Oracle performance tuning: a systematic approach

A mission critical application system is experiencing unsatisfactory performance. As an experienced Oracle performance specialist, you are called in to diagnose the problem. You're well versed in modern wait-based performance profiling oriented performance diagnostics (such as "YAPP"[1]), so the first thing you want to determine is which wait category is consuming the bulk of non-idle time. Looking at `V$SYSTEM_EVENT`, you immediately see that the database is spending the vast majority of it's time within '`db file sequential read`' events. Furthermore, the average time for each of these events – which represent single block reads against database files – is more than 20ms which is far higher than the service time you expect from the expensive and sophisticated disk array supporting the application.

You suspect that the disk array might have insufficient bandwidth to support the applications demands. Considering the average physical IO rate of 8,000 IOs per second, you determine that this corresponds to a rate of more than 100 IOs per second for each disk in the array and you also note that the disk devices are reporting that they are almost continuously 100% busy. You therefore conclude that the system is IO bound and that the solution is to increase IO bandwidth. You recommend increasing the number of disk devices in the array by a factor of four. The dollar cost is substantial as is the downtime required to redistribute data across the new disks within the array. Nevertheless, something has to be done, so management approve the expense and the downtime. Following the implementation, users report they are satisfied with performance and you modestly take all the credit.

A successful outcome? You think so, until….

- Within a few months performance is again problematic and disk IO is again the culprit.
- Another Oracle performance expert is called into the case and she reports that a single indexing change would have fixed the original problem with no dollar cost and no down time.
- The new index is implemented, following which the IO rate is reduced to one tenth of that observed during your original engagement. Management prepare to sell the now-surplus disk devices on E-Bay and mark your consulting record with a "do not re-engage" stamp.
- Your significant other leaves you, and you end up shaving your head and becoming a monk.

After years of silent mediation, you realize that while methodologies such as YAPP correctly focus your attention on the most time consuming activities performed by the database, they fail to differentiate between *causes* and *effects*. Consequently, you mistakenly dealt with an effect – the high disk IO rate – while neglecting the cause (a missing index).

A brief history of Oracle tuning philosophy and practice

In the early nineties, the discipline of tuning an Oracle server was nowhere near as well established as today. In fact, performance tuning was mostly limited to a couple of well known "rules of thumb".

---

[1] www.oraperf.com

The most notorious of these guidelines was that you should tune the “`buffer cache hit ratio`”: the ratio which describes the proportion of blocks of data found in memory when requested by an SQL. Increasing the buffer cache until the ratio reached 90-95% was often suggested. Similar target values were suggested for other ratios such as the rowcache hit ratio or the latch hit ratio.

The problem with these “ratio-based” techniques was that while the ratios usually reflected some measure of internal Oracle efficiency, they were often only loosely associated with the performance experienced by an application using the database. For instance, while it is obviously better for a block of data to be found in memory, high hit ratios will often reflect very inefficient repetitive accesses to the same block of memory and be associated with CPU bottlenecks. Furthermore, reducing disk IO might be fine, but if the application is spending 90% of it’s time waiting on locks that effort will ultimately be futile.

The emergence of “wait” information in Oracle version 7.1 provided an alternate method of approaching tuning. This wait information included the amount of time Oracle sessions spent waiting for resources (lock, IO, etc) to become available. By concentrating on the wait events that were consuming the most wait time, we were able to target our tuning efforts most effectively.

Pioneers of systematic Oracle performance tuning such as Cary Millsap promoted this technique vigorously. Anjo Kolk, with his “Yet Another Performance Profiling” (YAPP) methodology is probably the most well known advocate of this technique.

Wait based tuning took a surprisingly long time to become mainstream: 5-10 years passed between the original release of the wait information and widespread acceptance of the technique. However, today almost all Oracle professionals are familiar with the wait-based tuning.

Moving beyond a symptomatic approach

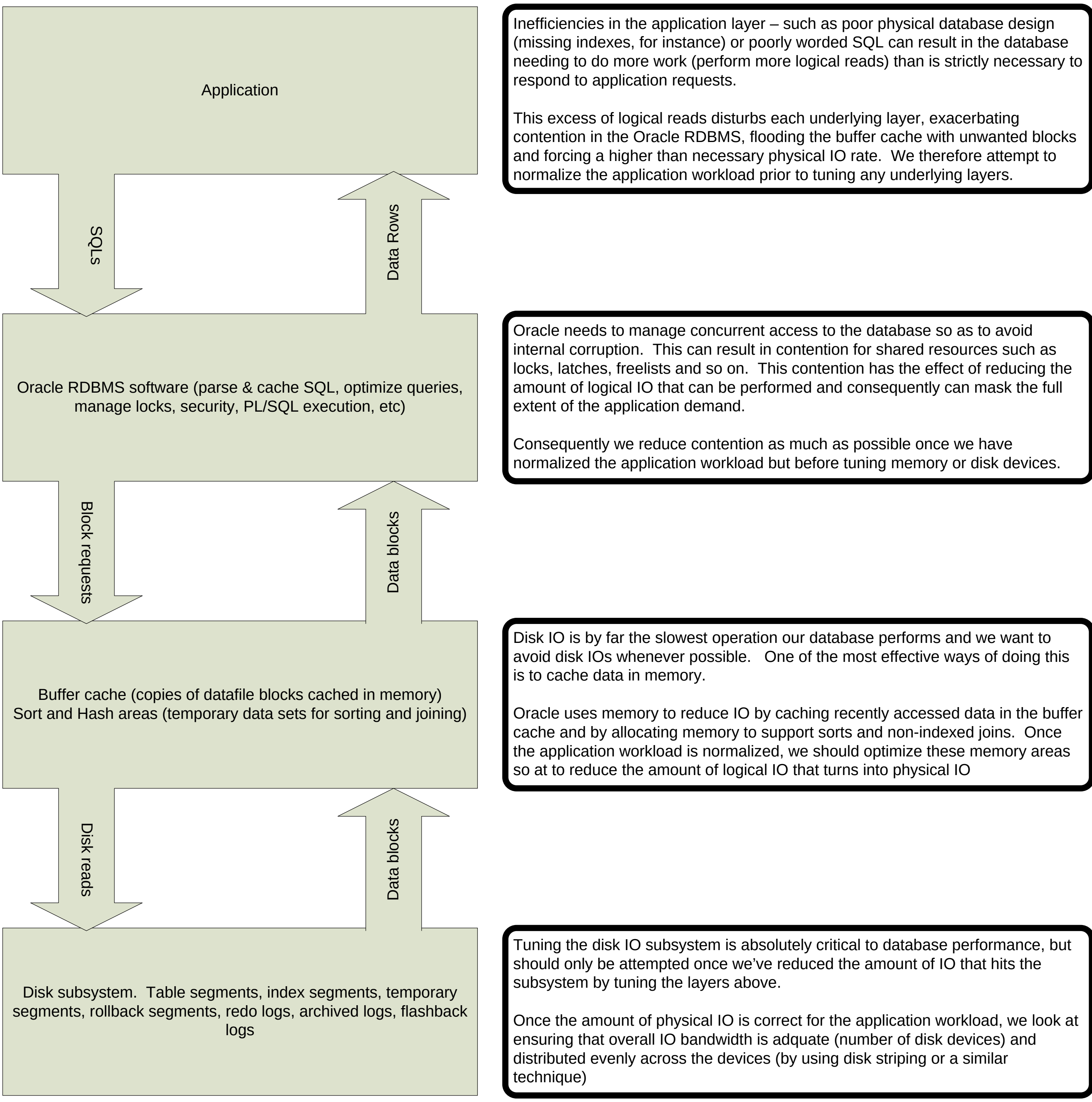
The shift from the “ratio-based” to “wait-based” tuning has resulted in a radical improvement in our abilities to diagnose and tune Oracle-based applications. However, as we noted earlier, simplistically focusing on the largest component of response time can have several undesirable consequences:

- We may treat the symptoms, rather than the causes of poor performance.
- We may be tempted to seek hardware-based solutions when configuration or application changes would be more cost effective.
- We might deal with today’s pain, but fail to achieve a permanent or scalable solution.

To avoid the pitfalls of a an overly simplistic wait-based analysis, we need to approach our tuning activities in a number of well defined stages. These stages are dictated by the reality of how applications, databases and operating systems interact:

1. Applications send requests to the database in the form of SQL statements (including PL/SQL requests). The database responds to these requests with return code and/or result sets.
2. To deal with an application request, the database must parse the SQL, perform various overhead operations (security, scheduling, isolation level management) before finally executing the SQL. These operations use operating system resources (CPU & memory) and may be subject to contention between multiple database sessions.

3. Eventually, the database request will need to access some number of database blocks. The exact number of blocks can vary depending on the database design (indexing for instance) and application (wording of the SQL for instance).
4. Some of these blocks required will be in memory. The chance that a block will be in memory will be determined mainly by the frequency with which the block is requested and the amount of memory available to cache such blocks.
5. If the block is not in memory it must be accessed from disk, resulting in real physical IO. Physical IO is by far the most expensive of all operations so needs to be minimized at all costs.



Inefficiencies in the application layer – such as poor physical database design (missing indexes, for instance) or poorly worded SQL can result in the database needing to do more work (perform more logical reads) than is strictly necessary to respond to application requests.

This excess of logical reads disturbs each underlying layer, exacerbating contention in the Oracle RDBMS, flooding the buffer cache with unwanted blocks and forcing a higher than necessary physical IO rate. We therefore attempt to normalize the application workload prior to tuning any underlying layers.

Oracle needs to manage concurrent access to the database so as to avoid internal corruption. This can result in contention for shared resources such as locks, latches, freelists and so on. This contention has the effect of reducing the amount of logical IO that can be performed and consequently can mask the full extent of the application demand.

Consequently we reduce contention as much as possible once we have normalized the application workload but before tuning memory or disk devices.

Disk IO is by far the slowest operation our database performs and we want to avoid disk IOs whenever possible. One of the most effective ways of doing this is to cache data in memory.

Oracle uses memory to reduce IO by caching recently accessed data in the buffer cache and by allocating memory to support sorts and non-indexed joins. Once the application workload is normalized, we should optimize these memory areas so at to reduce the amount of logical IO that turns into physical IO

Tuning the disk IO subsystem is absolutely critical to database performance, but should only be attempted once we've reduced the amount of IO that hits the subsystem by tuning the layers above.

Once the amount of physical IO is correct for the application workload, we look at ensuring that overall IO bandwidth is adquate (number of disk devices) and distributed evenly across the devices (by using disk striping or a similar technique)

Figure 1 Overview of tuning by layers[2]

---

[2] As far as I know, the general concept of "tuning by Layers" was first proposed by Steve Adams (http://www.ixora.com.au/tips/layers.zip).

The way in which these layers – application-Oracle code-Oracle cache-IO subsystem – interact suggest that problems detected in one layer might be caused or cured by configuration in the higher layer. In Figure 1, I summarize the interactions between layers and provide an overview of the optimizations appropriate at each layer. I'll elaborate on the steps we undertake at each level throughout the remainder of this article.

Using the time model

The wait interface – combined with the new "time model" table in Oracle Database 10g – is still your best friend when identifying tuning opportunities at a high level.

Figure 2 shows a query that returns data from both tables. I've eliminated "idle" waits and background process times in this query. We can use this query throughout our tuning efforts to identify major opportunities within each layer. For instance, in Figure 2 we see that SQL execution time dominates PL/SQL execution time and therefore that our priority when tuning the application layer will be to address SQL rather than PL/SQL.

```
SQL> l
  1  SELECT   wait_class, event, time_waited / 100 time_secs
  2      FROM v$system_event e
  3     WHERE e.wait_class <> 'Idle' AND time_waited > 0
  4  UNION
  5  SELECT   'Time Model', stat_name NAME,
  6           ROUND ((VALUE / 1000000), 2) time_secs
  7      FROM v$sys_time_model
  8     WHERE stat_name NOT IN ('background elapsed time', 'background cpu time')
  9* ORDER BY 3 DESC
SQL> /

WAIT_CLASS           EVENT                                    TIME_SECS
-------------------- ---------------------------------------- ---------
Time Model           DB time                                     369.75
Time Model           sql execute elapsed time                    316.55
System I/O           control file sequential read                157.03
Time Model           DB CPU                                      149.22
User I/O             db file sequential read                      74.30
Time Model           PL/SQL execution elapsed time                56.89
Time Model           parse time elapsed                           54.68
Time Model           hard parse elapsed time                      51.72
Time Model           inbound PL/SQL rpc elapsed time              42.14
Commit               log file sync                                29.12
System I/O           control file parallel write                  25.33
User I/O             db file scattered read                       20.06
```

Figure 2 Using the Time model and wait interface tables

Stage 1: Normalizing the application workload

Our first objective is to normalize the applications demand on the database. We aim to ensure that the demand is appropriate for the activities that the application is attempting. For instance, while relatively infrequent scans of large tables might be appropriate for weekly reconciliation reports, SQLs that execute many times per second should be supported by efficient indexed access paths.

The most useful single indicator of application demand is the logical IO rate, which represents the number of block reads required to satisfy application requests.

Broadly speaking, there are two main techniques by which we reduce application workload:

- Tuning the application code. This might involve changing application code so that it issues fewer requests to the database (by using a client side cache for instance). However, more often this will involve re-writing application SQL and/or PL/SQL.
- Modifying the physical implementation of the applications schema. This will typically involve indexing, de-normalization or partitioning.

I'm not going to attempt to provide an overview of SQL tuning and database physical design optimization in this short article. However, before you dismiss the option of re-writing SQL because of a "fixed" application code base (SAP for instance), make sure you are familiar with the Stored Outline facility which effectively allows you rewrite your SQL at the database layer without having to amend the application source.

Identifying SQL tuning opportunities

For many years, DBAs have used scripts based on queries against `V$SQL` to identify the "top" SQL. However, while SQL statements which consume the most logical IO are often good targets for tuning, it's often only examination of individual steps that will pinpoint the best tuning opportunity. In Oracle Database 10g, we can use cached query plan statistics to pinpoint individual steps within an SQL execution that might warrant attention. The view `V$SQL_PLAN` shows the execution plan for all cached SQL statements, while `V$SQL_PLAN_STATISTICS` shows execution counts, IO and rows processed by each step in the plan[3]. Figure 3 shows the essential columns in these new tables.

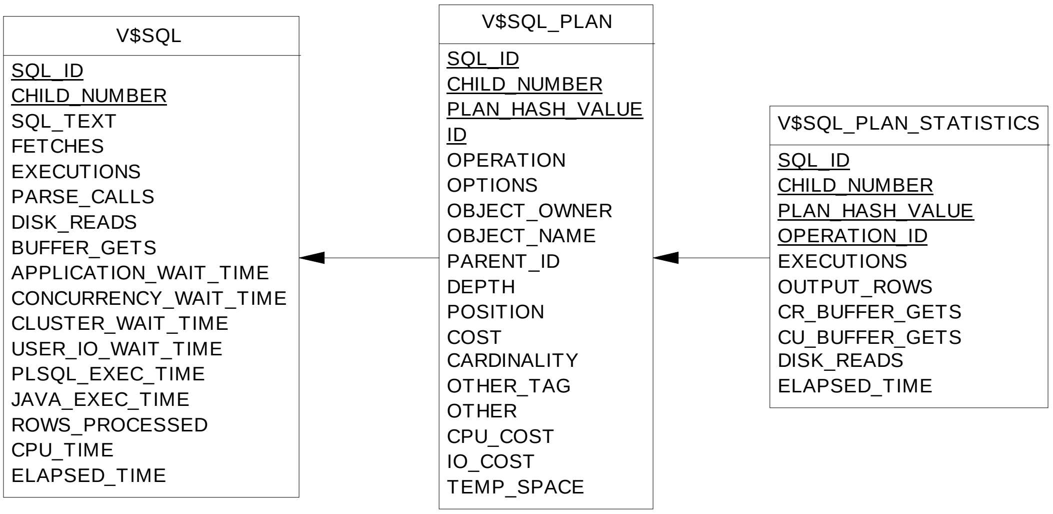


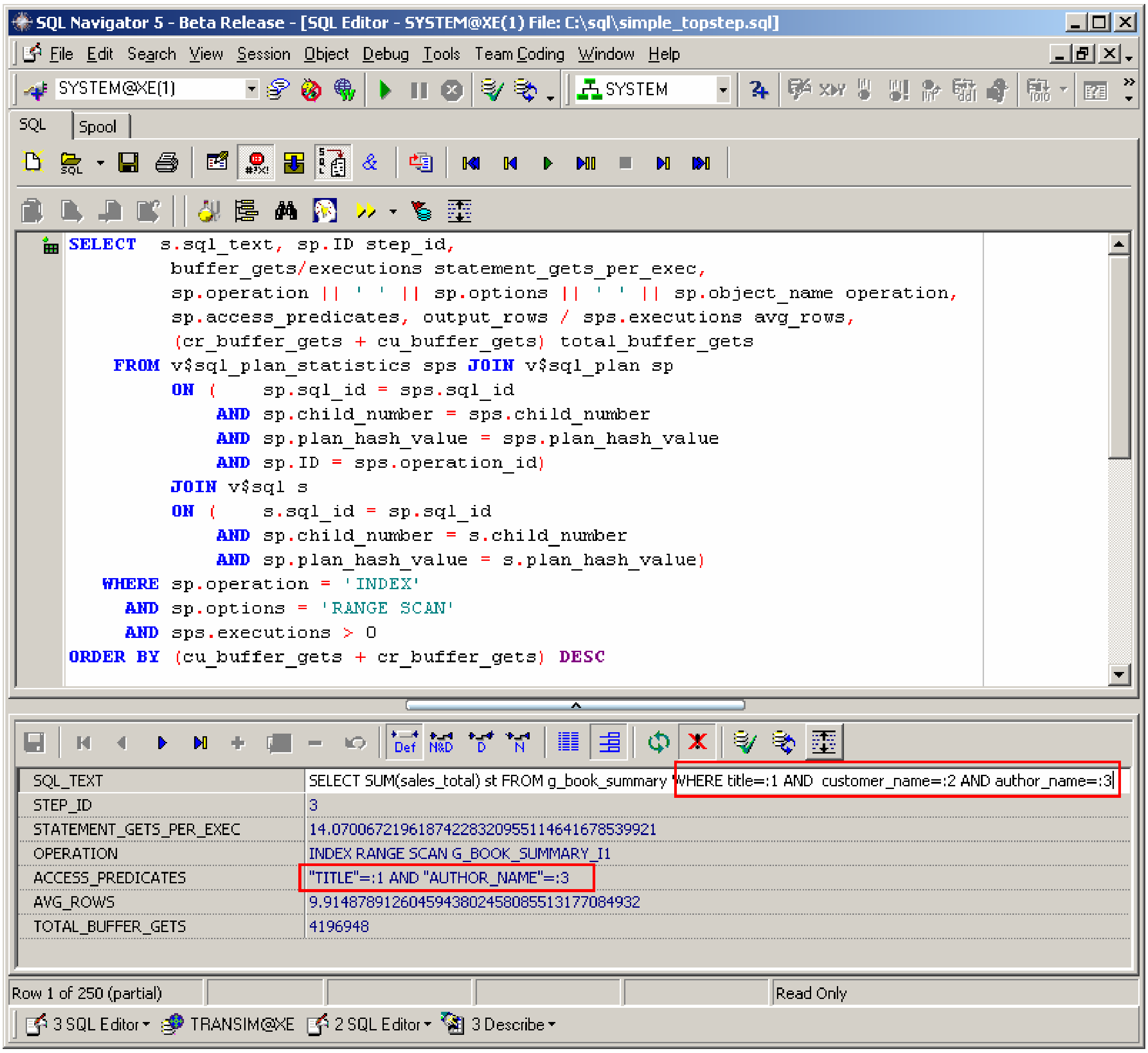
Figure 3 Essential SQL tuning view information

Using these tables allows us to more accurately identify SQL that might be improved by tuning. In Figure 4, we search for expensive index scans and also show the SQL clauses ("access predicates") responsible. This allows us to find indexes that don't include all the columns in the `WHERE` clause. In

---

[3] You may have to up your `STATISTICS_LEVEL` from `TYPICAL` to `ALL` to get some of this new information.

this case a frequently executed SQL is using an index with only 2 out of 3 where clause conditions. Creating an index with all three columns reduced the logical IO demand for the step from 14 logical I/Os to only 4 and resulted in a substantial reduction is database load.



```
SELECT  s.sql_text, sp.ID step_id,
         buffer_gets/executions statement_gets_per_exec,
         sp.operation || ' ' || sp.options || ' ' || sp.object_name operation,
         sp.access_predicates, output_rows / sps.executions avg_rows,
         (cr_buffer_gets + cu_buffer_gets) total_buffer_gets
    FROM v$sql_plan_statistics sps JOIN v$sql_plan sp
         ON (    sp.sql_id = sps.sql_id
             AND sp.child_number = sps.child_number
             AND sp.plan_hash_value = sps.plan_hash_value
             AND sp.ID = sps.operation_id)
         JOIN v$sql s
         ON (    s.sql_id = sp.sql_id
             AND sp.child_number = s.child_number
             AND sp.plan_hash_value = s.plan_hash_value)
   WHERE sp.operation = 'INDEX'
     AND sp.options = 'RANGE SCAN'
     AND sps.executions > 0
ORDER BY (cu_buffer_gets + cr_buffer_gets) DESC
```

```
SQL_TEXT                 SELECT SUM(sales_total) st FROM g_book_summary WHERE title=:1 AND  customer_name=:2 AND author_name=:3
STEP_ID                  3
STATEMENT_GETS_PER_EXEC  14.0700672196187422832095511464167853 9921
OPERATION                INDEX RANGE SCAN G_BOOK_SUMMARY_I1
ACCESS_PREDICATES        "TITLE"=:1 AND "AUTHOR_NAME"=:3
AVG_ROWS                 9.914878912604594380245808551317708 4932
TOTAL_BUFFER_GETS        4196948
```

Figure 4 Identifying expensive index range scans

Application workload: Other things to check

The logical read load generated by application SQL is undoubtedly the major factor governing the application’s demand on the database. However, there are measures beyond SQL tuning that may further reduce application demand. These include:

- SQL parse time. Parsing should be a very small part of overall demand, providing that bind variables, rather than literals, are used in application SQL. If parse activity appears to be excessive (as shown by the “parse time elapsed” category in our time model query) then you can try the “silver bullet” solutions offered by the `CURSOR_SHARING` and `SESSION_CACHED_CURSORS` parameters.

- Indexes contribute to the overhead of DML operations – especially `INSERT` and `DELETE` statements. You should drop any unused indexes, which you can identify by exploiting the `MONITORING USAGE` clause of `ALTER/CREATE INDEX`).
- The overhead of table scans can be reduced by optimizing table storage (`PCTFREE`/`PCTUSED`). You should also consider partitioning or relocating long infrequently used columns for tables that are subject to expensive scans that cannot be optimized by indexing. Also consider the `COMPRESS` option which increases CPU utilization slightly but can significantly reduce I/O - especially for table scans.
- Tune PL/SQL: in particular use array processing in your stored procedures. You can use the time model category “`PL/SQL execution elapsed time`” to determine if PL/SQL execution time is significant.

Stage 2: Reducing contention and bottlenecks

Once we’ve adjusted the application workload demand to a sensible minimum, we are ready to tackle contention within the Oracle server. Application demand manifests mainly as logical IO requests which in turn result in some amount of physical IO. However, contention prevents the application demand from being fully realized resulting in an underestimation of application load (and of course, poor performance for the application). We should eliminate as much of this contention as possible before optimizing IO.

The two most prevalent forms of contention observed in Oracle-based applications are:

(a) Contention for rows within tables (locks) and
(b) Contention for areas of shared memory (latches, buffer busy, free buffer,etc)

Lock contention – which exhibits as waits for events which include the ‘enq:’ prefix[4] - is largely a factor of application design: Oracle’s locking model allows for high concurrency since readers never wait for locks, writers never wait for readers and locks are applied at the row level only. Typically, lock contention is caused by an application design that involves very high simultaneous updates against a single row or in which locks are held for an excessive length of time, perhaps due to an overly pessimistic locking model. This sort of contention is almost impossible to eliminate without application logic changes.

Contention for shared memory occurs when sessions wish to read or write to shared memory in the SGA concurrently. All shared memory is protected by latches – which are similar to locks except that they prevent concurrent access to data in shared memory rather than data in tables. If a session needs to modify some data in memory it will acquire the relevant latch and if another session wants to read or modify the same data, then a latch wait may occur.

---

[4] Prior to Oracle 10g, all lock waits where summarized in the ‘enqueue’ event.

```
SQL> l
  1  SELECT   wait_class, event, time_waited / 100 time_secs
  2      FROM v$system_event e
  3     WHERE e.wait_class <> 'Idle' AND time_waited > 0
  4  UNION
  5  SELECT   'Time Model', stat_name NAME,
  6           ROUND ((VALUE / 1000000), 2) time_secs
  7      FROM v$sys_time_model
  8     WHERE stat_name NOT IN ('background elapsed time', 'background cpu time')
  9* ORDER BY 3 DESC
SQL> /

WAIT_CLASS           EVENT                                    TIME_SECS
-------------------- ---------------------------------------- ---------
Time Model           DB time                                    4139.88
Time Model           sql execute elapsed time                   4138.40
Time Model           PL/SQL execution elapsed time              1392.42
Application          enq: TX - row lock contention              1336.25
Time Model           parse time elapsed                          962.57
Time Model           DB CPU                                      908.20
Concurrency          library cache pin                           399.32
Concurrency          latch: library cache                        204.22
Concurrency          latch: library cache lock                    59.26
User I/O             db file sequential read                      54.98
System I/O           control file sequential read                 33.52
Time Model           hard parse elapsed time                      31.52
Concurrency          library cache lock                           23.38
Concurrency          latch: library cache pin                     22.33
Time Model           inbound PL/SQL rpc elapsed time              21.18
Concurrency          cursor: mutex S                              14.03
User I/O             db file scattered read                       11.09
Other                rdbms ipc reply                               9.17
```

Figure 5 Evidence of contention for latches and locks

In modern versions of Oracle, the most intractable latch contention is for the '`buffer cache chains`' latch that protect areas of the buffer cache. Some degree of latch contention may be inevitable, but there are things you can do to reduce even apparently intractable latch contention. In particular:

- Often latch contention occurs because of "hot" blocks and often these blocks are index root or branch blocks. Partitioning the table and associated indexes can often reduce the contention by spreading the demand across multiple partitions.
- The practice of adjusting the latch "spin count' was a frequent pastime in earlier versions of Oracle, but is now actively discouraged by Oracle. However, we've done research that suggests that adjusting spin count can be an effective measure when all else fails – see http://www.quest-pipelines.com/newsletter-v5/Resolving_Oracle_Latch_Contention.pdf .

Other contention points

There's a long list of other possible contention points, but here are some we see a lot:

- ‘`buffer busy`’ waits sometimes occur because of a ‘hot’ block, but probably more often because of tables that only have one freelist and are subject to concurrent insert. Modern Oracle databases (using ASSM tablespaces) should not have tables with single freelists, but if you’ve migrated an older database through multiple versions of Oracle, there may be legacy tables with freelist problems.
- Unindexed foreign keys can cause lock contention by causing table level share locks on the child table when the parent is updated.
- Sequences should be created with a `CACHE` size adequate to ensure that the cache is not frequently exhausted and should not normally use the `ORDER` clause.
- SQL statements that don’t use bind variables cause latch contention ‘`library cache latch`’ as well as high parse times. The `CURSOR_SHARING` parameter can help.

Step 3: Reducing physical IO

Now that the application demand is nominal, and contention that might otherwise mask that demand eliminated, we turn our attention to reducing time spent waiting for IO. However, before we turn our attention to the disks themselves, we concentrate on preventing as much physical IO as possible. We do this by configuring memory to cache and buffer IO requests.

Most physical IO in an Oracle application occurs either because:

(a) An application session requests data to satisfy a query or DML request or
(b) An application session must sort data or create a temporary segment in order to support a large join, `ORDER BY` or similar operation.

Memory allocated to the Oracle buffer cache stores copies of database blocks in memory and thereby eliminates the need to perform physical IO if a requested block is in that memory. Oracle DBAs traditionally would tune the size of the buffer cache by examining the “`buffer cache hit ratio`” – the percentage of IO requests that were satisfied in memory. However this approach has proved to be error prone, especially when performed prior to tuning the application workload or eliminating contention.

In modern Oracle, the effect of adjusting the size of the buffer cache can be accurately determined by taking advantage of the Oracle advisories. `V$DB_CACHE_ADVICE` shows the amount of physical I/O that would be incurred or avoided had the buffer cache been of a different size. Examining this advisory will reveal whether increasing the buffer cache will help avoid IO, or if reducing the buffer cache could free up memory without adversely affecting IO.

Oracle allows you to setup separate memory areas to cache blocks of different size and also allows you to nominate `KEEP` or `RECYCLE` areas to cache blocks from full table scans. You can optimize your IO by placing small tables accessed by frequent table scans in `KEEP`, and large tables subject to infrequent table scans only in `RECYCLE`. `V$DB_CACHE_ADVICE` will allow you to appropriately size each area, although this resizing will occur automatically in Oracle database 10g.

The buffer cache exists in the System Global Area (SGA) which also houses other important shared memory areas such as the shared pool, java pool and large pool. Oracle database 10g automatically sizes these areas within the constraint of the `SGA_MAX_SIZE` parameter.

In addition to disk reads to access data not in the buffer cache, Oracle may perform substantial IO when required to sort data or execute a hash join. Where possible, Oracle will perform a sort or hash join in memory using memory configured within the Program Global Area (PGA). However, if sufficient memory is not available, then Oracle will write to temporary segments in the "temporary" tablespace.

The amount of memory available for sorts and hash joins is determined primarily by the `PGA_AGGREGATE_TARGET` parameter. The `V$PGA_TARGET_ADVICE` advisory view will show how increasing or decreasing `PGA_AGGREGATE_TARGET` will affect this temporary table IO.

Oracle database 10g now manages the internal memory within the PGA and SGA quite effectively. But Oracle does not move memory between the PGA and SGA so it's up the DBA to make sure that memory is allocated effectively to these two areas. Unfortunately, the two advisories concerned do not measure IO in the same units – `V$DB_CACHE_ADVICE` uses IO counts, while `V$PGA_TARGET_ADVICE` uses bytes of IO. Consequently it is hard to work out if overall IO would be reduced if one of these areas were to be increased at the expense of another[5]. Also, it's difficult to associate the IO savings reported by the advisories with IO time as reported in the wait interface. Nevertheless, determining an appropriate trade-off between the PGA and SGA sizing is probably the most significant memory configuration decision facing today's DBA and can have a substantial impact on the amount of physical IO that the database must perform.

Stage 4: Optimizing disk IO

At this point, we've normalized the application workload – in particular the amount of logical IO demanded by the application. We've eliminated contention that might be blocking – and therefore masking - those logical IO requests. Finally, we've configured available memory to minimize the amount of logical IO that ends up causing physical IO. Now – and only now – it makes sense to make sure that our disk IO subsystem is up to the challenge.

To be sure, optimizing disk IO subsystems can be a complex and specialized task; but the basic principles are straightforward:

1. Ensure the IO subsystem has enough bandwidth to cope with the physical IO demand. This is primarily determined by the number of distinct disk devices you have allocated. Disks vary in performance, but the average disk device might be able to perform about 100 random IOs per second before becoming saturated. For most databases, this will mean acquiring much more disk than simple storage requirements dictate – you need to acquire enough disks to sustain your IO rate as well as enough disks to store all your data.

---

[5] See http://www.nocoug.org/download/2004-11/Optimising_Oracle9i_Instance_Memory3.pdf. Note also that Quest's Spotlight on Oracle can calculate the optimal sizes of the PGA and SGA.

2. Spread your load evenly across the disks you have allocated – the best way to do this is RAID 0 (Striping). The worst way – for most databases – is RAID 5 which incurs a 400% penalty on write IO.

The obvious symptom of an overly-stressed IO subsystem is excessive delays responding to IO requests. The expected delay – called *service time* – varies from disk to disk, but even on the slowest disks should not exceed about 15ms. Most production-quality SCSI disks should have a service time under about 10ms while disks inside a storage array boasting a large non-volatile cache might have service times under 5ms. Therefore, you need to understand your IO subsystems characteristics to correctly diagnose a disk IO bottleneck, since service times can vary so much.

Spreading the load across spindles is best done by hardware or software striping. Oracle's ASM technology provides a simple and always available method (for 10g at least) of doing this for ordinary disk devices. Alternating datafiles across multiple disks is usually less effective, though still better than no striping at all.

For most databases, optimizing the datafiles for read activity makes the most sense, because Oracle sessions do not normally wait for datafile writes - the database writer process (DBWR) writes to disk asynchronously. However if the DBWR cannot keep up with database activity, then the buffer cache will fill up with "dirty" blocks and sessions will experience "`free buffer waits`" or "`write complete waits`" as the DBWR struggles to write out the modified blocks. For this reason it's important to optimize the DBWR so that it can effectively write out modified blocks to the database files across all the disk devices simultaneously. On most systems, this means ensuring that asynchronous IO is enabled. If asynchronous IO is not available, configure multiple DBWRs.

Redo log IO activity has its own pattern with sequential writes to the on-line log generated by the log writer process (LGWR) coupled with reads of the off-line logs by the archiver process (ARCH). If the ARCH cannot keep up with the LGWR then "log switch" waits will occur while the LGWR waits for the ARCH to keep up. Either configure logs and archive destinations on wide fine-grained striped devices, or allocate logs on alternating disks (so that ARCH is reading from one disk while LGWR is writing to the other).

Isn't it all about reducing I/O?

For most databases, the ultimate aim is to reduce disk IO. Disk IO remains the most expensive operation faced by the database.

When faced with an obviously IO-bound database, it's tempting to deal with the most obvious (symptom) – the IO subsystem – immediately. Unfortunately, this usually results in treating the symptom rather than the cause, is often expensive and usually ultimately futile. Methodically tuning the database layer by layer almost always leads to a healthier database, happier users and a more appreciated DBA.