15

Tuning the Shared Pool and the Large Pool

This chapter describes how to tune the shared pool and the large pool. If you are using automatic memory management to manage the database memory on your system, or automatic shared memory management to configure the Shared Global Area (SGA), there is no need to manually tune the shared pool and the large pool as described in this chapter.

This chapter contains the following topics:

About the Shared Pool

Oracle Database uses the shared pool to cache many different types of data. Cached data includes the textual and executable forms of PL/SQL blocks and SQL statements, dictionary cache data, result cache data, and other data.

This section describes the shared pool and contains the following topics:

Benefits of Using the Shared Pool

Proper use and sizing of the shared pool can reduce resource consumption in at least four ways:

- If the SQL statement is in the shared pool, parse overhead is avoided, resulting in reduced CPU resources on the system and elapsed time for the end user.
- Latching resource usage is significantly reduced, resulting in greater scalability.
- Shared pool memory requirements are reduced, because all applications use the same pool of SQL statements and dictionary resources.
- I/O is reduced, because dictionary elements that are in the shared pool do not require disk access.

Shared Pool Concepts

The main components of the shared pool include:

- Library cache

  The library cache stores the executable (parsed or compiled) form of recently referenced SQL and PL/SQL code.
- Data dictionary cache

The data dictionary cache stores data referenced from the data dictionary.

- Server result cache (depending on the configuration)

  The server result cache is an optional cache that stores query and PL/SQL function results within the shared pool. For information about the server result cache, see "About the Result Cache".

Many of the caches in the shared pool—including the library cache and the dictionary cache—automatically increase or decrease in size, as needed. Old entries are aged out to accommodate new entries when the shared pool runs out of space.

A cache miss on the library cache or data dictionary cache is more expensive than a miss on the buffer cache. For this reason, the shared pool should be sized to ensure that frequently-used data is cached.

Several features require large memory allocations in the shared pool, such as shared server, parallel query, or Recovery Manager. Oracle recommends using a separate memory area—the large pool—to segregate the System Global Area (SGA) memory used by these features.

Allocation of memory from the shared pool is performed in chunks. This chunking enables large objects (over 5 KB) to be loaded into the cache without requiring a single contiguous area. In this way, the database reduces the possibility of running out of contiguous memory due to fragmentation.

Java, PL/SQL, or SQL cursors may sometimes make allocations out of the shared pool that are larger than 5 KB. To enable these allocations to occur more efficiently, Oracle Database segregates a small amount of the shared pool. The segregated memory, called the reserved pool, is used if the shared pool runs out of space.

The following sections provide more details about the main components of the shared pool:

- Library Cache Concepts
- Data Dictionary Cache Concepts
- SQL Sharing Criteria

Library Cache Concepts

The library cache stores executable forms of SQL cursors, PL/SQL programs, and Java classes, which are collectively referred to as the application code. This section focuses on tuning as it relates to the application code.

When the application code is executed, Oracle Database attempts to reuse existing code if it has been executed previously and can be shared. If the parsed representation of the SQL statement exists in the library cache and it can be shared, then the database reuses the existing code. This is known as a soft parse, or a library cache hit. If Oracle Database cannot use the existing code, then the database must build a new executable version of the application code. This is known as a hard parse, or a library cache miss. For information about when SQL and PL/SQL statements can be shared, see "SQL Sharing Criteria".

In order to perform a hard parse, Oracle Database uses more resources than during a soft parse. Resources used for a soft parse include CPU and library cache latch gets. Resources required for a hard parse include additional CPU, library cache latch gets, and shared pool latch gets. A hard parse may occur on either the parse step or the execute step when processing a SQL statement.

When an application makes a parse call for a SQL statement, if the parsed representation of the statement does not exist in the library cache, then Oracle Database parses the statement

and stores the parsed form in the shared pool. To reduce library cache misses on parse calls, ensure that all sharable SQL statements are stored in the shared pool whenever possible.

When an application makes an execute call for a SQL statement, if the executable portion of the SQL statement is aged out (or deallocated) from the library cache to make room for another statement, then Oracle Database implicitly reparses the statement to create a new shared SQL area for it, and executes the statement. This also results in a hard parse. To reduce library cache misses on execution calls, allocate more memory to the library cache.

For more information about hard and soft parsing, see "SQL Execution Efficiency".

Data Dictionary Cache Concepts

Information stored in the data dictionary cache includes:

- Usernames
- Segment information
- Profile data
- Tablespace information
- Sequence numbers

The data dictionary cache also stores descriptive information, or metadata, about schema objects. Oracle Database uses this metadata when parsing SQL cursors or during the compilation of PL/SQL programs.

SQL Sharing Criteria

Oracle Database automatically determines whether a SQL statement or PL/SQL block being issued is identical to another statement currently in the shared pool.

To compare the text of the SQL statement to the existing SQL statements in the shared pool, Oracle Database performs the following steps:

1. The text of the SQL statement is hashed.

   If there is no matching hash value, then the SQL statement does not currently exist in the shared pool, and a hard parse is performed.

2. If there is a matching hash value for an existing SQL statement in the shared pool, then the text of the matched statement is compared to the text of the hashed statement to verify if they are identical.

   The text of the SQL statements or PL/SQL blocks must be identical, character for character, including spaces, case, and comments. For example, the following statements cannot use the same shared SQL area:

   ```
   SELECT * FROM employees;
   SELECT * FROM Employees;
   SELECT *  FROM employees;
   ```

   Also, SQL statements that differ only in literals cannot use the same shared SQL area. For example, the following statements do not resolve to the same SQL area:

   ```
   SELECT count(1) FROM employees WHERE manager_id = 121;
   SELECT count(1) FROM employees WHERE manager_id = 247;
   ```

   The only exception to this rule is when the `CURSOR_SHARING` parameter is set to `FORCE`, in which case similar statements can share SQL areas. For information about the costs and benefits involved in cursor sharing, see "Sharing Cursors".

3. The objects referenced in the issued statement are compared to the referenced objects of all existing statements in the shared pool to ensure that they are identical.

   References to schema objects in the SQL statements or PL/SQL blocks must resolve to the same object in the same schema. For example, if two users each issue the following SQL statement but they each have their own `employees` table, then this statement is not considered identical, because the statement references different tables for each user:

   ```
   SELECT * FROM employees;
   ```

4. Bind variables in the SQL statements must match in name, data type, and length.

   For example, the following statements cannot use the same shared SQL area, because the bind variable names are different:

   ```
   SELECT * FROM employees WHERE department_id = :department_id;
   SELECT * FROM employees WHERE department_id = :dept_id;
   ```

   Many Oracle products, such as Oracle Forms and the precompilers, convert the SQL before passing statements to the database. Characters are uniformly changed to uppercase, white space is compressed, and bind variables are renamed so that a consistent set of SQL statements is produced.

5. The session's environment must be identical.

   For example, SQL statements must be optimized using the same optimization goal.

> **See Also:**
>
> *Oracle Database Reference* for information about the `CURSOR_SHARING` initialization parameter

Using the Shared Pool

An important purpose of the shared pool is to cache the executable versions of SQL and PL/SQL statements. This enables multiple executions of the same SQL or PL/SQL code to be performed without the resources required for a hard parse, which results in significant reductions in CPU, memory, and latch usage.

The shared pool is also able to support unshared SQL in data warehousing applications, which execute low-concurrency, high-resource SQL statements. In this situation, using unshared SQL with literal values is recommended. Using literal values rather than bind variables enables the optimizer to make good column selectivity estimates, thus providing an optimal data access plan.

In a high-currency online transaction processing (OLTP) system, efficient use of the shared pool significantly reduces the probability of parse-related application scalability issues. There are several ways to ensure efficient use of the shared pool and related resources in an OLTP system:

- Use Shared Cursors
- Use Single-User Logon and Qualified Table Reference
- Use PL/SQL
- Avoid Performing DDL Operations
- Cache Sequence Numbers

- Control Cursor Access
- Maintain Persistent Connections

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for information about impact of parallel query execution on the shared pool

Use Shared Cursors

Reuse of shared SQL for multiple users running the same application avoids hard parsing. Soft parses provide a significant reduction in the use of resources, such as the shared pool and library cache latches.

To use shared cursors:

- Use bind variables instead of literals in SQL statements whenever possible.

  For example, the following two SQL statements cannot use the same shared area because they do not match character for character:

  ```
  SELECT employee_id FROM employees WHERE department_id = 10;
  SELECT employee_id FROM employees WHERE department_id = 20;
  ```

  Replacing the literals with a bind variable results in only one SQL statement which can be executed twice:

  ```
  SELECT employee_id FROM employees WHERE department_id = :dept_id;
  ```

  For existing applications where rewriting the code to use bind variables is not possible, use the `CURSOR_SHARING` initialization parameter to avoid some of the hard parse overhead, as described in "Sharing Cursors".
- Avoid application designs that result in large numbers of users issuing dynamic, unshared SQL statements.

  Typically, the majority of data required by most users can be satisfied using preset queries. Use dynamic SQL where such functionality is required.
- Ensure that users of the application do not change the optimization approach and goal for their individual sessions.
- Establish the following policies for application developers:
  - Standardize naming conventions for bind variables and spacing conventions for SQL statements and PL/SQL blocks.
  - Consider using stored procedures whenever possible.

    Multiple users issuing the same stored procedure use the same shared PL/SQL area automatically. Because stored procedures are stored in a parsed form, their use reduces run-time parsing.
- For SQL statements which are identical but are not being shared, query the `V$SQL_SHARED_CURSOR` view to determine why the cursors are not shared.

  This includes optimizer settings and bind variable mismatches.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about cursor sharing

Use Single-User Logon and Qualified Table Reference

In large OLTP systems where users log in to the database with their own user logon, qualifying the segment owner explicitly instead of using public synonyms significantly reduces the number of entries in the dictionary cache.

An alternative to qualifying table names is to connect to the database through a single user logon, rather than individual user logons. User-level validation can take place locally on the middle tier.

Use PL/SQL

Using stored PL/SQL packages can overcome many of the scalability issues for systems with thousands of users, each with individual user logon and public synonyms. This is because a package is executed as the owner, rather than the caller, which reduces the dictionary cache load considerably.

> **Note:**
>
> Oracle encourages the use of definer's rights packages to overcome scalability issues. The benefits of reduced dictionary cache load are not as great with invoker's rights packages.

Avoid Performing DDL Operations

Avoid performing DDL operations on high-usage segments during peak hours. Performing DDL operations on these segments often results in the dependent SQL being invalidated and reparsed in a later execution.

Cache Sequence Numbers

Allocating sufficient cache space for frequently updated sequence numbers significantly reduces the frequency of dictionary cache locks, which improves scalability.

To configure the number of cache entries for each sequence:

- Use the `CACHE` keyword in the `CREATE SEQUENCE` or `ALTER SEQUENCE` statement.

Control Cursor Access

Depending on your application tool, you can control how frequently the application performs parse calls by controlling cursor access.

The frequency with which the application either closes cursors or reuses existing cursors for new SQL statements affects the amount of memory used by a session, and often the amount of parsing performed by that session. An application that closes cursors or reuses cursors (for

a different SQL statement) does not require as much session memory as an application that keeps cursors open. Conversely, that same application may need to perform more parse calls, using more CPU and database resources

Cursors associated with SQL statements that are not executed frequently can be closed or reused for other statements, because the likelihood of re-executing (and reparsing) that statement is low. Extra parse calls are required when a cursor containing a SQL statement that will be re-executed is closed or reused for another statement. Had the cursor remained open, it may have been reused without the overhead of issuing a parse call.

The ways in which you control cursor access depends on your application development tool. This section describes the methods used for Oracle Database tools:

- Controlling Cursor Access Using OCI
- Controlling Cursor Access Using Oracle Precompilers
- Controlling Cursor Access Using SQLJ
- Controlling Cursor Access Using JDBC
- Controlling Cursor Access Using Oracle Forms

> **See Also:**
>
> The tool-specific documentation for information about each tool

Controlling Cursor Access Using OCI

When using Oracle Call Interface (OCI), do not close and reopen cursors that you will be re-executing. Instead, leave the cursors open, and change the literal values in the bind variables before execution.

Avoid reusing statement handles for new SQL statements when the existing SQL statement will be re-executed in the future.

Controlling Cursor Access Using Oracle Precompilers

When using the Oracle precompilers, you can control when cursors are closed by setting precompiler clauses. In Oracle mode, the clauses are as follow:

- `HOLD_CURSOR = YES`
- `RELEASE_CURSOR = NO`
- `MAXOPENCURSORS = desired_value`

The precompiler clauses can be specified on the precompiler command line or within the precompiler program. Oracle recommends that you not use ANSI mode, in which the values of `HOLD_CURSOR` and `RELEASE_CURSOR` are switched.

> **See Also:**
>
> Your language's precompiler manual for information about these clauses

Controlling Cursor Access Using SQLJ

Prepare the SQL statement, then re-execute the statement with the new values for the bind variables. The cursor stays open for the duration of the session.

> **Note:**
>
> Starting with Oracle Database 12*c* Release 2 (12.2), server-side SQLJ code is not supported, that is, you cannot use SQLJ code inside stored procedures, functions, and triggers.

Controlling Cursor Access Using JDBC

Avoid closing cursors if they will be re-executed, because the new literal values can be bound to the cursor for re-execution. Alternatively, JDBC provides a SQL statement cache within the JDBC client using the `setStmtCacheSize()` method. Using this method, JDBC creates a SQL statement cache that is local to the JDBC program.

> **See Also:**
>
> *Oracle Database JDBC Developer's Guide* for information about using the JDBC SQL statement cache

Controlling Cursor Access Using Oracle Forms

With Oracle Forms, it is possible to control some aspects of cursor access at run time, the trigger level, or the form level.

Maintain Persistent Connections

Large OLTP applications with middle tiers should maintain connections, instead of connecting and disconnecting for each database request. Maintaining persistent connections saves CPU resources and database resources, such as latches.

Configuring the Shared Pool

This section describes how to configure the shared pool and contains the following topics:

Sizing the Shared Pool

When configuring a new database instance, it is difficult to know the correct size for the shared pool cache. Typically, a DBA makes a first estimate for the cache size, then runs a representative workload on the instance, and examines the relevant statistics to see whether the cache is under-configured or over-configured.

For most OLTP applications, shared pool size is an important factor in application performance. Shared pool size is less important for applications that issue a very limited number of discrete SQL statements, such as decision support systems (DSS).

If the shared pool is too small, then extra resources are used to manage the limited amount of available space. This consumes CPU and latching resources, and causes contention. Ideally, the shared pool should be just large enough to cache frequently-accessed objects. Having a significant amount of free memory in the shared pool is a waste of memory. When examining the statistics after the database has been running, ensure that none of these mistakes are present in the workload.

This section describes how to size the shared pool and contains the following topics:

Using Library Cache Statistics

When sizing the shared pool, the goal is to cache SQL statements that are executed multiple times in the library cache without over-allocating memory. To accomplish this goal, examine the following library cache statistics:

- `RELOADS`

  The `RELOADS` column in the `V$LIBRARYCACHE` view shows the amount of reloading (or reparsing) of a previously-cached SQL statement that aged out of the cache. If the application reuses SQL effectively and runs on a system with an optimal shared pool size, this statistic should have a value near zero.

- `INVALIDATIONS`

  The `INVALIDATIONS` column in `V$LIBRARYCACHE` view shows the number of times library cache data was invalidated and had to be reparsed. This statistic should have a value near zero, especially on OLTP systems during peak loads. This means SQL statements that can be shared were invalidated by some operation (such as a DDL).

- Library cache hit ratio

  The library cache hit ratio is a broad indicator of the library cache health. This value should be considered along with the other statistics, such as the rate of hard parsing and if there is any shared pool or library cache latch contention.

- Amount of free memory in the shared pool

  To view the amount of free memory in the shared pool, query the `V$SGASTAT` performance view. Ideally, free memory should be as low as possible, without causing any reparsing on the system.

The following sections describe how to view and examine these library cache statistics:

- Using the V$LIBRARYCACHE View
- Calculating the Library Cache Hit Ratio
- Viewing the Amount of Free Memory in the Shared Pool

Using the V$LIBRARYCACHE View

Use the `V$LIBRARYCACHE` view to monitor statistics that reflect library cache activity. These statistics reflect all library cache activity after the most recent database instance startup.

Each row in this view contains statistics for one type of item kept in the library cache. The item described by each row is identified by the value of the `NAMESPACE` column. Rows with the following `NAMESPACE` values reflect library cache activity for SQL statements and PL/SQL blocks:

- `SQL AREA`
- `TABLE/PROCEDURE`
- `BODY`
- `TRIGGER`

Rows with other `NAMESPACE` values reflect library cache activity for object definitions that Oracle Database uses for dependency maintenance.

Example 15-1 shows a query of this view to examine each namespace individually.

Example 15-1 Querying the V$LIBRARYCACHE View

```
SELECT namespace, pins, pinhits, reloads, invalidations
  FROM V$LIBRARYCACHE
 ORDER BY namespace;
```

The output of this query might look like the following:

```
NAMESPACE             PINS    PINHITS    RELOADS INVALIDATIONS
--------------- ---------- ---------- ---------- -------------
BODY                  8870       8819          0             0
CLUSTER                393        380          0             0
INDEX                   29          0          0             0
OBJECT                   0          0          0             0
PIPE                 55265      55263          0             0
SQL AREA          21536413   21520516      11204             2
TABLE/PROCEDURE   10775684   10774401          0             0
TRIGGER               1852       1844          0             0
```

In this example, the output shows that:

- For the `SQL AREA` namespace, there are 21,536,413 executions.
- 11,204 of these executions resulted in a library cache miss, requiring the database to implicitly reparse a statement or block, or reload an object definition because it aged out of the library cache.
- SQL statements are invalidated twice, again causing library cache misses.

> **Note:**
>
> This query returns data from instance startup. Using statistics gathered over an interval instead may better identify the problem. For information about gathering information over an interval, see Automatic Performance Diagnostics .

> **See Also:**
>
> *Oracle Database Reference* for information about the `V$LIBRARYCACHE` view

Calculating the Library Cache Hit Ratio

To calculate the library cache hit ratio, use the following formula:

```
Library Cache Hit Ratio = sum(pinhits) / sum(pins)
```

Applying the library cache hit ratio formula to Example 15-1 results in the following library cache hit ratio:

```
SUM(PINHITS)/SUM(PINS)
----------------------
            .999466248
```

In this example, the hit percentage is about 99.94%, which means that only .06% of executions resulted in reparsing.

Viewing the Amount of Free Memory in the Shared Pool

The amount of free memory in the shared pool is reported in the `V$SGASTAT` view.

Example 15-2 shows a query of this view.

Example 15-2 Querying the V$SGASTAT View

```
SELECT *
  FROM V$SGASTAT
 WHERE name = 'free memory'
   AND pool = 'shared pool';
```

The output of this query might look like the following:

```
POOL        NAME                            BYTES
----------- -------------------------- ----------
shared pool free memory                   4928280
```

If free memory is always available in the shared pool, then increasing its size offers little or no benefit. Yet, just because the shared pool is full does not necessarily mean there is a problem. It may be indicative of a well-configured system.

Using Shared Pool Advisory Statistics

The amount of memory available for the library cache can drastically affect the parse rate of Oracle Database. To help you correctly size the library cache, Oracle Database provides the following shared pool advisory views:

- V$SHARED_POOL_ADVICE
- V$LIBRARY_CACHE_MEMORY
- V$JAVA_POOL_ADVICE
- V$JAVA_LIBRARY_CACHE_MEMORY

These shared pool advisory views provide information about library cache memory, enabling you to predict how changing the size of the shared pool can affect aging out of objects in the shared pool. The shared pool advisory statistics in these views track the library cache's use of shared pool memory and predict how the library cache will behave in shared pools of different sizes. Using these views enable you to determine:

- How much memory the library cache is using
- How much memory is currently pinned
- How much memory is on the shared pool's Least Recently Used (LRU) list
- How much time might be lost or gained by changing the size of the shared pool

These views display shared pool advisory statistics when the shared pool advisory is enabled. The statistics reset when the advisory is disabled.

The following sections describe these views in more detail:

- About the V$SHARED_POOL_ADVICE View
- About the V$LIBRARY_CACHE_MEMORY View
- About V$JAVA_POOL_ADVICE and V$JAVA_LIBRARY_CACHE_MEMORY Views

About the V$SHARED_POOL_ADVICE View

The V$SHARED_POOL_ADVICE view displays information about estimated parse time in the shared pool for different pool sizes. The sizes range from 10% of the current shared pool size or the amount of pinned library cache memory—whichever is higher—to 200% of the current shared pool size, in equal intervals. The value of the interval depends on the current size of the shared pool.

> **See Also:**
>
> *Oracle Database Reference* for more information about the V$SHARED_POOL_ADVICE view

About the V$LIBRARY_CACHE_MEMORY View

The V$LIBRARY_CACHE_MEMORY view displays information about memory allocated to library cache memory objects in different namespaces. A memory object is an internal grouping of memory for efficient management. A library cache object may consist of one or more memory objects.

> **See Also:**
>
> *Oracle Database Reference* for more information about the `V$LIBRARY_CACHE_MEMORY` view

About V$JAVA_POOL_ADVICE and V$JAVA_LIBRARY_CACHE_MEMORY Views

The `V$JAVA_POOL_ADVICE` and `V$JAVA_LIBRARY_CACHE_MEMORY` views contain Java pool advisory statistics that track information about library cache memory used for Java and predict how changing the size of the Java pool affects the parse rate.

The `V$JAVA_POOL_ADVICE` view displays information about estimated parse time in the Java pool for different pool sizes. The sizes range from 10% of the current Java pool size or the amount of pinned Java library cache memory—whichever is higher—to 200% of the current Java pool size, in equal intervals. The value of the interval depends on the current size of the Java pool.

> **See Also:**
>
> *Oracle Database Reference* for more information about the "V$JAVA_POOL_ADVICE" and "V$JAVA_LIBRARY_CACHE_MEMORY" views

Using Dictionary Cache Statistics

Typically, if the shared pool is adequately sized for the library cache, it will also be adequate sized for the dictionary cache data.

Misses on the data dictionary cache are to be expected in some cases. When the database instance starts up, the data dictionary cache does not contain any data. Therefore, any SQL statement issued is likely to result in cache misses. As more data is read into the cache, the likelihood of cache misses decreases. Eventually, the database reaches a steady state, in which the most frequently-used dictionary data is in the cache. At this point, very few cache misses occur.

Each row in the `V$ROWCACHE` view contains statistics for a single type of data dictionary item. These statistics reflect all data dictionary activity since the most recent instance startup.

Table 15-1 lists the columns in the `V$ROWCACHE` view that reflect the use and effectiveness of the data dictionary cache.

Table 15-1 Data Dictionary Columns in the V$ROWCACHE View

Column	Description
`PARAMETER`	Identifies a particular data dictionary item. For each row, the value in this column is the item prefixed by `dc_`. For example, in the row that contains statistics for file descriptions, this column contains the value `dc_files`.

Table 15-1 (Cont.) Data Dictionary Columns in the V$ROWCACHE View

Column	Description
GETS	Shows the total number of requests for information about the corresponding item. For example, in the row that contains statistics for file descriptions, this column contains the total number of requests for file description data.
GETMISSES	Shows the number of data requests that are not satisfied by the cache and required an I/O.
MODIFICATIONS	Shows the number of times data in the dictionary cache was updated.

Example 15-3 shows a query of this view to monitor the statistics over a period while the application is running. The derived column PCT_SUCC_GETS can be considered as the item-specific hit ratio.

Example 15-3 Querying the V$ROWCACHE View

```
column parameter format a21
column pct_succ_gets format 999.9
column updates format 999,999,999

SELECT parameter,
       sum(gets),
       sum(getmisses),
       100*sum(gets - getmisses) / sum(gets) pct_succ_gets,
       sum(modifications) updates
  FROM V$ROWCACHE
 WHERE gets > 0
 GROUP BY parameter;
```

The output of this query might look like the following:

```
PARAMETER              SUM(GETS) SUM(GETMISSES) PCT_SUCC_GETS      UPDATES
---------------------- ---------- -------------- ------------- ------------
dc_database_links              81              1          98.8            0
dc_free_extents             44876          20301          54.8       40,453
dc_global_oids                 42              9          78.6            0
dc_histogram_defs            9419            651          93.1            0
dc_object_ids               29854            239          99.2           52
dc_objects                  33600            590          98.2           53
dc_profiles                 19001              1         100.0            0
dc_rollback_segments        47244             16         100.0           19
dc_segments                100467          19042          81.0       40,272
dc_sequence_grants            119             16          86.6            0
dc_sequences                26973             16          99.9       26,811
dc_synonyms                  6617            168          97.5            0
dc_tablespace_quotas          120              7          94.2           51
dc_tablespaces             581248             10         100.0            0
dc_used_extents             51418          20249          60.6       42,811
dc_user_grants              76082             18         100.0            0
dc_usernames               216860             12         100.0            0
dc_users                   376895             22         100.0            0
```

In this example, the output shows the following:

- There are large numbers of misses and updates for used extents, free extents, and segments. This implies that the database instance had a significant amount of dynamic space extension.

- Comparing the percentage of successful gets with the actual number of gets indicates the shared pool is large enough to adequately store dictionary cache data.

You can also calculate the overall dictionary cache hit ratio using the following query; however, summing up the data over all the caches will lose the finer granularity of data:

```
SELECT (SUM(gets - getmisses - fixed)) / SUM(gets) "row cache"
  FROM V$ROWCACHE;
```

Increasing Memory Allocated to the Shared Pool

Increasing the amount of memory for the shared pool increases the amount of memory available to the library cache, the dictionary cache, and the result cache. Before doing so, review the shared pool statistics and examine:

- If the value of the `V$LIBRARYCACHE.RELOADS` column is near zero
- If the ratio of total `V$ROWCACHE.GETMISSES` column to total `V$ROWCACHE.GETS` is less than 10% or 15% for frequently-accessed dictionary caches, depending on the application

If both of these conditions are met, then the shared pool is adequately sized and increasing its memory will likely not improve performance. On the other hand, if either of these conditions is not met, and the application is using the shared pool effectively, as described in "Using the Shared Pool", then consider increasing the memory of the shared pool.

To increase the size of the shared pool:

- Increase the value of the `SHARED_POOL_SIZE` initialization parameter until the conditions are met.

  The maximum value for this parameter depends on your operating system. This measure reduces implicit reparsing of SQL statements and PL/SQL blocks on execution.

IC - Need link to "Managing Server Result Cache Memory with Init Parameters"

Reducing Memory Allocated to the Shared Pool

If the value of the `V$LIBRARYCACHE.RELOADS` column is near zero, and there is a small amount of free memory in the shared pool, then the shared pool is adequately sized to store the most frequently-accessed data. If there are always significant amounts of free memory in the shared pool and you want to allocate this memory elsewhere, then consider reducing the shared pool size.

To decrease the size of the shared pool

- Reduce the value of the `SHARED_POOL_SIZE` initialization parameter, while ensuring that good performance is maintained.

Deallocating Cursors

If there are no library cache misses, then consider setting the value of the `CURSOR_SPACE_FOR_TIME` initialization parameter to `TRUE` to accelerate execution calls. This parameter specifies whether a cursor can be deallocated from the library cache to make room for a new SQL statement.

If the `CURSOR_SPACE_FOR_TIME` parameter is set to:

- `FALSE` (the default), then a cursor can be deallocated from the library cache regardless of whether application cursors associated with its SQL statement are open.

In this case, Oracle Database must verify that the cursor containing the SQL statement is in the library cache.

- `TRUE`, then a cursor can be deallocated only when all application cursors associated with its statement are closed.

  In this case, Oracle Database does not need to verify that a cursor is in the library cache because it cannot be deallocated while an application cursor associated with it is open.

Setting the value of the parameter to `TRUE` saves Oracle Database a small amount of time and may slightly improve the performance of execution calls. This value also prevents the deallocation of cursors until associated application cursors are closed.

Do not set the value of the `CURSOR_SPACE_FOR_TIME` parameter to `TRUE` if:

- Library cache misses are found in execution calls

  Library cache misses indicate that the shared pool is not large enough to hold the shared SQL areas of all concurrently open cursors. If the shared pool does not have enough space for a new SQL statement and the value for this parameter is set to `TRUE`, then the statement cannot be parsed and Oracle Database returns an error indicating that there is not enough shared memory.

- The amount of memory available to each user for private SQL areas is scarce

  This value also prevents the deallocation of private SQL areas associated with open cursors. If the private SQL areas for all concurrently open cursors fills the available memory so that there is no space for a new SQL statement, then the statement cannot be parsed and Oracle Database returns an error indicating that there is not enough memory.

If the shared pool does not have enough space for a new SQL statement and the value of this parameter is set to `FALSE`, then Oracle Database deallocates an existing cursor. Although deallocating a cursor may result in a subsequent library cache miss (if the cursor is re-executed), this is preferable to an error halting the application because a SQL statement cannot be parsed.

Caching Session Cursors

The session cursor cache contains closed session cursors for SQL and PL/SQL, including recursive SQL. This cache can be useful to applications that use Oracle Forms because switching from one form to another closes all session cursors associated with the first form. If an application repeatedly issues parse calls on the same set of SQL statements, then reopening session cursors can degrade performance. By reusing cursors, the database reduces parse times, leading to faster overall execution times.

This section describes the session cursor cache and contains the following topics:

- About the Session Cursor Cache
- Enabling the Session Cursor Cache
- Sizing the Session Cursor Cache

About the Session Cursor Cache

A session cursor represents an instantiation of a shared child cursor, which is stored in the shared pool, for a specific session. Each session cursor stores a reference to a child cursor that it has instantiated.

Oracle Database checks the library cache to determine whether more than three parse requests have been issued on a given statement. If a cursor has been closed three times, then

Oracle Database assumes that the session cursor associated with the statement should be cached and moves the cursor into the session cursor cache.

Subsequent requests to parse a SQL statement by the same session search an array for pointers to the shared cursor. If the pointer is found, then the database dereferences the pointer to determine whether the shared cursor exists. To reuse a cursor from the cache, the cache manager checks whether the cached states of the cursor match the current session and system environment.

> **Note:**
>
> Reuse of a cached cursor still registers as a parse, even though it is not a hard parse.

An LRU algorithm removes entries in the session cursor cache to make room for new entries when needed. The cache also uses an internal time-based algorithm to age out cursors that have been idle for an certain amount of time.

Enabling the Session Cursor Cache

The following initialization parameters pertain to the session cursor cache:

- `SESSION_CACHED_CURSORS`

  This parameter sets the maximum number of cached closed cursors for each session. The default value is 50. Use this parameter to reuse cursors from the cache for the statements that get executed repeatedly in the same session.

- `OPEN_CURSORS`

  This parameter specifies the maximum number of cursors a session can have open simultaneously. For example, if its value is set to 1000, then each session can have up to 1000 cursors open at one time.

These parameters are independent. For example, you can set the value of the `SESSION_CACHED_CURSORS` parameter higher than the value of the `OPEN_CURSORS` parameter because session cursors are not cached in an open state.

To enable the session cursor cache:

1. Determine the maximum number of session cursors to keep in the cache.
2. Do one of the following:
   - To enable static caching, set the value of the `SESSION_CACHED_CURSORS` parameter to the number determined in the previous step.
   - To enable dynamic caching, execute the following statement:

     ```
     ALTER SESSION SET SESSION_CACHED_CURSORS = value;
     ```

Sizing the Session Cursor Cache

Use the `V$SESSTAT` view to determine if the session cursor cache is adequately sized for the database instance.

To size the session cursor cache:

1. Query the `V$SESSTAT` view to determine how many cursors are currently cached in a particular session.
2. Query the `V$SESSTAT` view to find the percentage of parse calls that found a cursor in the session cursor cache.
3. Consider increasing the value of the `SESSION_CACHED_CURSORS` parameter if the following conditions are true:
   - The session cursor cache count is close to the maximum
   - The percentage of session cursor cache hits is low relative to the total parses
   - The application repeatedly performs parse calls for the same queries

Example 15-4 shows two queries of this view.

Example 15-4 Querying the V$SESSTAT View

The following query finds how many cursors are currently cached in a particular session:

```
SELECT a.value curr_cached, p.value max_cached,
       s.username, s.sid, s.serial#
  FROM v$sesstat a, v$statname b, v$session s, v$parameter2 p
 WHERE a.statistic# = b.statistic#  and s.sid=a.sid and a.sid=&sid
   AND p.name='session_cached_cursors'
   AND b.name = 'session cursor cache count';
```

The output of this query might look like the following:

```
CURR_CACHED MAX_CACHED USERNAME   SID    SERIAL#
----------- ---------- -------- ----- ----------
         49 50         APP         35        263
```

This output shows that the number of cursors currently cached for session 35 is close to the maximum.

The following query finds the percentage of parse calls that found a cursor in the session cursor cache:

```
SELECT cach.value cache_hits, prs.value all_parses,
       round((cach.value/prs.value)*100,2) as "% found in cache"
  FROM v$sesstat cach, v$sesstat prs, v$statname nm1, v$statname nm2
 WHERE cach.statistic# = nm1.statistic#
   AND nm1.name = 'session cursor cache hits'
   AND prs.statistic#=nm2.statistic#
   AND nm2.name= 'parse count (total)'
   AND cach.sid= &sid and prs.sid= cach.sid;
```

The output of this query might look like the following:

```
CACHE_HITS ALL_PARSES % found in cache
---------- ---------- ---------------
        34        700            4.57
```

This output shows that the number of hits in the session cursor cache for session 35 is low compared to the total number of parses.

In this example, setting the value of the `SESSION_CACHED_CURSORS` parameter to 100 may help boost performance.

Sharing Cursors

In the context of SQL parsing, an identical statement is a SQL statement whose text is identical to another statement, character for character, including spaces, case, and comments. A similar statement is identical except for the values of some literals.

The parse phase compares the statement text with statements in the shared pool to determine if the statement can be shared. If the value of the `CURSOR_SHARING` initialization parameter is set to `EXACT` (the default value), and if a statement in the shared pool is not identical, then the database does not share the SQL area. Instead, each SQL statement has its own parent cursor and its own execution plan based on the literal in the statement.

This section describes how cursors can be shared and contains the following topics:

- About Cursor Sharing
- Forcing Cursor Sharing

About Cursor Sharing

When SQL statements use literals rather than bind variables, setting the value of the `CURSOR_SHARING` initialization parameter to `FORCE` enables the database to replace literals with system-generated bind variables. Using this technique, the database may reduce the number of parent cursors in the shared SQL area.

When the value of the `CURSOR_SHARING` parameter is set to `FORCE`, the database performs the following steps during the parse phase:

1. Searches for an identical statement in the shared pool.

   If an identical statement is found, then the database skips the next step and proceeds to step 3. Otherwise, the database proceeds to the next step.

2. Searches for a similar statement in the shared pool.

   If a similar statement is *not* found, then the database performs a hard parse. If a similar statement *is* found, then the database proceeds to the next step.

3. Proceeds through the remaining steps of the parse phase to ensure that the execution plan of the existing statement is applicable to the new statement.

   If the plan is not applicable, then the database performs a hard parse. If the plan is applicable, then the database proceeds to the next step.

4. Shares the SQL area of the statement.

For details about the various checks performed by the database, see "SQL Sharing Criteria".

Forcing Cursor Sharing

The best practice is to write sharable SQL and use the default value of `EXACT` for the `CURSOR_SHARING` initialization parameter. By default, Oracle Database uses adaptive cursor sharing to enable a single SQL statement that contains bind variables to use multiple execution plans. However, for applications with many similar statements that use literals instead of bind variables, setting the value of the `CURSOR_SHARING` parameter to `FORCE` may improve cursor sharing, resulting in reduced memory usage, faster parses, and reduced latch contention. Consider this approach when statements in the shared pool differ only in the values of literals, and when response time is poor because of a high number of library cache misses. In this case, setting the value of the `CURSOR_SHARING` parameter to `FORCE` maximizes cursor sharing

and leverages adaptive cursor sharing to generate multiple execution plans based on different literal value ranges.

If stored outlines are generated with the value of the `CURSOR_SHARING` parameter set to `EXACT`, then the database does not use stored outlines generated with literals. To avoid this problem, generate outlines with `CURSOR_SHARING` set to `FORCE` and use the `CREATE_STORED_OUTLINES` parameter.

Setting the value of the `CURSOR_SHARING` parameter to `FORCE` has the following drawbacks:

- The database must perform extra work during the soft parse to find a similar statement in the shared pool.
- There is an increase in the maximum lengths (as returned by `DESCRIBE`) of any selected expressions that contain literals in a `SELECT` statement. However, the actual length of the data returned does not change.
- Star transformation is not supported.

When the value of the `CURSOR_SHARING` parameter is set to `FORCE`, the database uses one parent cursor and one child cursor for each distinct SQL statement. The same plan is used for each execution of the same statement. For example, consider the following SQL statement:

```
SELECT *
  FROM hr.employees
 WHERE employee_id = 101;
```

If the value of the `CURSOR_SHARING` parameter is set to `FORCE`, then the database optimizes this statement as if it contained a bind variable and uses bind peeking to estimate cardinality.

> **Note:**
>
> Starting with Oracle Database 11*g* Release 2, setting the value of the `CURSOR_SHARING` parameter to `SIMILAR` is obsolete. Consider using adaptive cursor sharing instead.

> **See Also:**
>
> - *Oracle Database Reference* for information about the `CURSOR_SHARING` initialization parameter
> - *Oracle Database SQL Tuning Guide* for information about adaptive cursor sharing

Keeping Large Objects to Prevent Aging

After an entry is loaded into the shared pool, it cannot be moved. Sometimes, as entries are loaded and aged out, the free memory may become fragmented. Shared SQL and PL/SQL areas age out of the shared pool according to an LRU algorithm that is similar to database buffers. To improve performance and avoid reparsing, prevent large SQL or PL/SQL areas from aging out of the shared pool.

The `DBMS_SHARED_POOL` package enables you to keep objects in shared memory, so that they do not age out with the normal LRU mechanism. By using the `DBMS_SHARED_POOL` package to

load the SQL and PL/SQL areas before memory fragmentation occurs, the objects can be kept in memory. This ensures that memory is available and prevents the sudden slowdowns in user response times that occur when SQL and PL/SQL areas are accessed after being aged out.

Consider using the `DBMS_SHARED_POOL` package:

- When loading large PL/SQL objects, such as the `STANDARD` and `DIUTIL` packages.

  When large PL/SQL objects are loaded, user response time may be affected if smaller objects must age out of the shared pool to make room for the larger objects. In some cases, there may be insufficient memory to load the large objects.
- To keep compiled triggers on frequently used tables in the shared pool.
- Support sequences.

  Sequence numbers are lost when a sequence ages out of the shared pool. The `DBMS_SHARED_POOL` package keeps sequences in the shared pool, thus preventing the loss of sequence numbers.

To keep a SQL or PL/SQL area in shared memory:

1. Decide which packages or cursors to keep in memory.
2. Start up the database.
3. Call the `DBMS_SHARED_POOL.KEEP` package to pin the objects.

   This procedure ensures that the system does not run out of shared memory before the pinned objects are loaded. Pinning the objects early in the life of the database instance prevents memory fragmentation that may result from keeping a large portion of memory in the middle of the shared pool.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information about using `DBMS_SHARED_POOL` procedures

Configuring the Reserved Pool

Although Oracle Database breaks down very large requests for memory into smaller chunks, on some systems there may be a requirement to find a contiguous chunk of memory (such as over 5 KB, the default minimum reserved pool allocation is 4,400 bytes).

If there is not enough free space in the shared pool, then Oracle Database must search for and free enough memory to satisfy this request. This operation may hold the latch resource for significant periods of time, causing minor disruption to other concurrent attempts at memory allocation.

To avoid this, Oracle Database internally reserves a small memory area in the shared pool by default that the database can use if the shared pool does not have enough space. This reserved pool makes allocation of large chunks more efficient. The database can use this memory for operations such as PL/SQL and trigger compilation, or for temporary space while loading Java objects. After the memory allocated from the reserved pool is freed, it is returned to the reserved pool.

For large allocations, Oracle Database attempts to allocate space in the shared pool in the following order:

1. From the unreserved part of the shared pool.
2. From the reserved pool.

   If there is not enough space in the unreserved part of the shared pool, then Oracle Database checks whether the reserved pool has enough space.
3. From memory.

   If there is not enough space in the unreserved and reserved parts of the shared pool, then Oracle Database attempts to free enough memory for the allocation. The database then retries the unreserved and reserved parts of the shared pool.

This section describes how to configure the reserved pool and contains the following topics:

- Sizing the Reserved Pool
- Increasing Memory Allocated to the Reserved Pool
- Reducing Memory Allocated to the Reserved Pool

Sizing the Reserved Pool

Typically, it is not necessary to change the default amount of space Oracle Database reserves for the reserved pool. However, there may be cases where you need to set aside space in the shared pool for unusually large allocations of memory.

You can set the reserved pool size by setting the value of the `SHARED_POOL_RESERVED_SIZE` initialization parameter. The default value for the `SHARED_POOL_RESERVED_SIZE` parameter is 5% of the `SHARED_POOL_SIZE` parameter.

If you set the value of the `SHARED_POOL_RESERVED_SIZE` parameter to more than half of the `SHARED_POOL_SIZE` parameter, then Oracle Database returns an error because the database does not allow you to reserve too much memory for the reserved pool. The amount of operating system memory available may also constrain the size of the shared pool. In general, set the value of the `SHARED_POOL_RESERVED_SIZE` parameter to no higher than 10% of the `SHARED_POOL_SIZE` parameter. On most systems, this value is sufficient if the shared pool is adequately tuned. If you increase this value, then the database takes additional memory from the shared pool and reduces the amount of unreserved shared pool memory available for smaller allocations.

When tuning these parameters, use statistics from the `V$SHARED_POOL_RESERVED` view. On a system with ample free memory to increase the size of the SGA, the value of the `REQUEST_MISSES` statistic should equal zero. If the system is constrained by operating system memory, then the goal is to have the `REQUEST_FAILURES` statistic equal zero, or at least prevent its value from increasing. If you cannot achieve these target values, then increase the value of the `SHARED_POOL_RESERVED_SIZE` parameter. Also, increase the value of the `SHARED_POOL_SIZE` parameter by the same amount, because the reserved list is taken from the shared pool.

The `V$SHARED_POOL_RESERVED` fixed view can also indicate when the value of the `SHARED_POOL_SIZE` parameter is too small. This can be the case if the `REQUEST_FAILURES` statistic is greater than zero and increasing. If the reserved list is enabled, then decrease the value of the `SHARED_POOL_RESERVED_SIZE` parameter. If the reserved list is not enabled, then increase the value of the `SHARED_POOL_SIZE` parameter, as described in "Increasing Memory Allocated to the Shared Pool".

Increasing Memory Allocated to the Reserved Pool

The reserved pool is too small if the value of the `REQUEST_FAILURES` statistic is higher than zero and increasing. In this case, increase the amount of memory available to the reserved pool.

> **Note:**
>
> Increasing the amount of memory available on the reserved list does not affect users who do not allocate memory from the reserved list.

To increase the size of the reserved pool:

- Increase the value of the `SHARED_POOL_RESERVED_SIZE` and `SHARED_POOL_SIZE` initialization parameters accordingly.

  The values that you select for these parameters depend on the system's SGA size constraints, as described in "Sizing the Reserved Pool".

Reducing Memory Allocated to the Reserved Pool

The reserved pool is too large if the:

- `REQUEST_MISSES` statistic is zero or not increasing
- `FREE_SPACE` statistic is greater than or equal to 50% of the `SHARED_POOL_RESERVED_SIZE` minimum

If either of these conditions is true, then reduce the amount of memory available to the reserved pool.

To reduce the size of the reserved pool:

- Decrease the value of the `SHARED_POOL_RESERVED_SIZE` initialization parameter.

Configuring the Large Pool

Unlike the shared pool, the large pool does not have an LRU list. Oracle Database does not attempt to age objects out of the large pool. Consider configuring a large pool if the database instance uses any of the following Oracle Database features:

- Shared server

  In a shared server architecture, the session memory for each client process is included in the shared pool.

- Parallel query

  Parallel query uses shared pool memory to cache parallel execution message buffers.

- Recovery Manager

  Recovery Manager (RMAN) uses the shared pool to cache I/O buffers during backup and restore operations. For I/O server processes, backup, and restore operations, Oracle Database allocates buffers that are a few hundred kilobytes in size.

This section describes how to configure the large pool for the shared server architecture and contains the following topics:

- Configuring the Large Pool for Shared Server Architecture
- Configuring the Large Pool for Parallel Query
- Sizing the Large Pool
- Limiting Memory Use for User Sessions

- Reducing Memory Use Using Three-Tier Connections

> **See Also:**
>
> - *Oracle Database Concepts* for information about the large pool
> - *Oracle Database Backup and Recovery User's Guide* for information about sizing the large pool with Recovery Manager

Configuring the Large Pool for Shared Server Architecture

As Oracle Database allocates shared pool memory to shared server sessions, the amount of shared pool memory available for the library cache and data dictionary cache decreases. If you allocate the shared server session memory from a different pool, then the shared pool can be reserved for caching shared SQL.

Oracle recommends using the large pool to allocate the User Global Area (UGA) for the shared server architecture. Using the large pool instead of the shared pool decreases fragmentation of the shared pool and eliminates the performance overhead from shrinking the shared SQL cache.

By default, the large pool is not configured. If you do not configure the large pool, then Oracle Database uses the shared pool for shared server user session memory. If you do configure the large pool, Oracle Database still allocates a fixed amount of memory (about 10K) for each configured session from the shared pool when a shared server architecture is used. In either case, consider increasing the size of the shared pool accordingly.

> **Note:**
>
> Even though use of shared memory increases with shared servers, the total amount of memory use decreases. This is because there are fewer processes; therefore, Oracle Database uses less PGA memory with shared servers when compared to dedicated server environments.

> **Tip:**
>
> To specify the maximum number of concurrent shared server sessions that the database allows, use the `CIRCUITS` initialization parameter.

> **Tip:**
>
> For best performance with sort operations using shared servers, set the values of the `SORT_AREA_SIZE` and `SORT_AREA_RETAINED_SIZE` initialization parameters to the same value. This keeps the sort result in the large pool instead of writing it to disk.

Configuring the Large Pool for Parallel Query

Parallel query uses shared pool memory to cache parallel execution message buffers when Automatic Memory Management or Automatic Shared Memory Management is not enabled. Caching parallel execution message buffers in the shared pool increases its workload and may cause fragmentation.

To avoid possible negative impact to performance, Oracle recommends that you do not manage SGA memory manually when parallel query is used. Instead, you should enable Automatic Memory Management or Automatic Shared Memory Management to ensure that the large pool will be used to cache parallel execution memory buffers.

See Also:

- "Automatic Memory Management"
- "Automatic Shared Memory Management"
- *Oracle Database VLDB and Partitioning Guide*

Sizing the Large Pool

When storing shared server-related UGA in the large pool, the exact amount of UGA that Oracle Database uses depends on the application. Each application requires a different amount of memory for session information, and configuration of the large pool should reflect the memory requirement.

Oracle Database collects statistics on memory used by a session and stores them in the `V$SESSTAT` view. Table 15-2 lists the statistics from this view that reflect session UGA memory.

Table 15-2 Memory Statistics in the V$SESSTAT View

Statistic	Description
`session UGA memory`	Shows the amount of memory in bytes allocated to the session.
`session UGA memory max`	Shows the maximum amount of memory in bytes ever allocated to the session.

There are two methods to use this view to determine a correct size for the large pool. One method is to configure the size of the large pool based on the number of simultaneously active sessions. To do this, observe UGA memory usage for a typical user and multiply this amount by the estimated number of user sessions. For example, if the shared server requires 200K to 300K to store session information for a typical user session and you anticipate 100 active user sessions simultaneously, then configure the large pool to 30 MB.

Another method is to calculate the total and maximum memory being used by all user sessions. Example 15-5 shows two queries of the `V$SESSTAT` and `V$STATNAME` views to do this.

Example 15-5 Querying the V$SESSTAT and V$STATNAME Views

While the application is running, issue the following queries:

```
SELECT SUM(value) || ' bytes' "total memory for all sessions"
  FROM V$SESSTAT, V$STATNAME
```

```
 WHERE name = 'session uga memory'
   AND V$SESSTAT.STATISTIC# = V$STATNAME.STATISTIC#;

SELECT SUM(value) || ' bytes' "total max mem for all sessions"
  FROM V$SESSTAT, V$STATNAME
 WHERE name = 'session uga memory max'
   AND V$SESSTAT.STATISTIC# = V$STATNAME.STATISTIC#;
```

These queries also select from the `V$STATNAME` view to obtain internal identifiers for `session memory` and `max session memory`.

The output of these queries might look like the following:

```
TOTAL MEMORY FOR ALL SESSIONS
-----------------------------
157125 BYTES

TOTAL MAX MEM FOR ALL SESSIONS
------------------------------
417381 BYTES
```

The result of the first query shows that the memory currently allocated to all sessions is 157,125 bytes. This value is the total memory with a location that depends on how the sessions are connected to the database. If the sessions are connected to dedicated server processes, then this memory is part of the memories of the user processes. If the sessions are connected to shared server processes, then this memory is part of the shared pool.

The result of the second query shows that the sum of the maximum size of the memory for all sessions is 417,381 bytes. The second result is greater than the first because some sessions have deallocated memory since allocating their maximum amounts.

Use the result of either queries to determine the correct size for the shared pool. The first value is likely to be a better estimate than the second, unless if you expect all sessions to reach their maximum allocations simultaneously.

To size the large pool:

1. Verify the pool (shared pool or large pool) in which the memory for an object resides by checking the `POOL` column in the `V$SGASTAT` view.
2. Set a value for the `LARGE_POOL_SIZE` initialization parameter.

   The minimum value for this parameter is 300K.

Limiting Memory Use for User Sessions

To restrict the memory used by each client session from the SGA, set a resource limit using `PRIVATE_SGA`.

`PRIVATE_SGA` defines the number of bytes of memory used from the SGA by a session. However, this parameter is rarely used, because most DBAs do not limit SGA consumption on a user-by-user basis.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about setting the `PRIVATE_SGA` resource limit

Reducing Memory Use Using Three-Tier Connections

If there is a high number of connected users, then consider reducing memory usage by implementing three-tier connections. Using a transaction process (TP) monitor is feasible only with pure transactional models because locks and uncommitted DML operations cannot be held between calls.

Using a shared server environment:

- Is much less restrictive of the application design than a TP monitor.
- Dramatically reduces operating system process count and context switches by enabling users to share a pool of servers.
- Substantially reduces overall memory usage, even though more SGA is used in shared server mode.