Understanding Shared Pool Memory Structures

Tips on how to optimize usage and avoid errors

An Oracle White Paper
September, 2005

ORACLE

NOTE:

This document is for informational purposes. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

Table of Contents

INTRODUCTION

The Oracle shared pool provides critical services for sharing of complex objects among large numbers of users. Historically, the structures and mechanisms of the shared pool have not been well understood, making it unintuitive to tune. Prior to 10g R1, DBAs often spent much time learning about the shared pool memory management to configure and tune shared pool usage; with the tight time constraints and ever-changing workloads, many found this task daunting.

ADDM automatically detects and reports the specific tuning issues encountered. It also provides a time impact for the findings, specific root cause analysis and corrective recommendations.

The Automatic Shared Memory Management (ASMM) features introduced in 10gR1 solved this problem by providing the DBA a simple, automatic self-tuning mechanism for configuring shared memory components of the SGA, including the buffer cache and shared pool. The Automatic Database Diagnostic Monitor (also introduced in 10gR1) further simplified shared pool related tuning efforts by providing automatic diagnosis and recommendations for application specific issues.

This information in this paper is not required to use any of the automatic self-tuning features available in the server (such as ASMM or ADDM). Instead this paper will interest the DBA who has always wanted a solid working understanding of shared pool mechanisms. It is hoped this information will further empower this DBA to correctly reason about, comprehend and manually solve shared pool issues.

PURPOSE OF THE SHARED POOL

The shared pool was introduced as a feature of the Oracle database in version 7, primarily as a repository for Shared SQL and PL/SQL. Since that time, although it has evolved to serve a wider function, it still serves in its primary role for caching of cursors shared between sessions connected to the database. For the purposes of this paper, most of the features presented will refer to Oracle releases 10.1 and 10.2. In essence, the architecture and concepts discussed in this paper won't have changed much either, although the manageability features and init.ora parameters have evolved over time to meet the changing needs of the user community.

At the most fundamental level, the shared pool is a metadata cache. Whereas the buffer cache is almost completely utilized for the caching of data, the shared pool is used for caching complex objects describing where the data is stored, how it relates to other data and how it can be retrieved.

Much of the shared pool usage is to support the execution of shared SQL and PL/SQL packages; but in order to build a cursor or compile a PL/SQL procedure, we need to know all about the database objects referenced by the SQL or PL/SQL being compiled. For example, in order to build a cursor for a simple select statement from a table, we need to know metadata about the table including column names, data types, indexes and optimizer statistics. All of this additional metadata is also cached in the shared pool, independently of the cursors or program unit. By caching this metadata independently, it can be used to build any number of cursors, without incurring the overhead of many unnecessary trips to query the same data from the Oracle data dictionary.

SHARED POOL COMPONENTS

In addition to shared SQL and PL/SQL, there area many other components that occupy memory in the shared pool; most of which are shared globally between the Oracle sessions. A number of components are fixed in size and space is allocated for them at instance startup. As far as shared pool performance is concerned, there is usually very little that the DBA can do with regards to these permanent allocations, although there are some notable exceptions that will be discussed later.

Generally what is of far greater interest to the DBA, are the allocations of memory that can be aged in and out of the cache since building new objects in the shared pool is expensive and impacts scalability and performance. Memory allocations for objects that can be rebuilt are sometimes referred to as 'recreatable'. Just as in the buffer cache, where it is important to avoid unnecessary I/O from constantly reloading the same data blocks, it is just as important to size the shared pool to avoid the overhead of constantly reloading the same recreatable objects.

V$SGASTAT

The easiest way for examining the contents of the shared pool at a given time is to query the fixed view V$SGASTAT. Since Oracle keeps running totals of component memory allocations in the shared pool, selecting from this view is inexpensive and will not impact the production system. This view has been enhanced in 10gR2 to provide a finer-granularity of memory allocation data. The following query gives the top 10 components to have allocated memory in the pool:

```
select * from
     (select name,bytes/(1024*1024) MB
        from v$sgastat
       where pool ='shared pool'
       order by bytes desc)
 where rownum < 11

NAME                              MB
-------------------------- --------
sql area                     91.805
library cache                70.819
free memory                  22.294
CCursor                      21.940
PL/SQL MPCODE                19.032
gcs shadows                  14.250
```

```
PCursor                        14.168
PL/SQL DIANA                   13.689
row cache                      13.569
gcs resources                  13.175
```

Multiple selects from V$SGASTAT will show that some components remain constant in size, whilst others will change from query to query. It is this latter class of objects that hold most interest, since these are the 'recreatable' components with which we are most concerned for shared pool sizing. The following table maps the internal allocation comments found in V$SGASTAT to the key recreatable components:

	Component Description	10.1	10.2
SQL (cursors)	Executable representation of a SQL statement that may be used repeatedly by many sessions.	sql area library cache	sql area sql area:PLSQL PCursor CCursor CURSOR STATS
PL/SQL	Executable representation of PL/SQL packages and procedures that may be used repeatedly by many sessions.	library cache PL/SQL MPCODE PL/SQL DIANA	Heap0: KGL PL/SQL MPCODE PL/SQL DIANA
Library Cache	Cached meta-data for tables, indexes, views, synonyms and other database objects. Also used for mapping dependencies and other relationships between objects including SQL and PL/SQL.	library cache KGLS heap KGL Handles	Heap0: KGL library cache KGLS heap KGL Handles
Row Cache	The dictionary cache – cached rows from the data dictionary used to build more complex SQL and Library Cache objects.	KQR S PO KQR M PO KQR L PO KQR S SO KQR M SO KQR L SO	KQR S PO KQR M PO KQR L PO KQR S SO KQR M SO KQR L SO

Note that the data reported in V$SGASTAT is really a summation of all the discrete allocations made for each component. There is no implication that the data represents any contiguous allocation for any given component, nor that any component has an area or the shared pool reserved solely for it's own use (although that is sometimes the case for the large permanent allocations made at startup).

Cache Dynamics

The fluctuations in the sizes of all of these components, is driven by the process of building new SQL and PL/SQL objects within the cache, a process which itself is driven by the dynamics of the application. Each of these components participate at some level in this process, with memory constantly in a state of flux, being freed from one component for allocation elsewhere. It should also be remembered that the relationship between the components is actually highly complex, and that it is unlikely that the compilation of any cursor or PL/SQL procedure would proceed in isolation. In fact, a 'hard parse' of a cursor could set in motion a chain of events which could be more or less costly, depending on the initial state of the caches.

The following diagram documents the typical relationships between the caches.

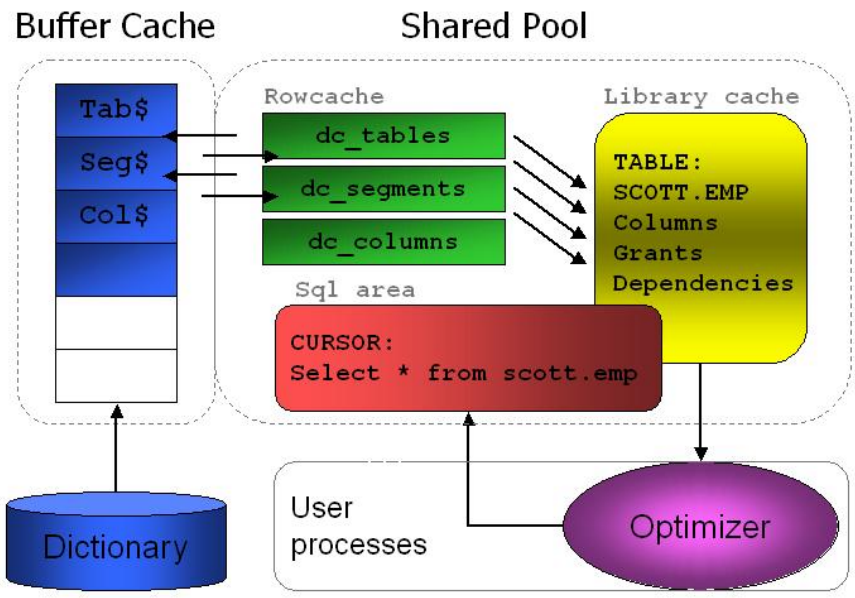

Figure 1: Shared pool cache dynamics of object creation

Object level view

At this point, it should be fairly obvious that each component has many hundreds of objects in the shared pool, where an object can be for example, a cursor, a table, a view or a PL/SQL package.

Usually these individual objects are not large, but they can grow to be several MB based on the object type and its defining attributes. Not surprisingly, a package body containing many procedures is likely to consume more space in the shared pool than a simple select statement.

The size and other details of individual objects in the shared pool can be obtained by querying the fixed view V$DB_OBJECT_CACHE (or related views such as V$SQL and V$SQLAREA). Note that querying these views may be expensive on large systems with high SQL throughput. For 10.2, Oracle recommends using V$SQLSTATS as a low impact alternative to the V$SQL family of views.

The query below returns the largest 2 objects from each Namespace in V$DB_OBJECT_ CACHE.

```
select *
  from (select row_number () over
              (partition by namespace
               order by sharable_mem desc) row_within
             , namespace, sharable_mem, substr(name,1,40) name
          from v$db_object_cache
         order by sharable_mem desc
       )
 where row_within <= 2
 order by namespace, row_within;


NAMESPACE         SHARABLE_MEM NAME
----------------- ------------ --------------------------------
BODY                   285,995 AWL_LRGREPORTS
BODY                   219,078 MGMT_JOB_ENGINE
CLUSTER                    354 C_OBJ#_INTCOL#
CLUSTER                    354 C_FILE#_BLOCK#
CURSOR                 343,612 INSERT INTO AIME$METRIC_SUMMARY
CURSOR                 294,572 INSERT INTO AIME$METRIC_VALUE SE
INDEX                    7,843 WRH$_INST_CACHE_TRANSFER_PK
INDEX                    7,738 WRH$_FILESTATXS_PK
TABLE/PROCEDURE        746,872 oracle/i18n/text/OraMapTable
TABLE/PROCEDURE        444,261 oracle/i18n/util/LocaleMapper
TRIGGER                  9,618 LABEL_BIU_ROW
TRIGGER                  5,526 LRGREPORT_BIU_ROW
...
```

Heaps

One of the popular misconceptions is that we can often run into problems when we try and allocate memory for one of these large objects. In extreme cases that can be true, but not this is not normally the case. In order to understand why, we have to understand that each individual object is not comprised of single large allocation, but is further partitioned into independent memory allocations called 'heaps'.

The number of heaps for an object depends on the object type. For example, a SQL cursor has 2 heaps: a smaller heap for the library cache metadata (usually called heap 0) and a larger heap containing the executable representation of the cursor (usually called sqlarea). Understanding this is important for comprehending the inner workings of some init.ora parameters like session_cached_cursors.

Although heaps themselves may be pretty large, each heap is itself comprised of one or more chunks of memory. The chunks comprising the heap do not need to be contiguously located, but the memory within each chunk must be contiguous. Usually (but not exclusively) each chunk is 4K or less in size. More details regarding 4K chunks of memory are discussed later in this paper.

The following diagram shows the heap layout for some typical shared pool objects:

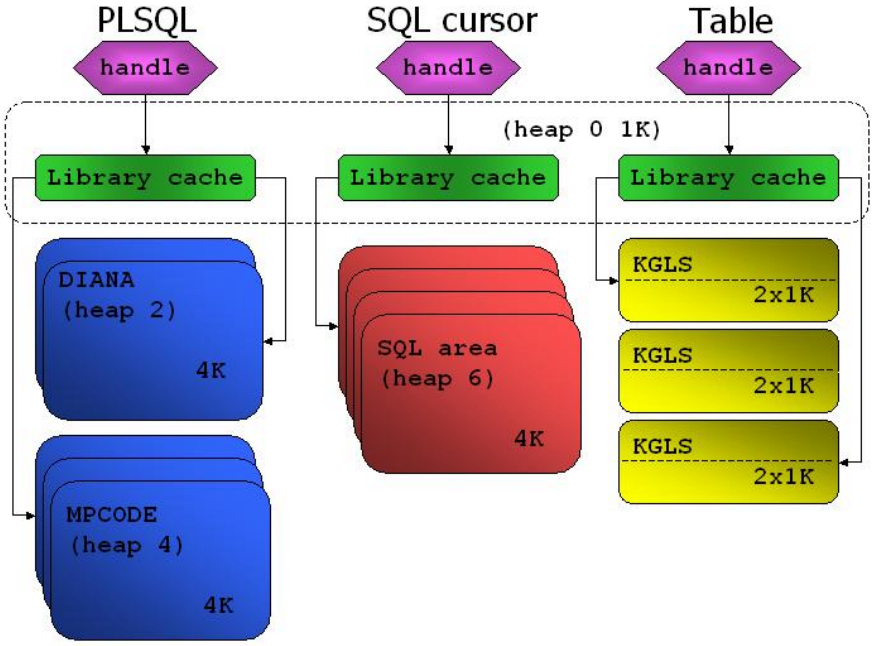

Figure 2: Memory heap structure for important library cache objects

SPACE ALLOCATION IN THE SHARED POOL

The first thing to understand here is that Oracle requires *contiguous* space to satisfy each memory request. For example, if a request is made for a 4K chunk of memory, then the heap manager (the bit of Oracle responsible for keeping track of shared pool space) cannot simply return a 3K chunk and a 1K chunk to the requestor; but must locate a 4K chunk of contiguous free memory to satisfy the request. If no such chunk exists, then a process begins of freeing batches of heaps from the shared pool, rechecking the free space after each batch to see if sufficient contiguous memory has been freed. In the usual steady state, we shouldn't have to free many objects to meet our request. This is because in the steady state, we are creating and destroying similar objects; thus for each cursor that we build, then on average we would need to free 1 cursor to make the necessary space available.

When the shared pool has no space to satisfy a request for memory then an ORA-4031 error is signaled. However, before signaling the error, the heap manager will always try to free as many objects as it can before giving up; it iterates through the

shared pool LRU list five times attempting to free space and so create a contiguous chunk of the requested size, before raising a 4031. For each chunk of memory freed, the memory is automatically coalesced wherever possible.

One of the most confusing aspects of the 4031 error is that it can be reported for relatively small allocations (say 4K) when there may be many megabytes of free space reported in the shared pool. This is somewhat counter-intuitive but should not be surprising. Since the ORA-4031 error is signaled after the heap manager has attempted to free all available space, of course there will be a large amount of free space available. None of this free memory however, will yield a contiguous extent large enough to satisfy the current request.

Note that an object in the shared pool cannot be freed while it is in use. For SQL and PL/SQL this means that any cursor or package currently executing cannot be aged out. Table and index meta-data cannot be freed while it is being used to build a cursor or package, or while it is participating in a DDL. The same principal applies to row cache entries. Objects in the shared pool that cannot be aged out are often referred to as 'pinned'. Pinning and unpinning is usually managed automatically by Oracle, although manually pinning (or keeping) objects can be achieved though use of the dbms_shared_pool package. Some init.ora parameters also have an impact on pinning and unpinning and these are discussed later in this paper.

We mentioned earlier that shared pool objects normally consist of many individual memory allocations, called chunks. In order that the process of allocating and freeing memory is made as efficient as possible, many of these chunks are standardized on a few chosen allocation sizes. The most prevalent units of memory allocation currently in use are 1K and 4K, although there are also many smaller units of allocation for components like the row cache, for example.

A small percentage of the memory allocations in the shared pool will exceed 4K. These larger allocations could cause a problem, since Oracle could free hundred's of objects and never get anything back large enough to meet the request. To address this issue, an area of the shared pool is set aside and only used for allocations greater than 4K; this is known as the reserved pool and by default it occupies 5% of the shared pool size. Note – large allocations are not equivalent to large objects! A large object may comprise totally of 4K allocations and never touch the reserved pool. Note also that the reserved pool will only be used if insufficient space is available in the shared pool; it is quite acceptable for large allocations to be made from the normal pool if space is available.

An example of a large contiguous allocation that one might see in a 10g database would be the 'session param values' memory allocated at the start of each session. These allocations could be as much as 30K and would typically be allocated from the reserved pool. If one were to get an ORA-4031 error signaled for an allocation such as this, the solution is usually to increase the size of the reserved pool to accommodate more, larger allocations.

In 10gR2 onwards, the attempted allocations of very large heaps (>2MB) by the Library Cache are signaled in the alert log (remember that a heap is comprised of non-contiguous chunks). The following is an example of such an alert log entry:

```
Wed Sep  7 15:56:56 2005
Memory Notification: Library Cache Object loaded into SGA
Heap size 3072K exceeds notification threshold (2048K)
Details in trace file /oracle/rdbms/log/main_ora_27395.trc
KGL object name :select xxx from yyy where ...
```

SQL AND PLSQL IN THE SHARED POOL

All SQL cursors and PL/SQL packages are allocated almost totally in 4K chunks. The bigger the cursor, the more 4K chunks we allocate to it; but rarely more than 4K unless there is some unusual requirement that needs to be met by the SQL optimizer or PLSQL runtime engine. Since the majority of allocation requests are made for 4K chunks, then all freed cursors will also return 4K chunks back to the pool. It follows that there should never really be a problem building a new cursor or PL/SQL package providing there are old cursors or packages that we can age out.

Parsing

Since we are on the subject of memory allocation for SQL and PL/SQL, it is also necessary to discuss the subject of parsing, since without hard parsing, there would rarely be any requirement to allocate space from the shared pool anyway!

Hard parsing affects the system at several levels. Even before the cursor can be built, the SQL has to be parsed and optimized. This process requires access to library cache and row cache resources and may incur recursive loading of library cache and row cache objects. This puts pressure on library cache, row cache and shared pool components and explains why high hard parse rates are often accompanied by latch contention for these systems.

The relative cost of soft parse to hard parse is several orders of magnitude less. However, even soft-parsing is not without cost. In particular, the lookup of the SQL statement in the shared pool and the lock and pin operations required to execute the cursor all incur latching overhead. Also, although the cost of soft parse is several orders of magnitude less, the frequency is often orders of magnitude more. This is why various SQL caching options have been introduced on both client and server sides to aid scalability on high throughput systems.

The most effective use of shared pool resources is to parse once, and execute many times; in this way the parse overhead is avoided. New applications should be coded with optimal resource usage in mind, to allow for maximum scalability.

Literals

Another application consideration is usage of literals. If there are many identical single-use SQL statements in the shared pool which differ only by the literal values, then this is a SQL sharing issue caused by the application generating SQL

statements with literals rather than bind variables. High-throughput OLTP systems should ideally use bind variables to allow for the re-use of existing SQL statements; if modifying the application is not possible, then consider using the init.ora parameter cursor_sharing set to FORCE to reduce the number of single-use statements which differ only in literals.

There are a number of different ways to identify literal SQL:

ADDM automatically detects literal SQL, its impact on the system and identifies the various ways of managing the problem.

- One method is to check V$SQLAREA for SQL statements which have the same execution plan (i.e. PLAN_HASH_VALUE), but differing SQL Id's (SQL_ID)
- A second method is to compare the leading *N* characters of SQL text in the V$SQLAREA view; those SQL statements with the same leading N characters may be the identical SQL with literals

CONFIGURING THE SHARED POOL

There is no single correct shared pool size for the vast majority of systems. For most, there are a range of values for the shared pool size that work well for the variation in load experienced at the site; experienced DBAs would use the 'not too big', 'not too small' rule of thumb to determine if any changes need to be made to the existing configuration.

However, arriving at the initial sizing of the shared pool has always been problematic because the dynamics of the application workload and the size of the global memory footprint are incredibly difficult to model. Even once an acceptable level of shared pool performance is attained for a given configuration, changes in the application, changes in user load and changes in initialization parameters can all negatively impact the system. Given that a system has been established and that a representative workload can be executed, the following section gives some useful tips on what to look for when assessing the SGA requirements.

Reactive tuning should be driven by the biggest bottleneck in the system. For example, using systematic sizing methodology for tuning the shared pool size is inappropriate if the problem is caused by a poorly performing SQL statement exhausting IO system resources.

Note that proactive sizing of the shared pool, and reactive tuning when problems arise should be handled differently; reactive tuning should always be driven by the biggest bottleneck in the system. The biggest bottleneck should be determined by examining the wait events along with the time model (V$SYS_TIME_MODEL) data. This data can be used to identify which is the largest problem area, and hence what to concentrate on.

Looking at where the most database time ('DB time' statistic) is spent during the problem period provides a method of identifying which areas require attention. For example, the following Time Model data from an AWR report indicates that the majority of time is spent executing SQL statements (98%), but only very little time is spent performing parse-related operations (2.5%):

```
Statistic Name                          Time (s) % of DB time
--------------------------------------- -------- ------------
sql execute elapsed time                 1,902.2         98.7
```

DB CPU	776.5	40.3
PL/SQL execution elapsed time	67.7	3.5
parse time elapsed	47.8	2.5
hard parse elapsed time	32.4	1.7
PL/SQL compilation elapsed time	4.6	.2
connection management call elapsed time	3.8	.2
hard parse (sharing criteria) elapsed ti	1.6	.1
hard parse (bind mismatch) elapsed time	0.6	.0
sequence load elapsed time	0.6	.0
Java execution elapsed time	0.5	.0
inbound PL/SQL rpc elapsed time	0.3	.0
repeated bind elapsed time	0.1	.0
failed parse elapsed time	0.0	.0
DB time	1,926.9	N/A
background elapsed time	285.9	N/A
background cpu time	156.7	N/A

Undersized Pool

Of the two possible extremes, 'too small' is definitely the most problematic. Having a shared pool that is too small could result in a myriad of performance issues:

- ORA-04031 out of shared pool memory
- Library cache lock and pin contention (high library cache reloads)
- Row cache enqueue contention (high row cache reloads)
- Latch contention (shared pool, library cache, row cache)

These should be examined in the order suggested here.

ORA-04031

This error should not appear in any of the application logs, the alert log or any trace files. Do not depend on ORA-04031 errors being written to the alert log, as 4031 errors only appear in the alert log if they affect background process operations (such as PMON activities). 4031's are not internal errors and so could be trapped and handled by the application (this is not recommended).

From 10gR1 onwards, a 4031 trace file is written to the user_dump_dest (or background_dump_dest) directory; this trace file is useful in diagnosing the nature of problem.

Library Cache Reloads

The shared pool should be large enough to avoid the excessive overhead from constantly reloading the same recreatable objects.

Before sizing the shared pool to avoid reloads, you should check that the reloads are not a consequence of invalidations. If the number of invalidations is a significant proportion of the number of reloads (where significant is any number greater than 20%), then investigate the cause and fix the source of the invalidations first.

The following query from V$LIBRARYCACHE will show haw many reloads and invalidations have occurred since instance startup. Use EM, AWR or Statspack to see this data for a given time interval.

The best way to look at the data in the majority of V$ performance views is to examine the data delta'd over an interval. EM, AWR or Statspack perform this function for you.

```
select namespace, pins, pins-pinhits loads
     , reloads, invalidations
     , 100*(reloads-invalidations)/(pins-pinhits) "%RELOADS"
  from v$librarycache
 where pins >0
 order by namespace;
```

```
NAMESPACE              PINS     LOADS  RELOADS INVALI %RELOADS
--------------- ------------ -------- -------- ------ --------
BODY             84,964,544      120       22      0       18
CLUSTER                 524       11        1      0        9
INDEX                 4,827      101       29      0       29
JAVA DATA             1,201      126      118      0       94
JAVA RESOURCE         3,223       18        0      0        0
SQL AREA        743,726,359  135,945  123,991    339       91
TABLE/PROCEDURE 101,451,378   11,199    1,751      0       16
TRIGGER              19,875       39       14      0       36
```

Although there are no absolute recommendations, library cache reloads should be only a small percentage (say 10%) of the total loads on the system.

Total loads can be calculated from (pins – pinhits) in V$LIBRARYCACHE. If there are is a significant number of invalidations on the system, then these should be also accounted for, since an invalidation will incur a reload in most cases.

Row Cache Misses

If library cache reloads are high, then it is likely that dictionary cache misses will be high also. Query V$ROWCACHE to view this data.

```
select parameter
     , sum(gets)      gets
     , sum(getmisses) getmisses
     , sum("COUNT")   num
     , sum(usage)     usage
  from v$rowcache
 where getmisses > 0
 group by parameter
 order by parameter;
```

```
PARAMETER                      GETS    GETMISSES        COUNT
---------------------- ------------ ------------ ------------
dc_awr_control                6,115          100            1
dc_constraints                4,203        2,441           18
dc_database_links            54,689           74            6
dc_files                        456           78            0
dc_histogram_data        43,002,286       40,017        3,820
dc_histogram_defs        16,657,962       79,374        5,211
dc_object_grants            200,013          963          150
dc_objects                1,338,748       19,037        1,260
dc_rollback_segments     12,934,400          113          113
dc_segments               6,712,351       19,672        1,488
dc_sequences                 11,227        2,614            9
dc_tablespace_quotas         23,608          254            3
dc_tablespaces            6,578,129           33            8
dc_usernames                250,343          194           21
dc_users                 15,806,141          390           67
```

. . .

Note that row cache misses are very cheap compared to library cache misses. Each row cache miss will result in a single row fetch from the data dictionary (assuming we don't have to hard parse the recursive query).

Latch Contention

The latches most affected by under-sizing are the shared pool, library cache and row cache latches. If there are no 4031 errors or problems due to reloads, then there should be little or no latch contention for these latches related to the shared pool size. If you see contention on any other latches, then it's not due to the shared pool size but to some other unrelated issue.

However, there are other conditions that may also result in latch contention on the shared pool, library cache and row cache latches when the shared pool may be perfectly adequately sized. These include very high hard parse rates, exceptionally high soft parse and execution rates, excessive shared pool monitoring activity and non-standard init.ora parameters.

Latch contention issues, particularly those due to soft parse and execution rates, may be addressed by setting some of the init.ora parameters that are described later. However, note that changing these parameters will also change the dynamics of the shared pool, requiring the tuning cycle to be repeated. For example, setting the parameter cursor_space_for_time may reduce latch contention for SQL execution but increase latch contention due to reloads (more on this later).

Corrective Actions

Remember when working to improving the performance of your system, always look at reducing or eliminating the major bottleneck in the system. Looking at potential problematic statistics such as high reloads in isolation of the primary bottleneck, will not provide the best performance solution for your problem.

The most obvious corrective action for 4031 errors is to increase the shared pool size. The view V$SHARED_POOL_ADVICE may be useful here in recommending the best shared pool size to avoid reloads.

As mentioned earlier, if the cause of the 4031 error is a large allocation (greater than 4K), the corrective action should be to increase the reserved pool.

If increasing the shared pool size is impossible, there are other solutions to relieve space pressure on the shared pool. For example, using PL/SQL native compilation will move all PL/SQL packages from being resident in the shared pool.

Space pressure is also increased by large setting of session_cached_cursors and by using cursor_space_for_time. Consider whether the setting value of the session cursor cache could be reduced, and whether cursor_space_for_time is needed at your site (particularly if using mutexes with 10.2.0.2).

Oversized Pool

If you have determined the shared pool is not too small, then you may consider downsizing to release memory back to the system.

An oversized shared pool may only be an issue if you have run out of physical memory on your host and the system is paging badly. For this reason it is usually easier to start large and then gradually decrease the pool size. This however has not always been the case. Historically, over-sizing the shared pool has resulted in latch related performance problems due to issues with free list management, but these issues were fixed by Oracle8.

Generally speaking, very large shared pools (Gigabytes in size), do not have any performance issues related to scalability, but may have issues specifically related to performance monitoring; these issues exist regardless of whether the pool is over-sized or not. This is because some of the performance tuning views such as V$SQL and V$OPEN_CURSORS have such a huge number of objects to traverse, that querying these views may result in latch contention. In 10gR2, the addition of new views and new implementations of some of the existing views will help avoid this problem.

When evaluating whether the shared pool is too large, then you may consider looking at the single use SQL. Most systems have ad-hoc SQL statements, which are executed once only. Ideally, it is these single use statements that are the ones that we want to age out when space is needed to build another cursor. As long as the shared pool has say 10 – 30% of single use statements around, then the pool is not oversized. If the number is drastically higher, then memory is probably being wasted. Whether that wastage is significant or not will depend on other factors like memory available for buffer cache, system paging etc.

The query below shows the number of single-execution (single-use) SQL statements, and also how much memory those statements occupy.

```
select count(1) num_sql
     , sum(decode(executions, 1, 1, 0)) num_1_use_sql
     , sum(sharable_mem)/1024/1024  mb_sql_mem
     , sum(decode(executions, 1, sharable_mem, 0))/1024/1024
           mb_1_use_sql_mem
  from v$sqlarea
 where sharable_mem > 0;

    NUM_SQL  NUM_1_USE_SQL   MB_SQL_MEM MB_1_USE_SQL_MEM
----------- ------------- ------------ ----------------
     18,411          1,269       447.63            50.42
```

When looking at the single-use SQL, you should look at the SQL to see whether it is truly ad-hoc, or whether the SQL is identical save for literals (see the section above on Literals for more details).

Tuning knobs

There are several parameters that are usually set to improve the scalability and performance of the instance and in particular, shared cursors. However, while it is fairly easy to measure the benefits of these settings, what is less commonly known is the impact that these parameters may have on your shared pool sizing.

Session cached cursors

To reduce contention for high-concurrency SQL statements that are closed between executions, it is often recommended that DBAs set the init.ora parameter session_cached_cursors. When enabled, this feature optimizes the way in which an Oracle session searches for cursors at parse time, by keeping a local session cache referencing the cursors most recently closed. To prevent bloating of the cache, only cursors that have been parsed more than 3 times by any session connected to the instance are candidates for addition to the session cursor cache. For each new parse, the session will optimistically search it's own session cursor cache, to see if the cursor can be located.

Why is this a good thing? When a parse call is issued, a session will usually locate the required cursor in the shared pool, assuming that the statement already exists. As mentioned earlier, if a compatible cursor is located, then this is called a soft parse. Despite using significantly less resources than hard parses, soft-parsing SQL statements still incurs library cache latch and CPU overhead that could prove significant in high throughput systems.

Remember that the optimal method for reducing use of parse-related resources is to keep cursor open and re-execute it.

Thus the benefit of session_cached_cursors is avoiding the overhead of a session locating the SQL statement in the shared pool when it next attempts to re-parse it. Avoiding this search means there will be fewer library cache latch gets required to locate the cursor, and also less CPU. Fewer latch gets on a high throughput system also means less latch contention for the library cache latches and greater scalability. Concurrency improvement will only apply to cursors that are closed, and then subsequently reparsed multiple times.

However, in order to achieve this optimization, Oracle must ensure that any cursor referenced in the cursor cache *of any session*, must remain 'partially pinned' down in the shared pool. The more cursors pinned in the shared pool, the fewer the candidate cursors available for freeing memory. If there are a large number of cursors pinned, it may mean you will need to allocate more shared pool memory to ensure you don't run out of memory. So if you set this parameter, or increase it, you may want to check that your shared pool performance has not been negatively impacted, even though you soft parse throughput may have improved significantly.

What do we mean by a 'partially pinned' cursor? Remember we discussed how cursors were allocated in two distinct heaps (heap 0 and sqlarea). When a cursor is added to the session cursor cache, this results in heap 0 being pinned in the shared pool, but not the sqlarea heap. Heap 0 is the smaller of the heaps, so the overhead is not huge (about 1K per cursor in the cache), but may still impact sizing if enough cursors are cached.

We should note here also, that this parameter also has an impact on PL/SQL. In addition to defining how many cursors may be cached per session in the session cursor cache, it also defines how many cursors may be cached by PL/SQL. The PL/SQL cursor cache exists and is managed independently of the session cursor cache. Just to confuse matters, the PL/SQL cache is not a closed cursor cache;

rather the cursors are cached in an open state. This is important with reference to the discussion on 'cursor space for time'.

Oracle uses a least recently used algorithm to replace entries in the session cursor cache to make room for new entries. The maximum number of cached cursors is the value of the parameter.

Cursor space for time

cursor_space_for_time is not frequently used, as it is only necessary for extremely high-concurrency OLTP systems.

In 10gR2, it is not necessary to use cursor_space_for_time to reduce contention for frequently executed cursors, as a new serialization mechanism which is smaller and faster than latches is available. This has the additional benefits of not locking down the cursor memory in the shared pool, and not requiring the cursor to remain open.

Cursor_space_for_time is an initialization parameter which can be used for optimizing the repeated execution of cursors that are parsed once and kept open. The distinction is important: the cursor must remain open for cursor_space_for_time to apply; concurrency improvement will not apply to cursors that are parsed, executed and immediately closed. As with most performance features, there is a space-time trade off, so understanding when it should be used, and the potential negative impact is important.

When executing a cursor, a session must pin the cursor to prevent it aging out of the shared pool while it is being executed. After the session completes executing the cursor, it normally deletes the pin, making the cursor a candidate for aging from the shared pool once more. When cursor_space_for_time is TRUE, a session executing a cursor will pin the cursor before execution, but the pin will not be deleted once the execution is complete. In fact, the pin is not deleted until the application actually closes the cursor.

Thus the benefit of cursor_space_for_time is to avoid the re-pin of a cursor each time a session wants to re-execute it. Avoiding pinning the cursor means there will be fewer library cache latch gets required to re-execute the cursor (both pin and unpin are latched operations) and less CPU utilization. Fewer latch gets on a highly-executed cursor means less latch contention for the library cache and library cache pin latches and therefore a system with greater scalability.

The downside of setting this parameter is that while a cursor is pinned by any session, it cannot be freed. Note that each session executing a cursor will have its own pin on that cursor. All sessions holding the cursor open must close it before the cursor becomes a candidate for aging out. This has huge implications for shared pool memory allocation and sizing.

Unlike session_cached_cursors where only heap 0 of the cursor is pinned, with cursor_space_for_time we pin the entire cursor in the pool until the cursor is closed. This is a significantly greater overhead since the sqlarea heap is usually many times larger than heap 0, possibly even Megabytes for highly complex queries.

The more cursors kept open in the shared pool, the fewer the candidate cursors available for freeing memory. If there are a large number of cursors kept open in the instance, it may mean you will need to allocate more shared pool memory to allow for pinning frequently executed open-cursors in memory.

Understanding this behaviour is particularly important with regards to PL/SQL based applications. PL/SQL functions and procedures will always cache cursors in an open state, even if the application explicitly closes the cursor in the code. This optimization means that re-execution of cursors from PL/SQL is always fast, but it also means that even closed PL/SQL cursors can pin down significant shared pool memory when this parameter is set.

Keeping objects in the pool

Sometimes it may be necessary to ensure that large objects that are expensive to load remain in the shared pool whenever possible. Large objects includes those objects which are comprised of many small chunks, as well as objects which are comprised of large allocations. Keeping objects avoids potential problems for getting ORA-04031 or performance problems due to a large spike in space allocation.

This type of problem is usually seen with objects that are sufficiently large so that they can impact shared pool performance when being reloaded. If such an object is only accessed occasionally, it may have may have enough time to age out of the cache between invocations. Keeping objects is not strictly necessary for objects that are in constant use, since they should never age out of the shared pool.

The method for keeping objects in the shared pool is through the dbms_shared_pool package. Keeping an object guarantees the object does not age out of the shared pool, once loaded. Objects that can be kept include PL/SQL objects, triggers, sequences, types and Java objects and SQL cursors. If you are keeping a large number of objects it may be worthwhile considering keeping objects immediately after instance startup to insure that sufficient free space is available for allocation and to avoid unnecessary fragmentation of the shared pool.

The following queries for kept objects in the shared pool:

```
select namespace, count(1) num_objects
     , sum(sharable_mem)/1024 kb
  from v$db_object_cache
 where kept = 'YES'
 group by namespace
 order by kb desc;
```

```
NAMESPACE                       NUM_OBJECTS              KB
--------------------------- --------------- ---------------
TABLE/PROCEDURE                         531        8,937.38
RSRC CONSUMER GROUP                       2        4,565.88
CURSOR                                1,319        2,202.17
INDEX                                     7            2.43
CLUSTER                                   6            2.04
JAVA SHARED DATA                          2             .00
```

The more objects kept in the shared pool, the fewer the candidate cursors available for freeing memory. If a lot of memory is dedicated to kept objects, you may need to allocate more shared pool memory to avoid out of memory (i.e. 4031) errors.

Keeping is not guaranteed for objects that become invalid through DDL. Invalid objects must be purged from the cache, regardless of whether they are kept or not. These objects will be automatically re-pinned when they are reloaded.

ORACLE 10G R2 IMPROVEMENTS

Extended use of Standard Chunk sizes

In 10gR2, the server has been enhanced to further leverage standard chunk allocation sizes. This additional improvement reduces the number of problems arising from memory fragmentation.

Mutexes

To improve cursor execution and also hard parsing, a new memory serialization mechanism has been created in 10gR2. For certain shared-cursor related operations, mutexes are used as a replacement for library cache latches and library cache pins. Using mutexes is faster, uses less CPU and also allows significantly improved concurrency over the existing latch mechanism. The use of mutexes for cursor pins can be enabled by setting the init.ora parameter _use_kks_mutex to TRUE.

V$SGASTAT

As mentioned earlier, V$SGASTAT has been enhanced to display a finer granularity of memory to component allocation within the shared pool. This allows faster diagnosis of memory usage (in prior releases many smaller allocations were grouped under the 'miscellaneous' heading).

V$SQLSTAT

A new view, V$SQLSTAT has been introduced which contains SQL related statistics (such as CPU time, elapsed time, sharable memory). This view is very cheap to query even on high-concurrency systems, as it does not require library cache latch use. It contains the most frequently used SQL statistics in the V$SQL family of views.

V$OPEN_CURSOR

This implementation of this view has also been enhanced to be latchless, making it inexpensive to query.

V$SQLAREA

The V$SQLAREA view has been improved in 10gR2; the view optimizes the aggregation of the SQL statements while generating the view data.

CONCLUSION

This paper has provided an insight into the internal mechanisms of the shared pool, focusing on its most common inhabitants – SQL and PL/SQL.

It is hoped that an understanding of the principles outlined here will provide the reader with a better comprehension of the tuning configuration options related to SQL performance. It is also hoped that the reader will have an improved knowledge of tips and techniques to employ when monitoring the performance and usage of the shared pool.

Understanding Shared Pool Memory Structures
September 2005
Author: Russell Green
Contributing Authors: Connie Dialeris Green, John Beresniewicz

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com