,000 out of the total data block waits out of 680,000). Some possibilities to check for:

- unselective indexes used in frequently executed (or resource intensive) SQL statements. This does not look likely, as the SQL statements are highly tuned.
- 'right-hand-indexes' (i.e. indexes which are inserted into at the same point by many processes e.g. if the key is based on a sequence number). This may be the case here, as sequence numbers are incremented frequently, and often with the same frequency as related INSERT statements.
- As the most contended for tablespace contains indexes, and the *enqueue* waits also involved INSERTS, there is a possibility the *buffer busy waits* event may be related to the *enqueue* problem.
- More data is required. Examine the *Instance Activity Stats* section for index maintenance data:

Instance Activity Stats for DB: XXX Instance: XXX Snaps: 46 -48

Statistic	Total	per Second	per Trans
leaf node splits	95,846	31.6	0.2

This data indicates a very high per-second rate for index block splits (this statistic is usually negligible). The contents of an index block is split into two blocks when an entry must be placed in the block, and there is insufficient free space. This data indicates the cause of the buffer busy waits may be due to right-hand indexes. This is possible considering the heavy reliance on sequence numbers indicated in the *SQL ordered by Gets*. If sequence numbers are used as the leading portion of an index, and INSERT's using the sequence number are being inserted out of order, that this would account for the high *leaf node splits* statistic value.

- Optimally in this case to determine the type of operation serializing on the block and the contended for segment, poll v\$session\_wait, and examine the p1 (file number) p2 (block number) and p3 (reason identifier) columns, and also determine the SQL statement being executed while those waits are occurring the following statements should be executed:

```
select event, p1 file, p2 block , p3 reason_id, count(*)
  from v$session_wait
 where wait_time=0
       and event = 'buffer busy waits'
 group by event, p1, p2, p3;

select ses.sql_hash_value, sql.sql_text
  from v$session ses
       , v$sql      sql
 where ses.sid IN (SELECT sid FROM v$lck WHERE request <> 0)
       and ses.sql_hash_value = sql.hash_value;
```

In this case, the data from the above SQL statements was not available.

Analysis:

- The evidence gathered implies this problem is caused by the application using a sequence number as the leading column in the unique key generation:
  - the majority of buffer busy waits are on an index tablespace
  - the high-load SQL report shows a lot of sequence number use, with the frequency of INSERT and the frequency of use of sequence number generator being exactly the same
- The evidence gathered in problem 1 may also be relevant. It is possible (although less likely) that this *buffer busy waits* problem may be caused, or exacerbated by the duplicate key generation occurring.
- If problem 1 is fixed, and problem 3 continues to occur, there are a number of possible workarounds, including:
  - re-order the keys if the index is a concatenated index, and it is practical to do so (i.e. it may not be practical if execution plans will be negatively impacted)
  - use reverse key indexes (this will work well, if the system does not require range scans of this index)
  - use a different unique key which is not dependent on a monotonically increasing sequence number

## OVERALL ANALYSIS OF THE STATSPACK REPORT

- The application has very good execution plans for the most frequently executed statements
- The application's scalability is limited by the three factors identified: *enqueue*, excessive soft-parse rate, and *buffer busy waits*.
- The *enqueue* problem should be relatively simple to address. Once this has been resolved, it is highly likely the soft-parse rate will be the next bottleneck. If performance is still not acceptable, and if this benchmark is truly

indicative of the application and expected throughput, the application will need to be re-architected to use bind variables on the most frequently executed cursors, and to keep the cursors open for re-use.

## **REFERENCE SECTION**

The reference section includes the following topics:

- DELTAS
- WAIT EVENTS
- DIFFERENCE BETWEEN WAIT EVENT STATISTICS AND OTHER STATISTICS
- HOW TO DETERMINE HIGH RATES OF ACTIVITY
- HOW TO SANITY CHECK THE OPERATING SYSTEM
- TOP TEN PITFALLS TO AVOID

### **DELTAS**

As the majority of Oracle statistics are incremental over the life of an instance, in order to determine what activity was most prevalent at a specific time, you need to look at the statistics which cover that time interval. The easiest way is to capture the raw data at time A, and again at time B then take the delta of the statistics. This indicates how many times a certain event was waited for, or a statistic was incremented in that interval. This is the method Statspack and BSTAT/ESTAT use.

### **WAIT EVENTS**

Wait events are statistics which are incremented by a server process to indicate it had to wait during processing. A server process can wait for: a resource to become available (such as a buffer, or a latch), an action to complete (such as an IO), or more work to do (such as waiting for the client to provide the next SQL statement to execute). Events which identify that a server process is waiting for more work, are known as idle events.

Wait event statistics include the number of times an event was waited for, and the time waited for the event to complete. To minimise user response time, reduce the time spent by server processes waiting for event completion.

### **DIFFERENCE BETWEEN WAIT EVENTS AND STATISTICS**

A wait event is a special statistic which is only incremented when an Oracle session had to wait for an event to occur or resource to become free, before it was able to continue processing. Statistics for wait events are queryable from the following views v\$system\_event, v\$session\_event, and v\$session\_wait.

Non-wait event statistics are incremented each time the session performs a certain action. Statistics for non-wait events are queryable from many v\$ views. A small subset of these are v\$sysstat, v\$latch, v\$filestat, etc.

#### *EXAMPLES OF WAIT EVENTS:*

- *latch free*: a wait for latch free occurs when an Oracle session waited for a latch to become free
- *db file sequential read*: a wait for db file sequential read occurs when an Oracle session waited for a single<sup>10</sup> block read to complete
- *db file scattered read*: a wait for a scattered read occurs when an Oracle session waits for a multiblock read to complete

---

<sup>10</sup> Although confusing, a *db file sequential read* event indicates a single block read is occurring, and a *db file scattered read* indicates a multiblock read is occurring. This terminology is industry standard terminology, and can be explained thus: the term *scattered* relates to the multiple disk blocks being read *into* scattered (i.e. discontinuous) memory locations. Similarly, the term *sequential* indicates the block is being read *into* a single contiguous memory location (in this case, a single database buffer).

- *buffer busy wait*: a buffer busy wait occurs when an Oracle session waited for a specific buffer to become available for use
- *log file sync*: occurs when an Oracle session waits for a commit to be flushed to the redo log

EXAMPLES OF STATISTICS (FROM `V$SYSSTAT`):

- *physical reads*: the number of times Oracle requested a block to be read by the OS (usually bumping this statistic would also require bumping the appropriate read wait event e.g. *db file sequential read*, *db file scattered read*, *db file parallel read*, *direct path read*, *direct path read (lob)*)
- *session logical reads*: the number of times a block was accessed in the cache. If a the buffer requested was not already in the cache, this will also require bumping the physical reads statistic, along with the appropriate IO wait event
- *parse count (total)*: the number of times a SQL statement was explicitly, or implicitly parsed
- *physical writes direct (lob)*: the number of direct physical writes (i.e. bypassing the buffer cache) for LOBs

### HOW TO DETERMINE HIGH RATES OF ACTIVITY

It would be very useful if it were possible to have generalized recommendations on threshold limits for a high rate of activity which can validly be applied at all sites. Unfortunately this is not possible.

What may be a low rate as identified on some sites, will be enough to cause a performance problem on other sites. The converse is also true; a value which is identified as high on one site may not be a problem on another.

What determines the threshold limits includes the amount and speed of the available hardware, the hardware configuration, the Operating System, the Oracle release, the application, the number of concurrent requests, and the most prominent bottleneck.

In other words, these statistics can only be site-specific, and assuming a perfectly scalable system, are only limited by the quantity and speed of the available hardware on the site.

Instead, the performance engineer can develop an awareness of typical values on their site, by looking at the values for the statistics identified in the *Load Profile* section of Statspack reports. Various reports should be examined, which cover the typical high and low usage times - this provides high values (peak usage) and low values (off-peak usage). Thus when a performance problem arises, the normal values are known, so any values which deviate greatly from the norm should be considered suspicious.

A way to identify whether an unusual value may be contributing to the most prominent bottleneck is to ask the question *Is the statistic with the unusual value possibly related to the most prominent Oracle wait events?* If not, then this may not be 'too high' (at least, not for the moment).

### HOW TO SANITY CHECK THE OPERATING SYSTEM

1. Check the CPU utilization in user and kernel space for the system as a whole, and on each CPU.
  - What is the percentage of user time to kernel time? Reasonable OLTP values can include 70% user 30% kernel.
  - How much idle CPU is there? (note that Wait IO should be considered as Idle CPU on most operating systems). If there is no idle CPU, it is not usually of benefit to examine the wait events. First the CPU usage must be decreased.

Is the CPU used being consumed by one or many processes, and are they Oracle or other? If non-Oracle processes, then identify how these processes can be modified to reduce their CPU use. If the majority of CPU is being consumed by a small number of Oracle processes, identify the most commonly executed SQL in the system, and the specific SQL statements they are executing to determine whether it can be tuned. Typically in this case there may be a small number of poorly executing statements which are executed frequently, by all processes (e.g. a login environment script); these may be visible in the *SQL ordered by Gets* section of the Statspack report. If the CPU is being used by a one or a few select Oracle processes, identify the SQL statements they are executing at this time, and tune them (usually a mechanism such as SQL trace is useful to do this).

2. Confirm that there is no paging or swapping.
3. Check that network latencies between machines are acceptable. Network latency can be a large portion of the actual user response time.
4. Identify disks with poor response times or long queues. An overly active I/O system can be evidenced by disk queue lengths greater than two, or disk service times that are over 20-30ms. In either of these cases, put disk contention on the list of cross-check data to examine in the Statspack wait events and IO sections.
5. Confirm that there are no hardware errors. This ensures that there are no hardware problems which may be affecting response time.

### TOP TEN PITFALLS TO AVOID

For more information on the top-10 pitfalls to avoid, please see the Oracle9i Performance Methods manual. Below is a brief overview.

1. Bad Connection Management
 

The application connects and disconnects for each database interaction. This problem is common with stateless middle ware in application servers. It has over two orders of magnitude impact on performance, and is unscalable. The middle ware should maintain a pool of connections to avoid this issue.
2. Bad Use of Cursors and the Shared Pool
 

Not re-using cursors results in repeated parses. If bind variables are not used, then there is hard parsing of all SQL statements. This has an order of magnitude impact in performance, and is unscalable. Use cursors with bind variables that open the cursor once and re-execute it many times. Be wary of generating dynamic SQL for this reason.
3. Getting Database I/O Wrong
 

Many sites lay out their databases ineffectively over the available disks. Other sites specify the number of disks incorrectly, because they configure disks by the amount of disk space required, and not the required I/O bandwidth (i.e. the number of disks are chosen by the total MB required, rather than the number required to sustain the required IO throughput; this issue is becoming commonplace with the advent of larger and larger disks).
4. Redo Log Setup Problems
 

Many sites run with too few redo logs that are too small. Small redo logs cause system checkpoints to continuously put a high load on the buffer cache and I/O system. If there are too few redo logs, then the archive cannot keep up, and the database will wait for the archive process to catch up.
5. Serialization of data blocks in the buffer cache
 

Serialization of access to data blocks in the buffer cache can occur due to lack of free lists, free list groups, transaction slots (INITRANS), or shortage of rollback segments. This is particularly common on INSERT-heavy applications, in systems that have increased the block size to 8K or 16K, or in systems with large numbers of active users and few rollback segments.
6. Long Full Table Scans
 

Long full table scans for high-volume or interactive online operations could indicate poor transaction design, missing indexes, or poor SQL optimization. As long table scans are highly I/O intensive, increased concurrency of this operation results in an unscalable system.
7. In Disk Sorting
 

In disk sorts for online operations could indicate poor transaction design, missing indexes, or poor SQL optimization. Disk sorts, by nature are I/O-intensive and unscalable.
8. High Amounts of Recursive (SYS) SQL

Large amounts of recursive SQL executed by SYS could indicate space management activities, such as extent allocations, taking place. This is unscalable and impacts user response time. Recursive SQL executed under another user ID is probably SQL and PL/SQL, and this is not a problem.

#### 9. Schema Errors and Optimizer Problems

In many cases, an application uses too many resources because the schema is not optimal, or the Optimizer statistics gathered are not representative of the current data. These problems can lead to sub-optimal execution plans, and hence poor interactive user performance and/or excessive resource utilization.

Common examples are:

- too many indexes (duplicated leading columns), non-selective index column choices, missing indexes
- incomplete optimizer statistics, or statistics which are not longer representative of the data, or the omission of collecting histogram data when appropriate

Schema errors may occur if the schema was not successfully migrated from the development environment or from an older implementation. When migrating applications of known performance, export the schema statistics to maintain consistent execution plans by using the DBMS\_STATS package.

Likewise, optimizer parameters set in the initialization parameter file can override proven optimal execution plans. For these reasons, schema, schema statistics, and optimizer settings should be managed together as a group to ensure consistency of performance.

#### 10. Use of Nonstandard Initialization Parameters

These might have been implemented based on poor advice or incorrect assumptions. In particular, parameters associated with SPIN\_COUNT on latches and undocumented optimizer features can cause a great deal of problems that can require considerable investigation.

### **ACKNOWLEDGMENTS:**

The Oracle Performance Method was formalised and documented by the following people: Bjorn Engsig, Cecilia Gervasio, Connie Dialeris Green, Karl Haas, Andrew Holdsworth, Mamdouh Ibrahim, Anjo Kolk, Virag Saksena, Graham Wood. As always, with thanks to Richard Powell for his excellent Notes on Performance Tuning.