

Top 10 strategies for Oracle performance (part 3)

This is the third of a four part series covering my “top 10” performance strategies for Oracle databases. In [part one](#), we looked at methodology, database and application design, and indexing. In [part two](#), we covered the essential tuning tools, the SQL optimizer and strategies for tuning SQL and PL/SQL. In this third instalment we’ll cover contention, memory management and IO optimization.

Each of these topics are huge in scope, so I can only provide an introduction in this series. You can find more information on each in my book [Oracle Performance Survival Guide](#).

Strategy 7: Identify and minimize contention

When two Oracle sessions want to simultaneously access a resource, one of them may have to wait while the other performs it’s operation. The two sessions are *contending* for the resource. This contention limits the amount of work which the oracle database can perform – it’s the proverbial “database bottleneck”.

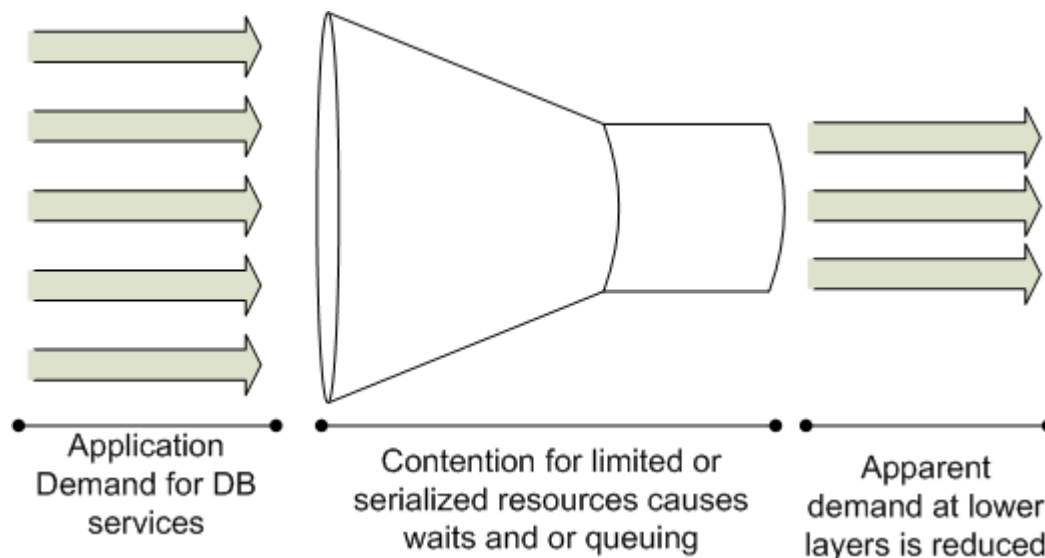


Figure 1 Contention is the proverbial Bottleneck

There are three main forms of contention that we are most concerned about in an Oracle database:

1. *Lock contention*, which occurs when sessions want to access or change the same table data.
2. *Latch contention*, which occurs when sessions want to change or access the same area of shared memory
3. *Buffer contention*, which occurs when sessions have to wait for a specific area of memory.

Lock Contention

The “ACID” (Atomic-Consistent-Independent-Durable) properties of transactions – especially the consistency characteristic – require that Oracle limit concurrent changes to table data. Locks are the mechanism by which Oracle implements these constraints.

Most locks are row level locks that prevent two sessions from changing the same row. And most of these row level locks occur as a result of DML issued by the application. Therefore, application design has a big impact on locking.

Measuring lock contention

Whenever an Oracle session needs to stop processing and wait for a lock – or for any other resource for that matter – it will record the wait in the various “wait interface” tables such as V\$SYSTEM_EVENT. We can interrogate these views to measure the extent of lock waits and hence to determine if we have a high level lock wait problem.

Lock waits – sometimes referred to as enqueue waits¹ - are identified by the ‘enq:’ prefix, which is then followed by the two character lock code. The two character lock codes are defined in the V\$LOCK_TYPE table. The wait identifier also includes a brief description of the wait and there can be more than one wait type for a particular lock code.

This query breaks out the lock waits and compares them to other high level wait categories and to CPU time. This query reveals amount of time spent waiting for locks relative to other activities:

```
SQL> WITH system_event AS
  2   (SELECT CASE WHEN event LIKE 'enq:%'
  3           THEN event ELSE wait_class
  4           END wait_type, e.*
  5   FROM v$system_event e)
  6 SELECT wait_type, SUM(total_waits) total_waits,
  7        round(SUM(time_waited_micro)/1000000,2) time_waited_seconds,
  8        ROUND( SUM(time_waited_micro)
  9              * 100
 10              / SUM(SUM(time_waited_micro)) OVER (), 2) pct
 11 FROM (SELECT wait_type, event, total_waits, time_waited_micro
 12        FROM system_event e
 13        UNION
 14        SELECT 'CPU', stat_name, NULL, VALUE
 15        FROM v$sys_time_model
 16        WHERE stat_name IN ('background cpu time', 'DB CPU')) l
 17 WHERE wait_type <> 'Idle'
 18 GROUP BY wait_type
 19 ORDER BY 4 DESC
 20 /
```

WAIT_TYPE	TOTAL_WAITS	TIME_WAITED_SECONDS	PCT
User I/O	4,140,679	11,987.46	33.56
System I/O	3,726,628	9,749.85	27.30
CPU		8,084.14	22.63
Other	69,987	1,380.59	3.87
Commit	684,784	1,327.28	3.72
enq: TX - row lock contention	13	1,218.91	3.41
enq: TM - contention	6	751.16	2.10
Concurrency	29,350	486.48	1.36
Configuration			

Application locking strategies

¹ Lock waits are referred to as enqueues because you wait in a queue to obtain the lock. This is in contrast to some other waits – latches in particular – for which there is no ordered queue.

Row level locks for application tables are usually a necessary consequence of transactional integrity and some degree of row level lock contention is to be expected. However, it's definitely an objective of application design to keep the amount of time spent waiting for locks to a minimum.

The essential principles of application lock management are:

- To place or acquire locks only when necessary.
- To minimize the amount of time the locks are held.

The techniques for adhering to these principles will vary from application to application. However, there are two common patterns of lock management which will have a fundamental impact on lock contention – the *optimistic* and *pessimistic* locking strategies.

The pessimistic locking strategy is based on the assumption that it is quite possible that a row will be updated by another user between the time you fetch it and the time you update it. To avoid any contention, the pessimistic locking strategy requires that you lock the rows as they are retrieved. The application is therefore assured that no changes will be made to the row between the time the row is retrieved and the time it is updated.

The optimistic locking strategy is based on the assumption that it is very unlikely that an update will be applied to a row between the time it is retrieved and the time it is modified. Based on this assumption, the optimistic locking strategy does not require that the row be locked when fetched. However, to cope with the situation in which the row *is* updated between retrieval and modification, it is necessary to check that the row has not been changed by another session before finally issuing the change.

The optimistic locking strategy tends to result in less lock contention than the pessimistic strategy, since locks are held for a briefer period of time. However, should the optimism be misplaced, the optimistic strategy will require that failed transactions be retried, resulting in an increase in overall transaction rates.

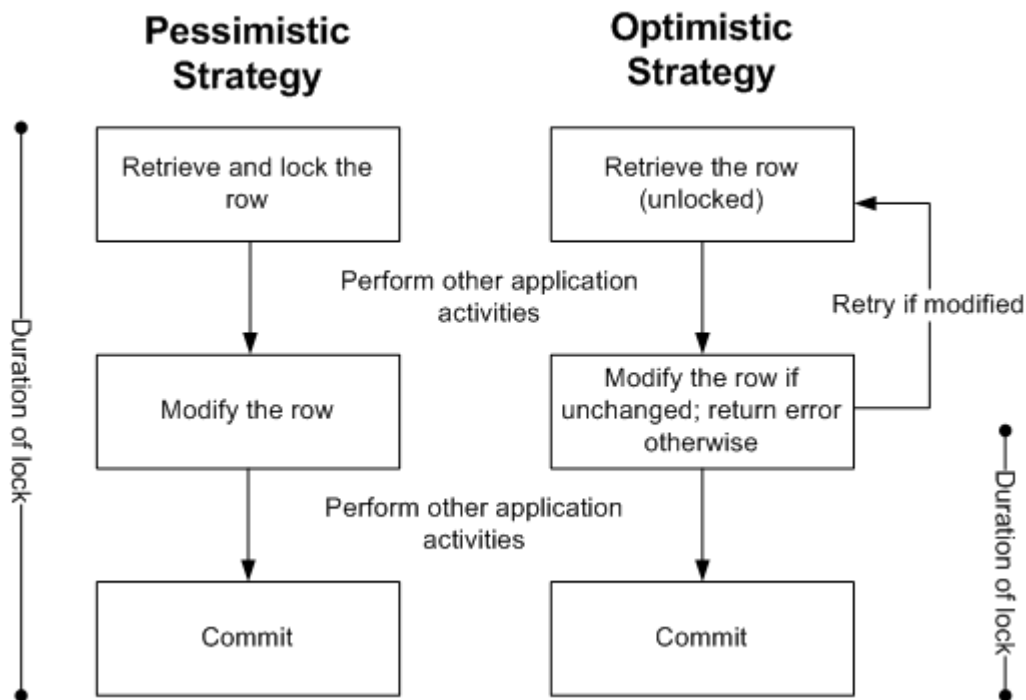


Figure 2 Optimistic and Pessimistic locking strategies

When row level locking fails

When determining an application locking strategy, it is fair to assume that Oracle's row-level locking strategy will result in locks being applied only to the rows that are updated, and not to entire tables or blocks of rows. However, there are well known circumstances in which Oracle's row level locking mechanisms can break down to block level or table level locks. Some of these circumstances include:

- un-indexed foreign key constraints on can result in table level locks being applied to the child table when a parent table row is deleted or has an update to a primary key.
- Insufficient Interested Transaction List entries in a block: this can happen (for instance) if PCTFREE is set very low.
- Bitmap indexes – where locks are applied to multiple rows during updates of index entries.
- Direct path inserts

There are also locks that Oracle uses for internal operations, such as allocating space or refreshing sequence cache entries. Examples are the High Watermark lock (HW), Sequence Cache lock (SQ) and the Space Transaction (ST) lock.

There's some more information about lock contention in the Toadworld article [Dealing with Oracle lock contention](#).

Latch and mutex contention

Oracle sessions share information in the buffer cache, shared pool and in other sections of the shared memory known as the System Global Area (SGA). It's essential that the integrity of SGA memory is maintained, so Oracle needs a way to prevent two sessions from trying to change the same piece of shared memory at the same time. Latches and mutexes serve this purpose.

Prior to Oracle 10g release 2, Oracle used latches for all shared memory synchronization. In 10g release 2, mutexes – a sort of a light-weight variation on the latch concept – replaced some latches.

The very nature of latches and mutexes creates the potential for contention. If one session is holding a latch which is required by another session then the sessions concerned are necessarily contending for the latch. Latch contention is indeed one of the most prevalent forms of Oracle contention.

Detecting latch contention

As with most contention scenarios, the wait interface and time model provide the best way to determine the extent of any contention that might exist. Time spent in latch or mutex “sleeps” will be recorded in V\$SYSTEM_EVENT and similar tables and will usually be the primary indication that a problem exists.

To break out mutex and latch waits and compare them to other high level wait categories, we could issue a query such as this:

```
SQL> WITH system_event AS
  2   (SELECT CASE WHEN (event LIKE '%latch%' or event
  3   LIKE '%mutex%' or event like 'cursor:%'))
  4   THEN event ELSE wait_class
  5   END wait_type, e.*
  6   FROM v$system_event e)
  7 SELECT wait_type,SUM(total_waits) total_waits,
  8   round(SUM(time_waited_micro)/1000000,2) time_waited_seconds,
  9   ROUND( SUM(time_waited_micro)
 10   * 100
 11   / SUM(SUM(time_waited_micro)) OVER (), 2) pct
 12 FROM (SELECT wait_type, event, total_waits, time_waited_micro
 13 FROM system_event e
 14 UNION
 15 SELECT 'CPU', stat_name, NULL, VALUE
 16 FROM v$sys_time_model
 17 WHERE stat_name IN ('background cpu time', 'DB CPU')) l
 18 WHERE wait_type <> 'Idle'
 19 GROUP BY wait_type
 20 ORDER BY 4 DESC
 21 /
```

WAIT_TYPE	TOTAL_WAITS	TIME_WAITED_SECONDS	PCT
CPU		1,494.63	69.26
latch: shared pool	1,066,478	426.20	19.75
latch free	93,672	115.66	5.36
wait list latch free	336	58.91	2.73
User I/O	9,380	27.28	1.26
latch: cache buffers chains	2,058	8.74	.40
Other	50	7.26	.34
System I/O	6,166	6.37	.30
cursor: pin S	235	3.05	.14
Concurrency	60	3.11	.14
library cache: mutex X	257,469	2.52	.12

Causes of latch contention

The most common scenarios leading to latch/mutex contention are:

- *library cache* latch or mutex waits when bind variables are not being used.
- *cache buffers chains* latch contention when there are certain very “hot” blocks.

Both these scenarios, and more information about latch contention can be found in the ToadWorld article [Resolving Latch Contention](#).

Shared memory contention

Oracle uses shared memory to improve performance by caching frequently accessed data in the buffer cache, reducing the amount of disk IO required to access that data. Oracle maintains other caches as well, such as the redo log buffer, which buffers IO to the redo log files.

The very sharing of memory creates the potential for contention, and requires that Oracle serialize – restrict concurrent access – to some areas of shared memory to prevent corruption. Oracle manages some of this serialization using the latching mechanisms discussed previously. However, contention for shared memory goes beyond latches, as sessions may have to wait for free buffers in memory when inserting new blocks, or for other sessions to finish processing blocks already in memory.

The most frequent causes of shared memory contention are:

- Free buffer waits, which occur when the Database writer cannot clear unmodified “dirty” blocks from the buffer cache fast enough. Sessions wanting to add blocks to the cache need to wait for the DBWR to catch up.
- Recovery writer waits, which occur for similar reasons but which involving the Flashback recovery writer and the flashback log files.
- Buffer busy waits, usually caused by multiple sessions trying to modify or access the same block of data at very high frequencies.

Free buffer waits and Recovery writer waits can often be a sign of an inadequate disk IO subsystem. Buffer busy waits might suggest a need to vary indexing strategies or employ Oracle partitioning. There’s some more information on Shared memory contention in the Toad World article [Buffer Cache Contention](#).

Strategy 8: Optimize memory to minimize IO

Now that we’ve tuned our SQL and eliminated contention, our *logical* IO demand will be realistic. Now is the time to prevent that logical IO from turning into physical IO.

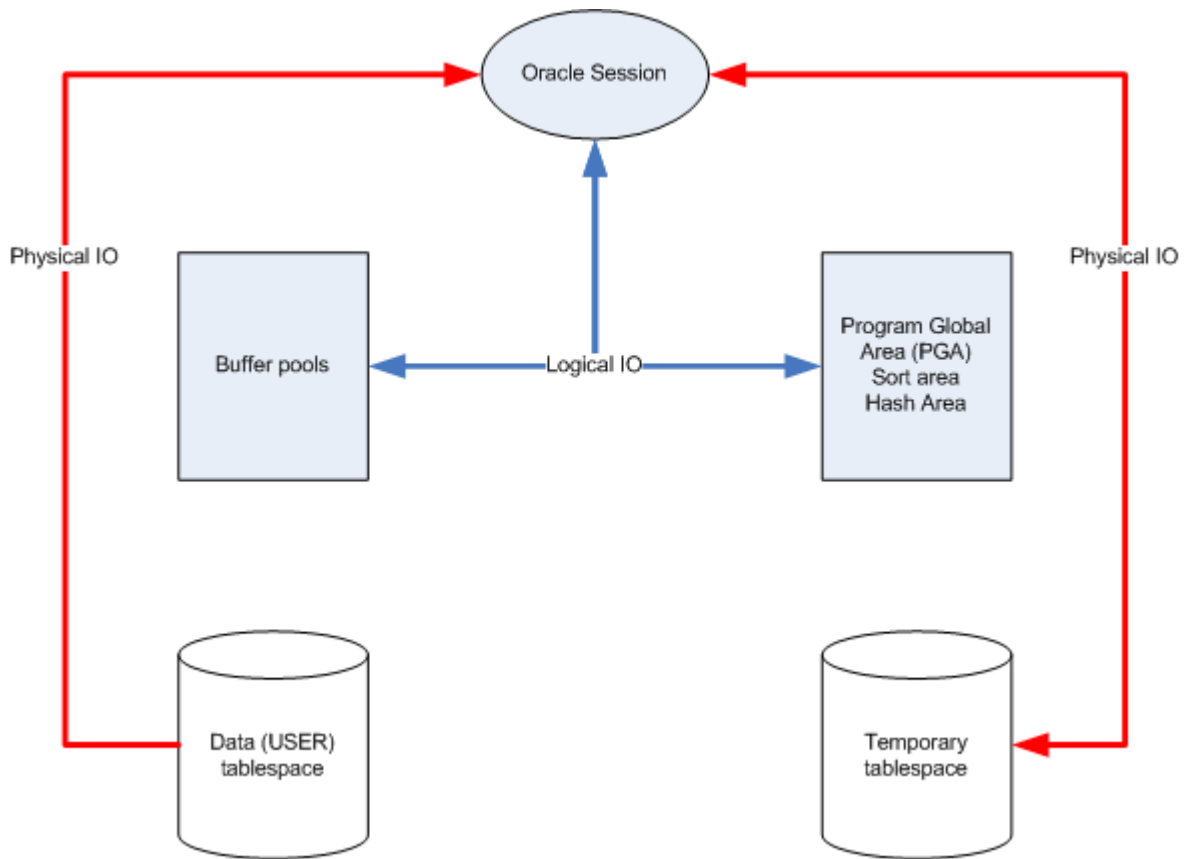


Figure 3. Oracle uses PGA to avoid temporary IO, and the buffer cache to avoid datafile IO

There are essentially two types of IO that we can avoid through optimizing memory. IO to the datafiles can be minimized through effective sizing of the buffer cache; while IO to the temporary tablespace can be minimized through optimal allocations to the PGA (see Figure 3).

Optimizing memory configurations *within* the SGA and *within* the PGA is critical, but arguably the most important memory configuration decision you will make is the distribution *between* the two areas. SGA – buffer cache – memory reduces the amount of physical IO that results from reading table and index blocks, while PGA memory reduces the amount of physical IO that results from sorting and hash operations. Minimizing total physical IO therefore requires that memory be distributed between the PGA and SGA correctly.

Examining the waits recorded for various types of IO – ‘db file’ waits and ‘direct path .. temp’ waits – provides some indication as to which type of IO is most prevalent and there where memory might most profitably be allocated. However, it is only the memory advisories – V\$PGA_TARGET_ADVICE and V\$SGA_TARGET_ADVICE – that can indicate how much IO would actually be avoided if memory configurations changed. You can use these advisories to determine whether an increase in one or the other memory areas is indicated.

In 11g you can use Oracle Automatic Memory Management (AMM) to move memory between the PGA and SGA dynamically based on workload demands. AMM is a significant advance and will often provide improved performance. However in some cases its decisions might be overly reactive or might conflict with business priorities (prioritizing sorts at the expense of index lookups for instance). Setting minimum values for key memory areas can often lead to a more optimal outcome.

Quest's Spotlight on Oracle – part of the Toad DBA suite – contains a powerful memory management capability that can help you determine an optimum memory configuration. Figure 4 shows Spotlight's memory manager.

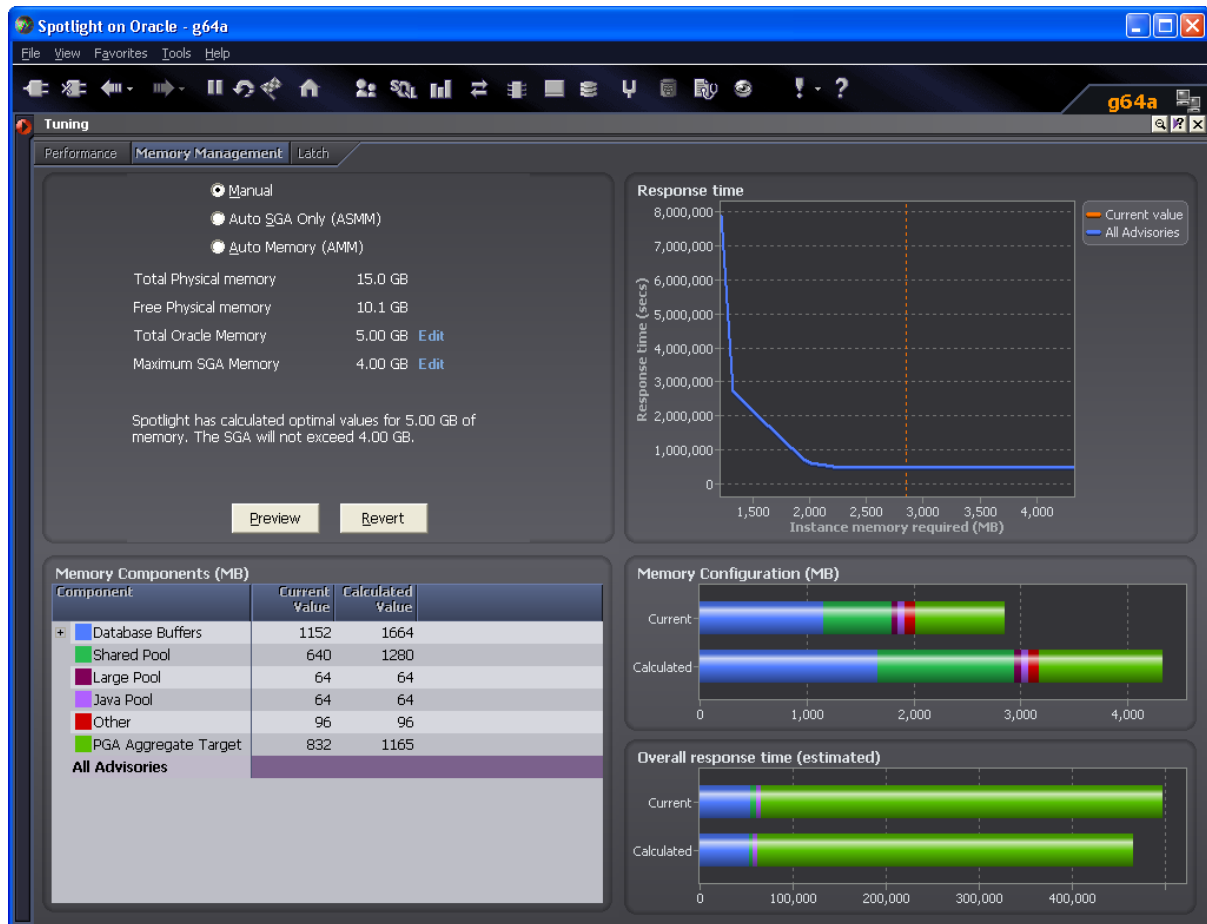


Figure 4. Spotlight on Oracle memory management

Oracle 11g introduces the result set cache, which allows complete result sets to be stored in memory. If a result set can be re-used then almost all the overhead of SQL execution can be avoided. The result set cache best suits small result sets from expensive queries on tables that are infrequently updated. Applying the result set cache to all SQLs or to all SQLs for a specific table is unlikely to be effective and can lead to significant latch contention.

Strategy 9: Optimizing IO

Most of the techniques we've looked at in preceding sections have been aimed at avoiding or minimizing disk IO. Tuning our SQL and PL/SQL reduces the workload demand – largely logical IO – on our database. Minimizing contention attacks the bottlenecks that might be preventing that workload demand from being achieved. Optimizing memory reduces the amount of workload that translates into disk activity. If you've applied the practices in the previous sections then your physical disk demand has been minimized: now it's time to optimize the disk sub-system to meet that demand.

There's quite a variety of Oracle IO activities, each of which associated with different characteristics and optimization strategies. We don't have space in this article to discuss all these in detail, though Figure 5 summarizes the IO types and associated wait events.

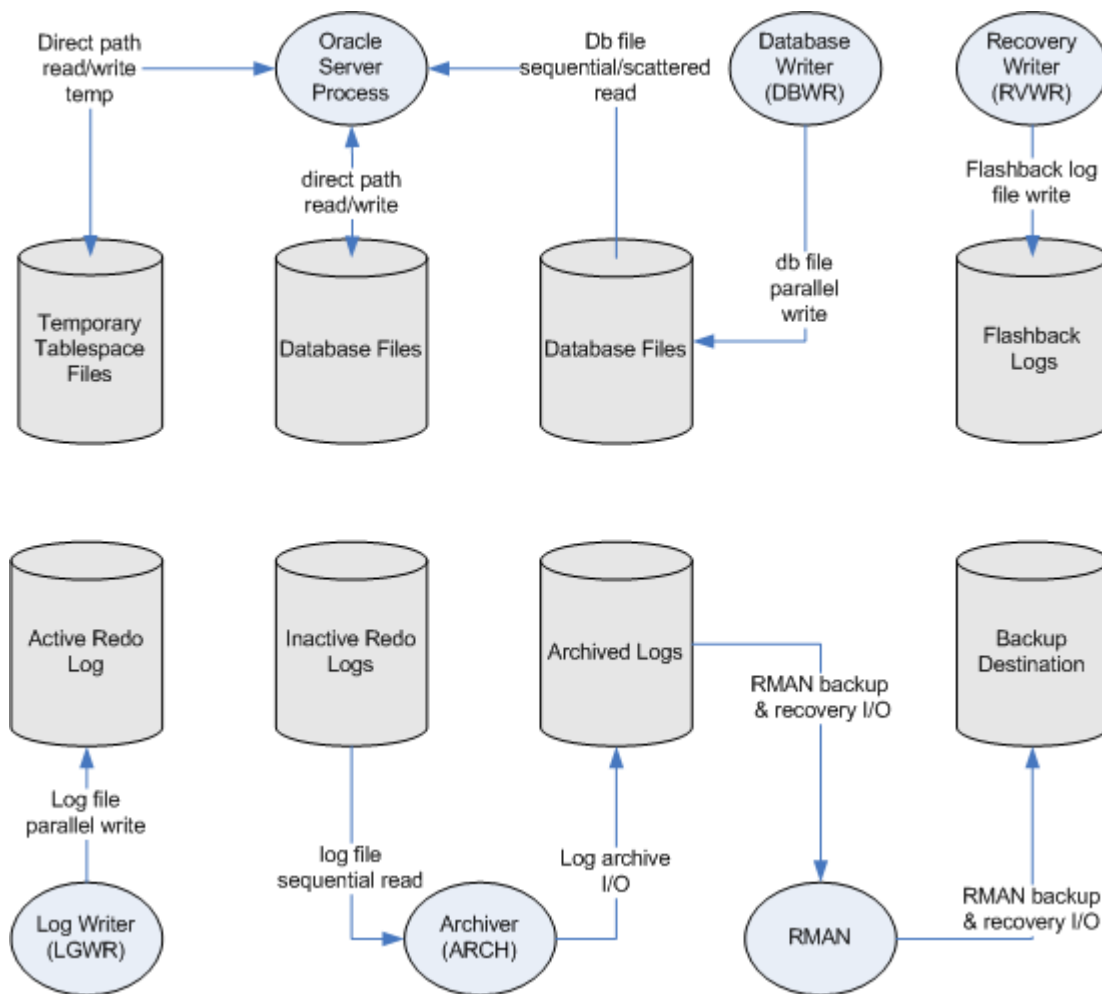


Figure 5. Oracle IO and related wait events

Here are some basic guidelines for optimizing Oracle IO:

- The delay for an individual IO is referred to as *latency* or *service time* and is typically measured in milliseconds. The amount of IO that can be done in a unit of time is referred to as throughput and is usually expressed in IOs per second (IOPS).
- Disk devices will provide the lower latency when they are only partially filled with data and when they are producing only a fraction of maximum possible throughput. Keeping disks less than 50% full and at less than 50-75% of maximum throughput is a possible rule of thumb when optimizing service time.
- Throughput is generally achieved by using multiple disk drives and striping data across the devices. Throughput goals can only be achieved if you acquire enough disks to meet the aggregate IO demand.
- The two most popular ways of spreading data across Oracle datafiles are RAID5 and striping (RAID0, RAID10, RAID 0+1). RAID5 imposes a very heavy penalty on write performance and

is not recommended even for primarily read only databases unless there is no temporary segment IO. Striping is the technique of choice on performance grounds.

- Since temporary segment IO and permanent segment IO have such different IO characteristics and diverse service level expectations, it can often be a good idea to separate temporary tablespace datafiles on their own disk volumes.
- For redo and archive logs, RAID5 is even more undesirable and should generally not be used unless performance is not important. Redo logs do not always benefit from striping in any case: alternating redo logs across two devices and placing the archive destination on a striped volume is often the high performance solution.
- Flashback logs can be stored together with archive logs on a fine grained striped device, although better performance will often be obtained by allocating the flashback recovery area its own dedicated disk volume.

Conclusion

Contention, memory optimization and IO are three massive topics, so I've only been able to provide a very brief overview in this article. All these topics are outlined in much more detail within my book [Oracle Performance Survival Guide](#). There's also plenty of resources at [ToadWorld](#): check out in particular my sections on [contention](#) and [SQL Tuning](#).

In the next – and final – instalment, we will wrap up with an in-depth look at how Oracle RAC can be used to maximize performance, and how to get the most out of an Oracle RAC deployment.

Guy Harrison is a Director of Research and Development at Quest Software, is an Oracle ACE and has over 20 years experience in application and database administration, performance tuning and software development. Guy is the author of [Oracle Performance Survival Guide](#) (Prentice Hall, 2009) and [MySQL Stored Procedure Programming](#) (O'Reilly with Steven Feuerstein) as well as other books, articles and presentations on database technology. Guy is the architect of Quest's Spotlight® family of diagnostic products and has contributed to the development of other Quest products, such as Toad®. Guy can be found on the Internet at www.guyharrison.net, on email at guy.harrison@quest.com and is [@guyharrison](#) on twitter.

