CHAPTER 4

The Memory Structures

Now we are ready to look at Oracle's major memory structures. There are three of them to be considered:

* *SGA, System Global Area* – This is a large, shared memory segment that virtually all Oracle processes will access at one point or another.
* *PGA, Process Global Area* – This is memory that is private to a single process or thread, and is not accessible from other processes/threads.
* *UGA, User Global Area* – This is memory associated with your session. It will be found either in the SGA or the PGA depending on whether you are connected to the database using shared server (then it'll be in the SGA), or dedicated server (it'll be in the PGA, in the process memory).

We will briefly discuss the PGA and UGA, and then move onto the really big structure, the SGA.

The PGA and UGA

The PGA is a process piece of memory. In other words, it is memory specific to a single operating system process or thread. This memory is not accessible by any other process/thread in the system. It is typically allocated via either of the C run-time calls, malloc() or memmap(), and may grow (and shrink even) at run-time. The PGA is never allocated in Oracle's SGA – it is always allocated locally by the process or thread.

The UGA is, in effect, your session's state. It is memory that your session must always be able to get to. The location of the UGA is wholly dependent on how you connected to Oracle. If you have connected via a shared server, then the UGA must be stored in a memory structure that everyone has access to – and that would be the SGA. In this way, your session can use any one of the shared servers, since any one of them can read and write your sessions data. On the other hand, if you are using a dedicated server connection, this need for universal access to your session state goes away, and the UGA becomes virtually synonymous with the PGA – it will in fact be contained in the PGA of your dedicated server. When you look at the system statistics, you'll find the UGA reported in the PGA in dedicated server mode (the PGA will be greater than, or equal to, the UGA memory used; the PGA memory size will include the UGA size as well).

So, the PGA contains process memory and may include the UGA. The other areas of PGA memory are generally used for in-memory sorting and hashing. It would be safe to say that, besides the UGA memory, these are the largest contributors by far to the PGA.

Starting with Oracle 9i Release 1 and above, there are two ways to manage this other non-UGA memory in the PGA

* **Manual PGA memory management**, where you tell Oracle how much ram is it allowed to use to sort and hash anytime it needs to sort or hash
* **Automatic PGA memory management**, whereby you tell Oracle how much memory *system wide* it should attempt to use.

The manner in which memory is allocated and used differs greatly in each case and, as such, we'll discuss each in turn. It should be noted that in Oracle 9i, when using a shared server connection, you can *only* utilize manual PGA memory management. This restriction was lifted with Oracle 10g R1 and above – in that release, you can use either automatic or manual PGA memory management with shared server connections.

PGA memory management is controlled by the database initialization parameter WORKAREA_SIZE_POLICY and may be altered at the session level. This initialization parameter defaults to AUTO, for automatic PGA memory management when possible.

Now, we'll take a look at both approaches.

Manual PGA Memory Management

In manual PGA memory management, the parameters that will have the largest impact on the size of your PGA, outside of the UGA, will be:

* SORT_AREA_SIZE: The total amount of RAM that will be used to sort information before swapping out to disk.
* SORT_AREA_RETAINED_SIZE: The amount of memory that will be used to hold sorted data after the sort is complete. That if, if SORT_AREA_SIZE was 512k and SORT_AREA_RETAINED_SIZE was 256k, then your server process would use up to 512k of memory to sort data during the initial processing of the query. When the sort was complete, the sorting area would be "shrunk" down to 256k and any sorted data that did not fit in that 256k would be written out to temporary segments.
* HASH_AREA_SIZE: The amount of memory your server process would use to store hash tables in memory. These structures are used during a hash join, typically when joining a large set with another set. The smaller of the two sets would be hashed into memory (hopefully, anything that doesn't fit in the hash area region of memory would be stored in the temporary tablespace) by the join key. .

These parameters control the amount of space Oracle will use to sort data before writing (swapping) it to disk, and how much of that memory segment will be retained after the sort is done. The SORT_AREA_SIZE is generally allocated out of your PGA and the SORT_AREA_RETAINED_SIZE will be in your UGA. We can discover our current usage of PGA and UGA memory and monitor its size by querying special Oracle V$ tables, also referred to as a dynamic performance table.

NOTE We'll find out more about these V$ tables in Chapter X, *Tuning Strategies and Tools*.

For example, I'll run a small test whereby in one session we will sort lots of data and, from a second session, we monitor the UGA/PGA memory usage in that first session. In order to do this in a predicable manner, we'll make a copy of the ALL_OBJECTS table without any indexes (so we know a sort has to happen):

```
ops$tkyte@ORA10G> drop table t;
Table dropped.

ops$tkyte@ORA10G> create table t as select * from all_objects;
Table created.

ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T' );
PL/SQL procedure successfully completed.
```

Now, in order to remove any side effects from the initial hard parsing of queries we will run the following script, but for now ignore it's output. We will be running it again, in a fresh session so as to see the effects on memory usage in a controlled environment. We will use the sort area sizes of 64 KB, 1 MB, and 1 GB in turn:

```
alter session set sort_area_size = 65536;
set autotrace traceonly statistics;
select * from t order by 1, 2, 3, 4;
set autotrace off
alter session set sort_area_size=1048576;
set autotrace traceonly statistics;
select * from t order by 1, 2, 3, 4;
set autotrace off
alter session set sort_area_size=1048576000;
set autotrace traceonly statistics;
select * from t order by 1, 2, 3, 4;
set autotrace off
```

NOTE: We will discuss parsing in detail in Chapter X, Tuning Strategies and Tools. When we process SQL in the database, we must first "parse" the SQL statement. There are two types of parses available, a 'hard' parse which is what happens the first time a query is parsed by the database instance and includes query plan generation and optimization. There is also a soft parse, which can skip many of the steps a hard parse must do. We are hard parsing the queries above so as to not measure the work performed by that operation in the following section.

Now, I would suggest logging out of that SQL*Plus session and logging back in before continuing, in order to get a consistent environment, one in which no work has been done yet. In order to ensure we are using manual memory management, we'll set it on specifically and specify our rather small sort area size of 64k, also, we'll identify our SID (session id) so we can monitor the memory usage for that session:

```
ops$tkyte@ORA10G> alter session set workarea_size_policy=manual;
Session altered.

ops$tkyte@ORA10G> select  sid from v$mystat where rownum = 1;

       SID
----------
       151
```

Now, we need to measure SID 151's memory from a second separate session. If we used the same session then our query to see how much memory we are using for sorting might itself influence the very numbers we are looking at! To measure the memory from this second session, I will be using a small SQL*Plus script I developed for this. It is actually a pair of scripts, one to reset a small table and set a SQL*Plus variable to the SID we want to watch, I call this reset_stat.sql:

```
drop table sess_stats;

create table sess_stats
( name varchar2(64), value number, diff number );

variable sid number
exec :sid := &1
```

NOTE: Before using this or in fact any script, make sure you understand what it does. I am dropping and recreating a table called SESS_STATS. If your schema already has such a table, well, you would probably want to use a different name!

The other script I call watch_stat.sql, and for this case study, it looks like this:

```
merge into sess_stats
using
(
select a.name, b.value
  from v$statname a, v$sesstat b
 where a.statistic# = b.statistic#
   and b.sid = :sid
   and (a.name like '%ga %'
        or a.name like '%direct temp%')
) curr_stats
on (sess_stats.name = curr_stats.name)
when matched then
  update set diff = curr_stats.value - sess_stats.value,
             value = curr_stats.value
when not matched then
  insert ( name, value, diff )
  values
  ( curr_stats.name, curr_stats.value, null )
/

select *
  from sess_stats
 order by name;
```

I said it looks like this for this case study, because of the line in bold, the names of the statistics I'm interested in looking at change from example to example. In this particular case we are interested in anything with 'ga ' in it (pga and uga) or anything with 'direct temp' which in Oracle 10g will show us the direct reads and writes against temporary space (how much IO we did reading and writing to temp).

NOTE: In Oracle 9i, direct IO to temporary space was not labeled as such, you would use a where clause that included `and (a.name like '%ga %'      or a.name like '%physical % direct%')` in it.

When that is run from the SQL*Plus command line , you'll see a listing of the PGA and UGA memory statistics for that session as well as temporary IO. Before we do anything in session 154, using manual PGA memory management, let's use this script to find out how much memory we are currently using and how many temporary IO's we have performed:

```
ops$tkyte@ORA10G> @watch_stat
6 rows merged.


NAME                                          VALUE       DIFF
------------------------------------------ ---------- ----------
physical reads direct temporary tablespace           0
physical writes direct temporary tablespace          0
session pga memory                              498252
session pga memory max                          498252
session uga memory                              152176
session uga memory max                          152176
```

So, before we begin we can see that we have about 149 KB (152176/1024) of data in the UGA and 487 KB of data in the PGA. The first question is: how much memory are we using between the PGA and UGA, that is – are we using 149+487 Kb of memory or some other amount? It is a trick question, and one that you cannot answer unless you know if you are connected via a dedicated server or a shared server, and even then it might be hard to figure out. In dedicated server mode, the UGA is totally contained within the PGA, in which case we would be consuming 487 KB of memory in our process or thread. In shared server, the UGA is allocated from the SGA, and the PGA is in the shared server. So, in shared server mode, by the time we get the last row from the above query, our process may be in use by someone else. That PGA isn't 'ours' anymore, so technically we are using 149 KB of memory (except when we are actually running the query at which point we are using 487 KB of memory between the combined PGA and UGA). So, let's now run the first big query in session 151, which is using manual PGA memory management, note that we are using the same script from above, so the SQL text matches exactly, thus avoiding the hard parse:

```
ops$tkyte@ORA10G> alter session set sort_area_size = 65536;
Session altered.

ops$tkyte@ORA10G> set autotrace traceonly statistics;
ops$tkyte@ORA10G> select * from t order by 1, 2, 3, 4;

48353 rows selected.


Statistics
----------------------------------------------------------
          8  recursive calls
        237  db block gets
        671  consistent gets
       2859  physical reads
          0  redo size
    3461588  bytes sent via SQL*Net to client
      35961  bytes received via SQL*Net from client
       3225  SQL*Net roundtrips to/from client
          0  sorts (memory)
          1  sorts (disk)
      48353  rows processed
ops$tkyte@ORA10G> set autotrace off
```

Now, if we run our script again in the second session, we'll see something like this. Notice this time that the session xxx memory and session xxx memory max values do not match. The "session xxx memory" value represents how much we are using right now. The max value represents the peak value we used at some time during our session, while processing the query:

```
ops$tkyte@ORA10G> @watch_stat
6 rows merged.

NAME                                           VALUE       DIFF
---------------------------------------- ---------- ----------
physical reads direct temporary tablespace       2859       2859
physical writes direct temporary tablespace      2859       2859
session pga memory                             563788      65536
session pga memory max                         694860     196608
session uga memory                             152176          0
session uga memory max                         283104     130928

6 rows selected.
```

As you can see, our memory usage went up – we've done some sorting of data. Our UGA temporarily increased to from 149 KB to 276 KB (about 64*2 KB) during the processing of our query, and then shrunk back down. In order to perform our query and the sorting, Oracle allocated some sort areas for our session. Additionally, the PGA memory when from 487 KB to 679 KB, a jump of 3*64 KB, apparently we had multiple sort areas in use – perhaps as a side effect of the system recursive SQL that took place. Also, we can see that we did 2,859 writes and reads to and from temp – all of the physical IO reported by the autotrace report was actually due to temporary space reads (and autotrace does not report physical write IO).

NOTE: System Recursive SQL is that SQL Oracle executes in order to process our queries, it could be for space allocation, parsing, temporary space management, quota checking and so on.

You can see from the Autotrace statistics, that Oracle has reported that we've done 8 recursive calls of some sort during the processing of our query. By the time we finished our query and exhausted the result set, we can see our UGA memory went back down to what it started at (we released the sort areas from our UGA) and the PGA shrunk back somewhat (note that in Oracle 8i and before, you would not expect to see the PGA shrink back at all, this is new with Oracle 9i and later). Now, let's retry that operation but play around with the size of our SORT_AREA increasing it to 1 MB:

```
ops$tkyte@ORA10G> alter session set sort_area_size=1048576;
Session altered.

ops$tkyte@ORA10G> set autotrace traceonly statistics;
ops$tkyte@ORA10G> select * from t order by 1, 2, 3, 4;
48353 rows selected.

Statistics
----------------------------------------------------------
          6  recursive calls
          8  db block gets
        671  consistent gets
        676  physical reads
          0  redo size
    3461588  bytes sent via SQL*Net to client
      35961  bytes received via SQL*Net from client
       3225  SQL*Net roundtrips to/from client
          0  sorts (memory)
          1  sorts (disk)
      48353  rows processed

ops$tkyte@ORA10G> set autotrace off
```

Now, in the other session we can measure our memory usage again:

```
ops$tkyte@ORA10G> @watch_stat
6 rows merged.

NAME                                          VALUE       DIFF
-------------------------------------- ---------- ----------
physical reads direct temporary tablespace       3535        676
physical writes direct temporary tablespace      3535        676
session pga memory                             563788          0
session pga memory max                        2464332    1769472
session uga memory                           152176          0
session uga memory max                        1265064     981960

6 rows selected.
```

As you can see, our PGA had grown considerably this time during the processing of our query. It temporarily grew by about 1,728 KB, but the amount of physical IO we had to do in order to sort this data dropped considerably as well (use more memory, swap to disk less often). Now, going to an extreme here:

```
ops$tkyte@ORA10G> alter session set sort_area_size=1048576000;
Session altered.

ops$tkyte@ORA10G> set autotrace traceonly statistics;
ops$tkyte@ORA10G> select * from t order by 1, 2, 3, 4;
48353 rows selected.

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        671  consistent gets
          0  physical reads
          0  redo size
    3461588  bytes sent via SQL*Net to client
      35961  bytes received via SQL*Net from client
       3225  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
      48353  rows processed
ops$tkyte@ORA10G> set autotrace off
```

And measuring from the other session, we can see the memory used so far:

```
ops$tkyte@ORA10G> @watch_stat
6 rows merged.

NAME                                            VALUE       DIFF
---------------------------------------- ---------- ----------
physical reads direct temporary tablespace        3535          0
physical writes direct temporary tablespace       3535          0
session pga memory                           563788          0
session pga memory max                        7510604    5046272
session uga memory                           152176          0
session uga memory max                        7091360    5826296
```

6 rows selected.

We can observe that even though we allocated up to 1 gigabyte of memory to the SORT_AREA, we really only used about 10 megabytes . This shows that the SORT_AREA_SIZE setting is an upper bound, not the default and only allocation size. Here notice also that we did only 1 sort again, but this time it was entirely in memory, there was no temporary space on disk used as evidenced by the lack of physical IO.

Now, if you run this same test on various versions of Oracle, or perhaps even on different operating systems, you *might* see different behavior, and I would expect that your numbers in all cases would be a little different from mine. But the general behavior should be the same. In other words, as you increase the permitted sort area size, and perform large sorts, the amount of memory used by your session will increase. You might notice the PGA memory going up and down, or it might remain constant over time, as we did above. For example, if you were to execute the above test in Oracle 8i, I am sure that you would notice that PGA memory does not shrink back in size (that is, the session pga memory = session pga memory max in all cases). This is to be expected as the PGA is managed as a heap in 8i releases and is created via malloc()'ed memory. In 9i and 10g, there are new methods to attach and release workareas as needed using operating specific memory allocation calls.

Some processes within Oracle will explicitly free PGA memory – others allow it to remain in the heap (sort space for example stays in the heap). Since the UGA is a 'sub-heap' (the 'parent' heap being either the PGA or SGA) it is made to shrink, to give back its memory typically. If we wish, we can force the PGA to shrink, and in fact even in our example from above, it has some effect on the numbers observed:

```
ops$tkyte@ORA10G> exec dbms_session.free_unused_user_memory;
PL/SQL procedure successfully completed.
```

In the other session you might observe now:

```
ops$tkyte@ORA10G> @watch_stat
6 rows merged.

NAME                                           VALUE       DIFF
---------------------------------------- ---------- ----------
physical reads direct temporary tablespace       3535          0
physical writes direct temporary tablespace      3535          0
session pga memory                             498252     -65536
session pga memory max                        7510604          0
session uga memory                             152176          0
session uga memory max                        7091360          0
```

Notice how the current session PGA memory decreased by a small amount. However, you should be aware that on most systems, this is somewhat a waste of time. You may have shrunk the size of the PGA heap as far as Oracle is concerned, but you haven't really given the OS any memory back in most cases. In fact, depending on the OS method of managing memory, you may actually be using more memory in total according to the OS. It will all depend on how malloc(), free(), realloc(), brk(), and sbrk() (the C memory management routines) are implemented on your platform.

The important thing to remember about using the *_AREA_SIZE parameters, is this:

* They control the maximum amount of memory used by a sort/hash operation
* A single query may have many operations taking place that use this memory and multiple sort/hash areas could be created. So, if you set the sort area size to 10MB, you could use 10, 100, 1000 or more meg of RAM in your session. These settings are not session limits, they are limits on a single operation and your session could have many sorts in a single query, or have many queries open that require a sort.

* The memory for these areas is allocated on an "as needed basis", if you set the sort area size to 1 gigabyte as we did, it does not mean you will allocate 1 gigabyte of RAM. It only means that you have given the Oracle process the permission to allocate that much memory for a sort/hash operation

Automatic PGA Memory Management

Starting with Oracle 9i, Release 1 a new way to manage PGA memory was introduced that avoids using the SORT_AREA_SIZE, HASH_AREA_SIZE parameters. It was introduced to try and address a couple of issues:

* Ease of use. Much confusion surrounded how to set the proper *_AREA_SIZE parameters. There was also much confusion over how those parameters actually worked and how memory was allocated.
* Manual allocation was a "one size fits all" method. Typically as the number of users running similar applications against a database went up, the amount of memory used for sorting/hashing went up linearly as well. If 10 concurrent users with a sort area size of 1 megabyte used 10 megabytes of memory, 100 concurrent users would probably use 100 megabytes, 1000 would probably use 1000 megabytes and so on. Unless the DBA was sitting at the console all day long, adjusting the sort/hash area size settings, everyone would pretty much use the same values all day long. Consider our example above, you saw for yourself how the physical IO to temp decreased as the amount of RAM we allowed ourselves to use went up. If you run that example for yourself, you would almost certainly see a decrease in response time as the amount of RAM available for sorting was increased. Manual allocation fixes the amount of memory to be used for sorting at a more or less constant number, regardless of how much memory is actually available, automatic memory management allows us to use the memory when it is available, it dynamically adjusts the amount of memory we use based on the workload.
* As a result of the previous point, it was hard, if not impossible, to keep the Oracle instance inside a "box" memory wise. You could not attempt to limit the amount of memory the instance was going to use as you had no real control over the number of simultaneous sorts/hashes taking place. It was far too easy to use more real memory than was available on the machine.

Enter automatic PGA memory management. Here, you first simply set up and size the SGA. The SGA is a fixed size piece of memory, so you can very accurately see how big it is and that will be it's total size (until and if you change that). You then tell Oracle "this is how much memory you should try to use for workareas – a new umbrella term for the sorting and hashing areas you use". Now, you could in theory take a machine with 2 gigabytes of RAM and allocate 768 MB of RAM to the SGA and 768 MB of RAM to the PGA, leaving 512 MB of RAM for the OS and other processes. I say in theory, because it doesn't work exactly that cleanly, but it is close. Before I discuss why that is true, we'll take a look at how to set it up and turn it on.

The first step is to set two instance initialization parameters, namely:

* WORKAREA_SIZE_POLICY: this parameter may be set to either MANUAL, which will use the sort area and hash area size parameters to control the amount of RAM allocated or AUTO, in which case the amount of RAM allocated will vary based on the current workload present in the database. The default and recommended value is AUTO.
* PGA_AGGREGATE_TARGET: this parameter controls how much memory the instance should allocate, in total, for all workareas used to sort/hash data. It's default value varies by version and may be set by various tools such as DBCA (the Database Configuration Assistant). In general, if you are using automatic PGA memory management, this is a parameter you should explicitly set.

So, assuming that WORKAREA_SIZE_POLICY is set to AUTO, and the PGA_AGGREGATE_TARGET has a non-zero value, you will be using the new automatic PGA memory management.

NOTE Bear in mind the previously-discussed caveat: in Oracle9i – shared server connections will *not* use automatic memory management, they will use the SORT_AREA_SIZE and HASH_AREA_SIZE parameters to decide how much RAM to allocate for various operations. In Oracle 10g and up, automatic PGA memory management is available to both connection types. It is important to properly set the SORT_AREA_SIZE and HASH_AREA_SIZE parameters when using shared server connections with Oracle 9i.

So, the entire goal of automatic PGA memory management is to maximize the use of RAM while at the same time not using more RAM than you want. Under manual memory management, this was a hard goal to achieve. If you set the SORT_AREA_SIZE to 10 MB, when one user was performing a sort operation, the would use up to 10 MB and no more. If 100 users were doing the same, they would use up to 1,000 MB of memory. If you had 500 MB of memory, the single user performing a sort by themselves could have used much more memory and the 100 users should have used much less. That is what automatic PGA memory management was designed to do. Under a light workload, memory usage could be maximized, as the load increases on the system, as more users perform sort or hash operations, the amount of memory allocated to them would decrease – to obtain the goal of utilizing all available RAM, but not attempting to use more than physically exists.

How is the Memory allocated?

A question that comes up frequently is "how is this memory allocated? What will be the amount of RAM used by my session?" It is a hard question to answer for the simple reason that the algorithms for serving out memory under the automatic scheme are not documented and can and will change from release to release. When using things that begin with A – for automatic – you lose a degree of control, the underlying algorithms decide what to do, how to control things.

Some observations can easily be made, based on some information from Metalink note 147806.1 we can see:

* The PGA_AGGREGATE_TARGET is a goal, a goal of an upper limit. It is not a value that is pre-allocated when the database is started up. You can observe this by setting the PGA_AGGREGATE_TARGET to a value much higher than the amount of physical memory you have available on your server. You will not see any large allocation of memory as a result.
* A serial (non-parallel query) session will utilize a small percentage of the PGA_AGGREGATE_TARGET, about 5% or less. So, if you have set the PGA_AGGREGATE_TARGET to 100 MB, you would expect to use no more than about 5 MB per workarea (sort or hash workarea for example). You may well have multiple workareas in your session for multiple queries or more than one sort/hash operation in a single query, but each workarea will be about 5% or less of the PGA_AGGREGATE_TARGET.
* As the workload on my server goes up (more concurrent queries, concurrent users), the amount of PGA memory allocated to my workareas will go down. The database will try to keep the sum of all PGA allocations under the threshold set by PGA_AGGREGATE_TARGET. It would be analogous to having a DBA sit at a console all day and set the SORT_AREA_SIZE and HASH_AREA_SIZE parameters based on the amount of work being performed in the database. We will directly observe this behavior in a test below.
* A parallel query may utilize up to 30% of the PGA_AGGREGATE_TARGET, with each parallel process getting it's slice of that 30%. That is, each parallel process would be able to utilize about 0.3 * PGA_AGGREGATE_TARGET / (number of parallel processes)

Ok, so how can we observe the different workarea sizes being allocated to our session? By using the same technique we used above, in the manual memory management section, to observe the memory used by our session and the amount of IO to temp we performed. The following test was performed on a RedHat Advanced Server 3.0 Linux machine using Oracle 10.1.0.3. This was a 2 CPU Dell Poweredge with hyperthreading enabled, so it was as if there were 4 CPU's available. Using slightly modified versions of the reset_stat.sql and watch_stat.sql scripts from above, I would capture the session statistics for a session *as well as the total statistics for the instance.* The slightly modified watch_stat.sql captured this information via the MERGE statement:

```
merge into sess_stats
using
(
select a.name, b.value
  from v$statname a, v$sesstat b
 where a.statistic# = b.statistic#
   and b.sid = &1
   and (a.name like '%ga %'
        or a.name like '%direct temp%')
 union all
select 'total: ' || a.name, sum(b.value)
  from v$statname a, v$sesstat b, v$session c
 where a.statistic# = b.statistic#
   and (a.name like '%ga %'
        or a.name like '%direct temp%')
   and b.sid = c.sid
   and c.username is not null
 group by 'total: ' || a.name
) curr_stats
on (sess_stats.name = curr_stats.name)
when matched then
  update set diff = curr_stats.value - sess_stats.value,
             value = curr_stats.value
when not matched then
  insert ( name, value, diff )
  values
  ( curr_stats.name, curr_stats.value, null )
/
```

I simply added the UNION ALL section to capture the total PGA/UGA and sort writes by summing over all sessions, in addition to the statistics for a single session. I then ran the following SQL*Plus script in that particular session, the table BIG_TABLE had been created beforehand with 50,000 rows in it. I dropped the primary key from this table, so that all that remained was the table itself (ensuring that a sort process would have to be performed):

```
set autotrace traceonly statistics;
select * from big_table order by 1, 2, 3, 4;
set autotrace off
```

NOTE: See the setup section at the front of this book for details on BIG_TABLE. It is created as a copy of ALL_OBJECTS with a primary key and can have as many or as few rows as you like. The big_table.sql script is documented in that section.

Now, I ran that script against a database with a PGA_AGGREGATE_TARGET of 256 MB, meaning I wanted Oracle to use up to about 256 MB of PGA memory for sorting. I set up another script to be run in other sessions to generate a large sorting load on the machine. This script would loop and using a built-in package DBMS_ALERT, see if it should continue processing. If it should, it would run the same big query, sorting the entire BIG_TABLE table. When the simulation was over, a session could signal all of the sorting processes, the load generators to "stop" and exit. The script used was:

```
declare
    l_msg   long;
    l_status number;
begin
    dbms_alert.register( 'WAITING' );
    for i in 1 .. 999999 loop
        dbms_application_info.set_client_info( i );
        dbms_alert.waitone( 'WAITING', l_msg, l_status, 0 );
        exit when l_status = 0;
        for x in ( select * from big_table order by 1, 2, 3, 4 )
        loop
            null;
        end loop;
    end loop;
end;
/
```

```
exit
```

In order to observe the differing amounts of RAM allocated to the session we were measuring, I initially ran the SELECT in isolation – as the only session. I captured the same six statistics and saved them into another table, along with the count of active sessions. Then I added 25 sessions to the system (ran the above benchmark script with the loop in 25 new sessions). Waited a short period of time – one minute for the system to adjust to this new load and then created a new session, captured the statistics for it with reset_stat.sql, ran the query that would sort, and then ran watch_stat.sql to capture the differences. I did this repeatedly, up to 500 concurrent users. It should be noted that we asked the database instance to do an impossible thing here. As noted above, each connection to Oracle, before even doing a single sort, consumed almost ½ MB of RAM – at 500 users, we would be very close to the PGA_AGGREGATE_TARGET setting just by having them all logged in, let alone actually doing any work! This drives home the point that the PGA_AGGREGATE_TARGET is just that, a target and not a directive. We can and will exceed this value for various reasons. The following table summarizes the findings using approximately 25 user increments:

Active Sessions	PGA used by single Session	PGA in use by system	Writes to Temp by single session	Reads from Temp by single session
1	7.5	2	0	0
27	7.5	189	0	0
51	4.0	330	728	728
76	4.0	341	728	728
101	3.2	266	728	728
126	1.5	214	728	728
151	1.7	226	728	728
177	1.4	213	728	728
201	1.3	218	728	728
226	1.3	211	728	728
251	1.3	237	728	728
276	1.3	251	728	728
301	1.3	281	728	728
326	1.3	302	728	728
351	1.3	324	728	728
376	1.3	350	728	728
402	1.3	367	728	728
426	1.3	392	728	728
452	1.3	417	728	728
476	1.3	439	728	728
501	1.3	467	728	728

NOTE: You might wonder why only 2meg of RAM is reported in use by the system with one active user. It has to do with the way I measured. The simulation would: a) snapshot the single session of interests statistics, we would then b) run the big query in the single session of interest and then c) snapshot that sessions statistics again and finally d) measure how much PGA was used by the system. By the time d) came around, the single session of interest had already completed and given back some of the PGA it was using to sort. So, the number for PGA used by system is an accurate measurement of the systems PGA memory at the time it was measured.

As you can see, when we have one or few active sessions – we performed our sorts entirely in memory, for an active session count of 1 to somewhere less than 50, we were allowed to sort entirely in memory. However, by the time we go 50 users logged in, actively sorting, the database started reigning in the amount of memory we were allowed to utilize at a time. It took a couple of minutes before the amount of PGA being used fell back within acceptable limits (our 256 MB request) but it eventually did. The amount of PGA memory allocated to our session dropped from 7.5 MB to 4 MB to 3.2 MB and eventually in the area of 1.7 to 1.3 MB (remember, parts of that PGA are not for sorting but rather for other operations, just the act of logging in created a ½ MB PGA). The total PGA in use by the system remained within tolerable limits until somewhere around 300 to 351 users. There we see we started to exceed on a regular basis the PGA_AGGREGATE_TARGET and continued to do so until the end of the test. We gave the database instance in this case an impossible task – just the very act of having 350 users, most executing a PL/SQL, plus the sort they were all requesting just did not fit into the 256 MB of RAM we had targeted. It simply could not be done. Each session therefore used as little memory as possible, but had to allocate as much memory as it needed. By the time we finished this exercise, 500 active sessions were using a total of 467 MB of PGA memory – as little as they could.

You should however consider what the above chart would have looked like under a manual memory management situation. Suppose the SORT_AREA_SIZE had been set to 5 MB. The math is very straightforward, each session would have been able to perform the sort in RAM (or virtual memory as the machine ran out of real RAM) and thus would have consumed 6 to 7 MB of RAM per session (the amount used without sorting to disk in the single user case above). I ran the above test with SORT_AREA_SIZE set to 5 MB and as we went from 1 user and added 25 at a time, the numbers remained consistent:

Active Sessions	PGA used by single Session	PGA in use by system	Writes to Temp by single session	Reads from Temp by single session
1	6.4	5	728	728
26	6.4	137	728	728
51	6.4	283	728	728
76	6.4	391	728	728
102	6.4	574	728	728
126	6.4	674	728	728
151	6.4	758	728	728
176	6.4	987	728	728
202	6.4	995	728	728
226	6.4	1227	728	728
251	6.4	1383	728	728
277	6.4	1475	728	728
302	6.4	1548	728	728

Had I been able to complete the test (I have 2gig of real memory on this server, my SGA was 600 MB, by the time I got to 325 users, the machine was paging and swapping to the point where it was impossible to continue), at 500 users we would have allocated around

2,750 MB of RAM! So, the DBA would probably not set the SORT_AREA_SIZE to 5 MB on this system, but rather about ½ MB, in an attempt to keep the maximum PGA usage at a bearable level at peak. Now at 500 users we would have had about 500 MB of PGA allocated, perhaps similar to what we observed with automatic memory management – but at times when we had fewer users, we would have still written to temp, not performed the sort in memory. In fact, when running the above test with a SORT_AREA_SIZE of ½ MB, you would observe:

Active Sessions	PGA used by single Session	PGA in use by system	Writes to Temp by single session	Reads from Temp by single session
1	1.2	1	728	728
26	1.2	29	728	728
51	1.2	57	728	728
76	1.2	84	728	728
101	1.2	112	728	728
126	1.2	140	728	728
151	1.2	167	728	728
176	1.2	194	728	728
201	1.2	222	728	728
226	1.2	250	728	728

This represents a very predicable, but suboptimal, use of memory as the workload increases or decreases over time. Automatic PGA memory management was designed exactly to allow the small community of users use as much RAM as possible when it was available and back off on this allocation over time as the load increased, and increase over time as the load decreased.

Using PGA_AGGREGATE_TARGET to control memory allocation

Above I wrote that "in theory" we can use the PGA_AGGREGATE_TARGET to control the overall amount of PGA memory used by the instance. We saw in the last example that this is not a hard limit however, the instance will attempt to stay within the bounds of the PGA_AGGREGATE_TARGET, but if it cannot, it will not stop processing, it will just be forced to exceed that threshold.

Another reason this limit is "in theory" is because the workareas, while a large contributor to PGA memory, are not the *only* contributors to PGA memory. Many things contribute to your PGA memory allocation and only the workareas are under the control of the database instance. If you create and execute a PL/SQL block of code that fills in a large array with data – Oracle cannot do anything but allow you to do it. Consider the following quick example. We'll create a package that can hold some persistent (global) data in the server:

```
ops$tkyte@ORA10G> create or replace package demo_pkg
  2  as
  3          type array is table of char(2000) index by binary_integer;
  4          g_data array;
  5  end;
  6  /
Package created.
```

Now we'll measure the amount of memory our session is currently using in the PGA/UGA (I was using dedicated server in this example, so the UGA is a subset of the PGA memory):

```
ops$tkyte@ORA10G> select a.name, to_char(b.value, '999,999,999') value
  2    from v$statname a, v$mystat b
  3   where a.statistic# = b.statistic#
  4     and a.name like '%ga memory%';

NAME                           VALUE
------------------------------ ------------
session uga memory              1,212,872
session uga memory max          1,212,872
session pga memory              1,677,900
session pga memory max          1,677,900
```

So, initially we are using about 1.5 MB of PGA memory in our session (as a result of compiling a PL/SQL package, running this query and so on). Now, we'll run our query against BIG_TABLE again using the same 256 MB PGA_AGGREGATE_TARGET (this was done in an otherwise idle instance, we are the only session requiring memory right now)

```
ops$tkyte@ORA10GR1> set autotrace traceonly statistics;
ops$tkyte@ORA10GR1> select * from big_table order by 1,2,3,4;
50000 rows selected.

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        721  consistent gets
          0  physical reads
          0  redo size
    2644246  bytes sent via SQL*Net to client
      37171  bytes received via SQL*Net from client
       3335  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
      50000  rows processed
ops$tkyte@ORA10GR1> set autotrace off
```

As you can see, the sort was done entirely in memory, and in fact if we peek at our session's PGA/UGA usage we can see how much we used:

```
ops$tkyte@ORA10GR1> select a.name, to_char(b.value, '999,999,999') value
  2    from v$statname a, v$mystat b
  3   where a.statistic# = b.statistic#
  4     and a.name like '%ga memory%';

NAME                           VALUE
------------------------------ ------------
session uga memory              1,212,872
session uga memory max          7,418,680
session pga memory              1,612,364
session pga memory max            7,838,284
```

The same 7.5 MB of RAM we observed earlier. Now, we will proceed to fill up that CHAR array we have in the packge (a CHAR datatype is blank padded so each of these array elements is at least 2,000 characters in length):

```
ops$tkyte@ORA10G> begin
  2          for i in 1 .. 100000
  3          loop
  4                  demo_pkg.g_data(i) := 'x';
  5          end loop;
  6  end;
  7  /
PL/SQL procedure successfully completed.
```

Upon measuring our session's current PGA utilization after that, you'll find something similar to the following:

```
ops$tkyte@ORA10GR1> select a.name, to_char(b.value, '999,999,999') value
  2    from v$statname a, v$mystat b
  3   where a.statistic# = b.statistic#
  4     and a.name like '%ga memory%';

NAME                           VALUE
------------------------------ ------------
session uga memory              312,952,440
session uga memory max          312,952,440
session pga memory              313,694,796
session pga memory max          313,694,796
```

Now, that is memory allocated in the PGA that the database itself cannot control. We already exceed the PGA_AGGREGATE_TARGET and there is quite simply nothing the database can do about it – it would have to fail our request if it did anything and it will only do that when the OS reports back "there is no more memory to give". If we wanted, we could allocate more space in that array, place more data in it – and the database would just have to do it for us.

However, the database is aware of the fact that we have done this thing – it does not ignore the memory is cannot control, rather it recognizes that it is being used and backs off the size of memory allocated for workareas accordingly, so if we re-run the same sort query:

```
ops$tkyte@ORA10GR1> set autotrace traceonly statistics;
ops$tkyte@ORA10GR1> select * from big_table order by 1,2,3,4;
50000 rows selected.

Statistics
----------------------------------------------------------
          6  recursive calls
          2  db block gets
        721  consistent gets
        728  physical reads
          0  redo size
    2644246  bytes sent via SQL*Net to client
      37171  bytes received via SQL*Net from client
       3335  SQL*Net roundtrips to/from client
```

```
          0  sorts (memory)
          1  sorts (disk)
      50000  rows processed
ops$tkyte@ORA10GR1> set autotrace off
```

Note that this time we sorted to disk – the database did not give us the 7 or so MB of RAM needed to do this in memory since we had already exceeded the PGA_AGGREGATE_TARGET. So, since some PGA memory is outside of Oracle's control, it is easy for us to exceed the PGA_AGGREGATE_TARGET by simply allocating lots of really large data structures in our PL/SQL code. I am not *recommending* you do that by any means, just pointing out that the PGA_AGGREGATE_TARGET is a more of a request than a hard limit.

Manual or Auto Memory Management?

So, which method should you use – manual or automatic? My preference is to use the automatic PGA memory management by default.

CAUTION I'll say this from time to time in the book but please, do not make any changes to a production system, a live system, without testing for any side effects first. For example, please do not read this chapter – check your system and find you are using manual memory management and just turn on the automatic memory management. Query plans may change, performance may be impacted. One of three things could happen. 1) Things run exactly the same, 2) Things run better than they did before, or 3) Things run much worse then they did before. Exercise caution before making changes, test the proposed change first.

One of the most perplexing things for a DBA can be setting the individual parameters - especially parameters such as SORT|HASH_AREA_SIZE and so on. Many times, I see systems running with incredibly small values for these parameters; values so small that system performance is massively impacted in a negative way. This is probably a result of the fact that the default values are very small themselves – 64 KB for sorting and 128 KB for hashing. There is a lot of confusion over how big or small these should be. Not only that, but the values you would like to use for them might vary over time, as the day goes by. At 8am in the morning, with 2 users, a 50 MB sort area size might be reasonable for the single user logged in. However, at 12 noon with 500 users, 50 MB might be not be appropriate. This is where the WORKAREA_SIZE_POLICY = AUTO setting and the corresponding PGA_AGGREGATE_TARGET come in useful. Setting the PGA_AGGREGATE_TARGET, the amount of memory you would like Oracle to feel free to use to sort and hash, is conceptually easier than trying to figure out the perfect SORT|HASH_AREA_SIZE especially since there isn't a perfect value for these parameters, the perfect value varies by workload.

Historically, the DBA configured the amount of memory used by Oracle by setting the size of the SGA (the buffer cache, the log buffer, the shared/large/java pools). The remaining memory on the machine would then be used by the dedicated or shared servers in the PGA region. The DBA had little control over how much of this memory would or would not be used. They could set the SORT_AREA_SIZE, but if there were 10 concurrent sorts, then Oracle could use as much as 10 * SORT_AREA_SIZE bytes of RAM. If there were only 100 concurrent sort, then Oracle would use 100 * SORT_AREA_SIZE bytes, for 1,000 concurrent sorts, 1,000 times SORT_AREA_SIZE and so on. Couple that with the fact that there are other things that go into the PGA - and you really don't have good control over the maximal use of PGA memory on the system.

What you would like to have happen is for this memory to be used differently as the load on the system grows and shrinks. The more users, the less RAM they should use. The less users, the more RAM they should use. Setting WORKAREA_SIZE_POLICY = AUTO is just such a way to achieve this. The DBA will specify a single size now, the PGA_AGGREGATE_TARGET, the maximum amount of PGA memory that the database should strive to use. Oracle will then distribute this memory over the active sessions as it sees fit. Further, with Oracle9i release 2 and up there is even PGA advisory (part of statspack, available via a V$ dynamic performance view and visible in enterprise manager), much like the buffer cache advisor. It will tell you over time what the optimal PGA_AGGREGATE_TARGET for your system would be. You can use this to either dynamically change the PGA size online (if you have sufficient RAM), or to decide whether you might need more RAM on your server to achieve optimal performance.

Are there times however when you don't want to use it? Absolutely, and fortunately, they seem to be the exception and not the rule. The automatic memory management was designed to be "fair", to be multi-user "fair". In anticipation of additional users joining the system, the automatic memory management will limit the amount of memory you would be allocated as a percentage of the PGA_AGGREGATE_TARGET. But what happens when you don't want to be fair? When you *know* that you should get all of the memory available? Well, that would be time to use the ALTER SESSION command to disable automatic memory management in your session (leaving it in place for all others), and to manually set your SORT|HASH_AREA_SIZE as needed. For example, that large batch process that takes place at 2am in the morning and does tremendously large hash joins, some index builds and the like? It should be permitted to use all of the resources on the machine, it does not want to be "fair" about memory use, it wants it all – as it knows it is the only thing happening in the database right now. That batch job can certainly issue the ALTER SESSION commands and make use of all resources available.

So, in short, I prefer to use automatic PGA memory management for end user sessions – for the applications that run day to day against my database. For large batch jobs that run during periods of time when they are the only thing in the database – using manual memory management makes sense.

PGA/UGA Wrap-up

So, here we have looked at the two memory structures, the PGA and the UGA. We understand now that the PGA is private to a process. It is the set of variables that an Oracle dedicated or shared server needs to have independent of a session. The PGA is a 'heap' of memory in which other structures may be allocated. The UGA on the other hand, is also a heap of memory in which various session-specific structures may be defined. The UGA is allocated from the PGA when you use a dedicated server to connect to Oracle, and from the SGA under a shared server connection. This implies that when using shared server, you must size your SGA's large pool to have enough space in it to cater for every possible user that will ever connect to your database concurrently. So, the SGA of a database supporting shared server connections is generally much larger than the SGA for a similarly configured, dedicated server mode only, database.

The SGA

Every Oracle instance has one big memory structure collectively referred to as the *SGA*, the **S**ystem **G**lobal **A**rea. This is a large, shared memory structure that every Oracle process will access at one point or another. It will vary in size from a couple of MBs on small test systems to hundreds of MBs on medium to large systems, and into many GBs in size for really big systems.

On a UNIX operating system, the SGA is a physical entity that you can 'see' from the operating system command line. It is physically implemented as a shared memory segment – a standalone piece of memory to which processes may attach. It is possible to have an SGA on a system without having any Oracle processes; the memory stands alone. It should be noted however that if you have an SGA without any Oracle processes, it indicates that the database crashed in some fashion. It is not a normal situation but it can happen. This is what an SGA 'looks' like on RedHat Linux:

```
[tkyte@localhost tkyte]$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x99875060 491522     ora10g     660        606076928  14
0x0d998a20 425992     ora9ir2    660        356515840  55
```

There are two SGA's represented here, one owned by the OS user ora10g and the other by the OS user ora9ir2. They are about 578 and 340 MB in size, respectively.

On Windows, you really cannot see the SGA as you can in UNIX/Linux. Since, on the Windows platform, Oracle executes as a single process with a single address space, so the SGA is allocated as private memory to the ORACLE.EXE process. If you use the Windows Task Manager or some other performance tool, you can see how much memory ORACLE.EXE has allocated, but you cannot see what is the SGA versus any other piece of allocated memory, you cannot see the SGA on Windows as a distinct entity like you can on Unix/Linux.

Within Oracle itself we can see the SGA regardless of platform, using another magic V$ table called V$SGASTAT. It might look like this:

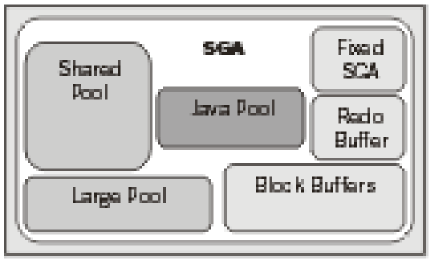
```
ops$tkyte@ORA10G> compute sum of bytes on pool
ops$tkyte@ORA10G> break on pool skip 1
ops$tkyte@ORA10G> select pool, name, bytes
  2  from v$sgastat
  3  order by pool, name;

POOL         NAME                                BYTES
------------ ------------------------------ ----------
java pool    free memory                      16777216
************                                ----------
sum                                           16777216

large pool   PX msg pool                         64000
             free memory                      16713216
************                                ----------
sum                                           16777216

shared pool  ASH buffers                       2097152
```

```
          FileOpenBlock                746704
          KGLS heap                   777516
          KQR L SO                      29696
          KQR M PO                     599576
          KQR M SO                      42496
…
          sql area                  2664728
          table definiti                280
          trigger defini               1792
          trigger inform                1944
          trigger source                 640
          type object de              183804
************                       ----------
sum                              352321536

streams pool free memory              33554432
************                       ----------
sum                               33554432

          buffer_cache             1157627904
          fixed_sga                   779316
          log_buffer                  262144
************                       ----------
sum                             1158669364


43 rows selected.
```

The SGA is broken up into various *pools*. They are:

* *Java pool* – The Java pool is a fixed amount of memory allocated for the JVM running in the database. In Oracle10g the java pool may be resized online while the database is up and running.
* *Large pool* – The large pool is used by shared server connections for session memory, by Parallel Execution for message buffers, and by RMAN Backup for disk I/O buffers. This pool is resizable online in both Oracle 10g and 9i.
* *Shared pool* – The shared pool contains shared cursors, stored procedures, state objects, dictionary caches, and many dozens of other bits of data. This pool is resizable online in both Oracle 10g and 9i.
* *Streams pool* – This is a pool of memory used exclusively by Oracle streams, a data sharing tool within the database. This pool is new in Oracle 10g and is resizable online. In the event the streams pool is not configured and you use the streams functionality, Oracle will utilize up to 10% of the shared pool for streams memory.
* *The 'Null' pool* – This one doesn't really have a name. It is the memory dedicated to block buffers (cached database blocks), the redo log buffer and a 'fixed SGA' area.

So, a typical SGA might look like this:



Insert 2433f0211scrap.gif

Figure 2-11. Typical SGA.

The parameters that have the most effect on the overall size of the SGA are:

* JAVA_POOL_SIZE – controls the size of the Java pool.
* SHARED_POOL_SIZE – controls the size of the shared pool, to some degree.
* LARGE_POOL_SIZE – controls the size of the large pool.
* DB_*_CACHE_SIZE – there are eight of these cache_size parameters to control the sizes of the various buffer caches available.
* LOG_BUFFER – controls the size of the redo buffer to some degree.
* SGA_TARGET – used with automatic SGA memory management in Oracle 10g and above.
* SGA_MAX_SIZE – used to control the maximum size the SGA can be resized to while the database is up and running.

In Oracle9i, the various SGA components must be manually sized by the DBA but, starting in Oracle 10g, there is a new option to consider – automatic SGA memory management, whereby the database instance will allocate and reallocate the various SGA components at runtime, in response to workload conditions. When using the automatic memory management with Oracle 10g, it is a matter of simply setting the SGA_TARGET parameter to the desired SGA size. The database instance will take it from there, allocating memory to the various pools as needed, and even taking memory away from one pool to give to another over time.

Regardless of whether you are using automatic or manual memory management, you will find that memory is allocated to the various pools in units called *granules*. A single granule is an area of memory from 4 to 8 to 16 MB in size. The granule is the smallest unit of allocation, so if you ask for a Java pool of 5 MB and your granule size is 4 MB, Oracle will actually allocate 8 MB to the Java pool (8 being the smallest number greater than or equal to 5 that is a multiple of the granule size 4). The size of a granule is determined by the size of your SGA (sounds recursive to a degree, as the size of the SGA is dependent on the granule size). You can view the granule sizes used for each pool by querying V$SGA_DYNAMIC_COMPONENTS. In fact, we can use this view to see how the total SGA size might affect the size of the granules:

```
sys@ORA10G> show parameter sga_target
```

```
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
sga_target                           big integer 576M


sys@ORA10G> select component, granule_size from v$sga_dynamic_components;

COMPONENT                 GRANULE_SIZE
------------------------- ------------
shared pool                    4194304
large pool                     4194304
java pool                      4194304
streams pool                   4194304
DEFAULT buffer cache           4194304
KEEP buffer cache              4194304
RECYCLE buffer cache           4194304
DEFAULT 2K buffer cache        4194304
DEFAULT 4K buffer cache        4194304
DEFAULT 8K buffer cache        4194304
DEFAULT 16K buffer cache       4194304
DEFAULT 32K buffer cache       4194304
OSM Buffer Cache               4194304

13 rows selected.
```

In this example, I was using automatic SGA memory management and controlled the size of the SGA via the single parameter SGA_TARGET. When my SGA size is under about 1 GB in size, the granule size is 4 MB. When the SGA size is increased to some threshold over 1 GB (it will vary slightly from operating system to operating system and even release to release), I see an increased granule size:

```
sys@ORA10G> alter system set sga_target = 1512m scope=spfile;
System altered.

sys@ORA10G> startup force
ORACLE instance started.

Total System Global Area 1593835520 bytes
Fixed Size                   779316 bytes
Variable Size             401611724 bytes
Database Buffers         1191182336 bytes
Redo Buffers                 262144 bytes
Database mounted.
Database opened.
sys@ORA10G> select component, granule_size from v$sga_dynamic_components;

COMPONENT                 GRANULE_SIZE
------------------------- ------------
shared pool                   16777216
large pool                    16777216
java pool                     16777216
streams pool                  16777216
DEFAULT buffer cache          16777216
KEEP buffer cache             16777216
```

```
RECYCLE buffer cache          16777216
DEFAULT 2K buffer cache       16777216
DEFAULT 4K buffer cache       16777216
DEFAULT 8K buffer cache       16777216
DEFAULT 16K buffer cache      16777216
DEFAULT 32K buffer cache      16777216
OSM Buffer Cache              16777216

13 rows selected.
```

As you can see, at 1.5 GB of SGA, my pools will be allocated using 16 MB granules, so any given pool size will be some multiple of 16 MB.

With this in mind, let's look at each of the major SGA components in turn.

Fixed SGA

The *fixed SGA* is a component of the SGA that varies in size from platform to platform and release to release. It is 'compiled' into the Oracle binary itself at installation time (hence the name 'fixed'). The fixed SGA contains a set of variables that point to the other components of the SGA, and variables that contain the values of various parameters. The size of the fixed SGA is something over which we have no control, and it is generally very small. Think of this area as a 'bootstrap' section of the SGA, something Oracle uses internally to find the other bits and pieces of the SGA.

Redo Buffer

The redo buffer is where data that needs to be written to the online redo logs will be cached temporarily, before it is written to disk. Since a memory-to-memory transfer is much faster then a memory to disk transfer, use of the redo log buffer can speed up operation of the database. The data will not reside in the redo buffer for a very long time. In fact, the contents of this area are flushed:

* Every three seconds, or
* Whenever someone commits, or
* When it gets one third full or contains 1 MB of cached redo log data.

For these reasons, it will be a very rare system that will benefit from a redo buffer of more than a couple of megabytes in size. A large system with lots of concurrent transactions could benefit somewhat from large redo log buffers because while LGWR (the process responsible for flushing the redo log buffer to disk) is writing a portion of the log buffer, other sessions could be filling it up. In general, a long running transaction that generates a lot of redo log will benefit the most from a larger than normal log buffer, as it will be continuously filling up part of the redo log buffer while LGWR is busy writing out some of it. The larger and longer the transaction, the larger the benefit it could receive from a generous log buffer.

The default size of the redo buffer, as controlled by the LOG_BUFFER parameter, is whatever the greater of 512 KB and (128 * number of CPUs) KB. The minimum size of this area is operating system dependent. If you would like to find out what that is, just set your

LOG_BUFFER to 1 byte and restart your database. For example, on my RedHat Linux instance I see:

```
sys@ORA10G> alter system set log_buffer=1 scope=spfile;
System altered.

sys@ORA10G> startup force
ORACLE instance started.

Total System Global Area 1593835520 bytes
Fixed Size                   779316 bytes
Variable Size             401611724 bytes
Database Buffers         1191182336 bytes
Redo Buffers                 262144 bytes
Database mounted.
Database opened.
sys@ORA10G> show parameter log_buffer

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
log_buffer                           integer     262144
```

The smallest log buffer I can really have, regardless of my settings, is going to be 256 KB on this system.

Block Buffer Cache

So far, we have looked at relatively small components of the SGA. Now we are going to look at one that is possibly huge in size. The block buffer cache is where Oracle will store database blocks before writing them to disk, and after reading them in from disk. This is a crucial area of the SGA for us. Make it too small and our queries will take forever to run. Make it too big and we'll starve other processes (for example, you will not leave enough room for a dedicated server to create its PGA and you won't even get started).

In earlier releases of Oracle, there was a single block buffer cache, all blocks from any segment went into this single area. Starting with Oracle 8.0 we had three places to store cached blocks from individual segments in the SGA:

* **Default Pool**: Where all segment blocks are cached normally, this is the original and previously only buffer pool
* **Keep Pool**: An alternate buffer pool where by convention you would assign segments that were accessed fairly frequently, but still got aged out of the default buffer pool due to other segments needing space, perhaps from a large table full table scan.
* **Recycle Pool**: An alternate buffer pool where by convention you would assign large segments that you full scan frequently. You would separate them out from the segments in the Default and Keep pools so that they would not cause those blocks to age out of the cache.

Note that for the Keep and Recycle pool I stated "by convention". There is nothing in place to ensure that you use the Keep pool in the fashion described, nor the Recycle pool. In fact, the three pools manage blocks in an identical fashion, they do not have different algorithms for aging or caching blocks. The goal here was to give the DBA the ability to segregate segments to "hot", "warm" and "do not care to cache" areas. The theory was that objects in the default pool would be "hot" enough (used enough) to warrant staying in the cache all by themselves, the cache would keep them in memory since they were very popular blocks (hot blocks). You might have had some segments that were fairly popular, but not really hot, we'll call these the warm blocks. These segments blocks could get flushed from the cache to make room for some blocks you used infrequently (the "do not care to cache" blocks). In order to keep these warm segments blocks cached, you could either:

* Assign these segments to the Keep Pool, in an attempt to let the warm blocks stay in the buffer cache longer.
* Assign the "do not care to cache" segments to the Recycle Pool, keeping the recycle pool fairly small so as to let the blocks come into the cache and leave the cache rapidly (decrease the overhead of managing them all)

This increased the management work the DBA had to perform – there are three caches to think about, you have to size them, you have to assign objects to them. All in all, they were generally used as a very fine, low level tuning device – after most all other tuning attempts had been looked at (if you could rewrite a query to do 1/10th the IO rather then setup multiple buffer pools, that would be my choice!).

Starting in Oracle 9i, the DBA had up to 4 more optional caches, the db_Nk_cache's, to consider in addition to the Default, Keep and Recycle pools. These caches were added in support of multiple blocksizes in the database. Prior to Oracle 9i, a database would have a single blocksize (typically 2, 4, 8, 16 or 32k in size). Starting with Oracle 9i, a database can have a default blocksize, this will be the size of the blocks stored in the Default, Keep or Recycle pools as well as up to 4 non-default blocksizes, as explained in chapter 3.

The blocks in these buffer caches are managed in the same way as the blocks in the original Default pool, there are no special algorithm changes for them either. So, onto how the blocks are managed in these pools

Managing Blocks in the Buffer Cache

For simplicity, we'll assume in this discussion that there is just the single Default pool – since the other pools are managed in the same way, we only need discuss one of them.

The blocks in the buffer cache are basically managed in two different lists:

* The 'dirty' list of blocks that need to be written by the database block writer (DBWn – we'll be taking a look at that process a little later).
* A list of 'not dirty' blocks.

The list of non-dirty blocks used to be a *LRU* (**L**east **R**ecently **U**sed) list in Oracle 8.0 and before. Blocks were listed in order of use. The algorithm has been modified slightly in Oracle 8i and in later versions. Instead of maintaining the list of blocks in some physical order, Oracle employs a 'touch count' algorithm. This effectively increments a counter associated with a block as you hit it in the cache. We can see this in one of the truly magic set of tables, the X$ tables. These X$ tables are wholly undocumented by Oracle, but information about them leaks out from time to time.

The X$BH table shows information about the blocks in the block buffer cache (more information than the documented V$BH view). Here, we can see the 'touch count' get incremented as we hit blocks. We can run a query against that view to find the five "currently hottest blocks" and join that information to the DBA_OBJECTS view to see what segments they belong to. The query you can use is as follows. It orders the rows in X$BH by the TCH (touch count) column and keeps the first 5. Then we join the X$BH information to DBA_OBJECTS by X$BH.OBJ to DBA_OBJECTS.DATA_OBJECT_ID:

```
sys@ASKTOM> select tch, owner, object_name, object_type
   from dba_objects,
       (select *
          from ( select tch, obj
                   from x$bh
                  order by tch desc
               )
         where rownum <= 5
       ) hottest_blocks
  where dba_objects.data_object_id = hottest_blocks.obj
  order by tch desc
 /

       TCH OWNER        OBJECT_NAME                    OBJECT_TYPE
---------- ------------ ------------------------------ -------------------
        65 FLOWS_010403 WWV_FLOW_DATA_IDX1             INDEX
        55 FLOWS_010403 WWV_FLOW_STEP_ITEMS            TABLE
        43 FLOWS_010403 WWV_FLOW_STEP_BRANCHES         TABLE
        42 FLOWS_010403 WWV_FLOW_PAGE_PLUGS_PK         INDEX
        42 FLOWS_010403 WWV_FLOW_PATCHES               TABLE
```

You can even watch as Oracle increments the touch count on a block that you query repeatedly. We will use the magic table "DUAL" in this example – we know it is a one row, one column table. What we need to know about it is the block information for that single row. The built-in DBMS_ROWID package is good for getting that. Additionally, since we query ROWID from DUAL, we are making Oracle actually read the real DUAL table from the buffer cache, not the "virtual" DUAL table enhancement of Oracle 10g

---

NOTE: Prior to Oracle 10g, querying DUAL would incur a full table scan of a real table named DUAL stored in the data dictionary. If you set autotrace on and query "SELECT DUMMY FROM DUAL", you will observe some IO in all releases of Oracle (consistent gets). In 9i and before, if you query "SELECT SYSDATE FROM DUAL", you will also see real IO occur. However, in Oracle 10g, that SELECT SYSDATE is recognized as not needing to actually query the DUAL table (since we are not asking for the column or rowid from dual) and is done in a manner similar to calling a function. The table is not full scanned, just SYSDATE is returned to the application. This small change can dramatically decrease the amount of consistent gets a system that uses DUAL heavily performs.

So every time we run the following query, we should be hitting the real DUAL table:

```
sys@ORA10G> select tch, file#, dbablk
   from x$bh
  where (dbablk,file#) in
  (select dbms_rowid.rowid_block_number(rowid),
          dbms_rowid.rowid_to_absolute_fno(rowid,'SYS','DUAL')
         from dual)
  7     and state = 1
  8   /

       TCH      FILE#     DBABLK
---------- ---------- ----------
        18          1       1858

sys@ORA10G> /

       TCH      FILE#     DBABLK
---------- ---------- ----------
        19          1       1858

sys@ORA10G> /

       TCH      FILE#     DBABLK
---------- ---------- ----------
        19          1       1858

sys@ORA10G> /

       TCH      FILE#     DBABLK
---------- ---------- ----------
        20          1       1858
```

Almost every time I touch that block, the counter goes up. Note that I said "almost", the touch count is, by design, an imprecise number. It will be incremented most of the times, but it is not considered important that the number be 100% accurate, it is close. If you run this on your system, you may well see different numbers.

So, in Oracle 8i and above, a block buffer no longer moves to the head of the list as it is used to, rather it stays where it is in the list, and has its 'touch count' incremented. Blocks will naturally tend to move in the list over time however, because blocks are taken out of the list and put into the dirty list (to be written to disk by DBWn). Also, as they are reused over time -- when the buffer cache is effectively full, and some block with a small 'touch count' is taken out of the list, it will be placed back into approximately the middle of the list with the new data block. The whole algorithm used to manage these lists is fairly complex and changes subtly from release to release of Oracle as improvements are made. The actual full details are not relevant to us as developers, beyond the fact that heavily used blocks will be cached and blocks that are not used heavily will not be cached for long.

Multiple Block Sizes

Staring in Oracle 9i and above, you can have multiple database block sizes in the same database. Prior to this, all blocks in a single database were the same size and in order to have a different blocksize, you had to rebuild the entire database. Now, you can have a mixture of the "default" blocksize (the blocksize you used when you initially created the database, the size that is used for the SYSTEM tablespace) and up to 4 other blocksizes. Each unique blocksize must have its own buffer cache area, the default, keep and recycle pools will only cache blocks of the default size. In order to have a non-default blocksize in your database you will need to have configured a buffer pool to hold them.

In this example, my default blocksize is 8k. I will attempt to create a tablespace with a 16k blocksize:

```
ops$tkyte@ORA10G> create tablespace ts_16k
  2  datafile size 5m
  3  blocksize 16k;
create tablespace ts_16k
*
ERROR at line 1:
ORA-29339: tablespace block size 16384 does not match configured block sizes

ops$tkyte@ORA10G> show parameter 16k

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
db_16k_cache_size                    big integer 0
```

Right now, since I have not configured a 16 KB cache, I cannot create such a tablespace. I could do one of a couple things right now in order to rectify this. I could set the db_16k_cache_size parameter in my init.ora or spfile and restart the database. I could shrink one of my other SGA components in order to make room for a 16 KB cache in the existing SGA. Or, I might be able to just allocate a 16 KB cache if the SGA_MAX_SIZE parameter was larger than my current SGA size. In this example, I will shrink my db_cache_size since I current have it set rather large:

```
ops$tkyte@ORA10G> show parameter db_cache_size

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
db_cache_size                        big integer 1G

ops$tkyte@ORA10G> alter system set db_cache_size = 768m;
System altered.

ops$tkyte@ORA10G> alter system set db_16k_cache_size = 256m;
System altered.

ops$tkyte@ORA10G> create tablespace ts_16k
  2  datafile size 5m
  3  blocksize 16k;
Tablespace created.
```

So, now I have another buffer cache set up, one to cache any blocks that are 16 KB in size. The default pool, controlled by the db_cache_size parameter, is 768 MB in size and the 16 KB cache, controlled by the db_16k_cache_size is 256 MB in size. These two caches are mutually exclusive, if one "fills up", it cannot use space in the other. It gives the DBA a very fine degree of control over memory use, but that comes at a price. A price of complexity and management. These multiple block sizes were not intended as a performance or tuning feature, but rather came about in support of transportable tablespaces – the ability to take formatted data files from one database and transport or attach them to another database. They were implemented in order to take datafiles from a transactional system that was using an 8 KB blocksize and transport that information to a data warehouse that was utilizing a 16 or 32 KB blocksize.

They do serve a good purpose however in testing a theories. If you wanted to see how your database would operate with a different block size, how much space for example would a certain table consume if you used a 4 KB block instead of a 8 KB block, you can now test easily without having to create an entirely new database instance.

You may also be able to use this as a very finely focused tuning tool – for a specific set of segments, giving them their own private buffer pools. Or, in a system that is a hybrid system with transactional users using one set of data and reporting/warehouse users querying a separate set of data. The transactional data would benefit from the smaller blocksizes due to less contention on the blocks (less data per block means less people in general would go after the same block at the same time) as well as better buffer cache utilization – they read into the cache only the data they are interested in, the single row or small set of rows. The reporting/warehouse data, which might be based on the transactional data, would benefit from the larger blocksizes due in part to less block overhead (it takes less storage overall), and larger IO sizes perhaps. And since they do not have the some update contention issues, the fact that there are more rows per block is not a concern, but a benefit. Additionally, the transactional users get their own buffer cache in effect, they do not have to worry about the reporting queries overrunning their cache.

But in general, the default, keep and recycle pools should be sufficient for fine tuning the block buffer cache and multiple blocksizes would be used primarily for transporting data from database to database and perhaps a hybrid reporting/transactional system.

Shared Pool

The shared pool is one of the most critical pieces of memory in the SGA, especially in regards to performance and scalability. A shared pool that is too small can kill performance to the point where the system appears to hang. A shared pool that is too large can do the same thing. A shared pool that is used incorrectly will be a disaster as well.

So, what exactly is the shared pool? The shared pool is where Oracle caches many bits of 'program' data. When we parse a query, the results of that are cached here. Before we go through the job of parsing an entire query, Oracle searches here to see if the work has already been done. PL/SQL code that you run is cached here, so the next time you run it, Oracle doesn't have to read it in from disk again. PL/SQL code is not only cached here, it is shared here as well. If you have 1000 sessions all executing the same code, only one copy of the code is loaded and shared amongst all sessions. Oracle stores the system parameters in the shared pool. The data dictionary cache, cached information about database objects, is stored here. In short, everything but the kitchen sink is stored in the shared pool.

The shared pool is characterized by lots of small (4 KB or thereabouts) chunks of memory. The memory in the shared pool is managed on a LRU basis. It is similar to the buffer cache in that respect – if you don't use it, you'll lose it. There is a supplied package, DBMS_SHARED_POOL, which may be used to change this behavior – to forcibly pin objects in the shared pool. You can use this procedure to load up your frequently used procedures and packages at database startup time, and make it so they are not subject to aging out. Normally though, if over time a piece of memory in the shared pool is not reused, it will become subject to aging out. Even PL/SQL code, which can be rather large, is managed in a paging mechanism so that when you execute code in a very large package, only the code that is needed is loaded into the shared pool in small chunks. If you don't use it for an extended period of time, it will be aged out if the shared pool fills up and space is needed for other objects.

The easiest way to break Oracle's shared pool is to not use bind variables. As we saw in Chapter 1 on *Developing Successful Oracle Applications*, not using bind variables can bring a system to its knees for two reasons:

* The system spends an exorbitant amount of CPU time parsing queries.
* The system expends an extremely large amount of resources managing the objects in the shared pool as a result of never reusing queries.

If every query submitted to Oracle is a unique query with the values hard-coded, the concept of the shared pool is substantially defeated. The shared pool was designed so that query plans would be used over and over again. If every query is a brand new, never before seen query, then caching only adds overhead. The shared pool becomes something that *inhibits performance*. A common, misguided technique that many try in order to solve this issue is to add more space to the shared pool, but this typically only makes things worse than before. As the shared pool inevitably fills up once again, it gets to be even *more* of an overhead than the smaller shared pool, for the simple reason that managing a big, full-shared pool takes more work than managing a smaller full-shared pool.

The only true solution to this problem is to utilize shared SQL – to reuse queries. Later, in Chapter X on *Tuning Strategies and Tools*, we will take a look at the parameter CURSOR_SHARING that can work as a short term 'crutch' in this area, but the only real way to solve this issue is to use reusable SQL in the first place. Even on the largest of large systems, I find that there are typically at most 10,000 to 20,000 unique SQL statements. Most systems execute only a few hundred unique queries.

The following real-world example demonstrates just how bad things can get if you use of the shared pool poorly. I was asked to work on a system where the standard operating procedure was to shut down the database each and every night, in order to wipe out the SGA, and restart it clean. The reason for doing this was that the system was having issues during the day whereby it was totally CPU-bound and, if the database were left to run for more than a day, performance would really start to decline. This was solely due to the fact that, in the time period from 9am to 5pm, they would fill a 1 GB shared pool inside of a 1.1 GB SGA. This is true – 0.1 GB dedicated to block buffer cache and other elements, and 1 GB dedicated to caching unique queries that would never be executed again. The reason for the cold start was that if they left the system running for more than a day, they would run out of free memory in the shared pool. At that point, the overhead of aging structures out (especially from a structure so large) was such that it overwhelmed the system and performance was massively degraded (not that performance was that great anyway, since they were managing a 1 GB shared pool). Additionally, the people working on this system constantly wanted to add more and more CPUs to the machine, due to the fact that hard parsing SQL is so CPU-intensive. By correcting the application, allowing it to use bind variables, not only did the physical machine requirements drop (they then had many times more CPU power then they needed), the memory utilization was reversed. Instead of a 1 GB shared pool, they had less then 100 MB allocated – and never used it all over many weeks of continuous uptime.

One last comment about the shared pool and the parameter, SHARED_POOL_SIZE. In Oracle 9i and before, there is no direct relationship between the outcome of the query:

```
ops$tkyte@ORA9IR2> select sum(bytes) from v$sgastat where pool = 'shared pool';

SUM(BYTES)
----------
 100663296
```

and the SHARED_POOL_SIZE parameter:

```
ops$tkyte@ORA9IR2> show parameter shared_pool_size
```

```
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
shared_pool_size                     big integer 83886080
```

other than the fact that the SUM(BYTES) FROM V$SGASTAT will always be larger than the SHARED_POOL_SIZE. The shared pool holds many other structures that are outside the scope of the corresponding parameter. The SHARED_POOL_SIZE is typically the largest contributor to the shared pool as reported by the SUM(BYTES), but it is not the only contributor. For example, the parameter, CONTROL_FILES, contributes 264 bytes per file to the 'miscellaneous' section of the shared pool. It is unfortunate that the 'shared pool' in V$SGASTAT and the parameter SHARED_POOL_SIZE are named as they are, since the parameter contributes to the size of the shared pool, but it is not the *only* contributor.

In Oracle 10g and above however, you should see a direct one to one correspondence between the two, assuming you are using manual SGA memory management (that is, you have set the shared_pool_size parameter yourself):

```
ops$tkyte@ORA10G> select sum(bytes)/1024/1024 mbytes
  2  from v$sgastat where pool = 'shared pool';

    MBYTES
----------
       128

ops$tkyte@ORA10G> show parameter shared_pool_size;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
shared_pool_size                     big integer 128M
```

This is a relatively important change as you go from Oracle 9i and before to 10g. In Oracle 10g, the shared_pool_size parameter controls the size of the shared pool, whereas in Oracle 9i and before, it was just the largest contribution to the shared pool. You would want to review your 9i and before actual shared pool size (based on v$sgastat) and use that figure to set your shared_pool_size parameter in Oracle 10g and above. The various other components that used to add onto the size of the shared pool now expect that memory to have been allocated for them by you.

Large Pool

The large pool is not so named because it is a 'large' structure (although it may very well be large in size). It is so named because it is used for allocations of large pieces of memory, bigger than the shared pool is designed to handle. Prior to its introduction in Oracle 8.0, all memory allocation took place in the shared pool. This was unfortunate if you were using features that made use of 'large' memory allocations such as MTS. This issue was further confounded by the fact that processing, which tended to need a lot of memory allocation, would use the memory in a different manner to the way in which the shared pool managed it. The shared pool manages memory in a LRU basis, which is perfect for caching and reusing data. Large memory allocations, however, tended to get a chunk of memory, use it, and then were done with it – there was no need to cache this memory.

What Oracle needed was something similar to the RECYCLE and KEEP buffer pools implemented for the block buffer cache. This is exactly what the large pool and shared pool are now. The large pool is a RECYCLE-style memory space whereas the shared pool is more like the KEEP buffer pool – if people appear to be using something frequently, then you keep it cached.

Memory allocated in the large pool is managed in a heap, much in the way C manages memory via malloc() and free(). As soon as you 'free' a chunk of memory, it can be used by other processes. In the shared pool, there really was no concept of 'freeing' a chunk of memory. You would allocate memory, use it, and then stop using it. After a while, if that memory needed to be reused, Oracle would age out your chunk of memory. The problem with using just a shared pool is that one size doesn't always fit all.

The large pool is used specifically by:

* *Shared Server Connections* – to allocate the UGA region in the SGA.
* *Parallel Execution of statements* – to allow for the allocation of inter-process message buffers, used to coordinate the parallel query servers.
* *Backup* – for RMAN disk I/O buffers in some cases.

As you can see, none of the above memory allocations should be managed in an LRU buffer pool designed to manage small chunks of memory. With shared server connection memory, for example, once a session logs out, this memory is never going to be reused so it should be immediately returned to the pool. Also, shared server UGA memory allocations tends to be 'large'. If you review our earlier examples, with the SORT_AREA_RETAINED_SIZE or PGA_AGGREGATE_TARGET, the UGA can grow very large, and is definitely bigger than 4 KB chunks. Putting MTS memory into the shared pool causes it to fragment into odd sized pieces of memory and, furthermore you will find that large pieces of memory that will never be reused will age out memory that could be reused. This forces the database to do more work to rebuild that memory structure later.

The same is true for parallel query message buffers. Once they have delivered their message, they are no longer needed. Backup buffers, even more so – they are large, and once Oracle is done using them, they should just 'disappear'.

The large pool is not mandatory when using shared server connections, but it is highly recommended. If you do not have a large pool and use shared server connection, the allocations come out of the shared pool as they always did in Oracle 7.3 and before. This will definitely lead to degraded performance over some period of time, and should be avoided. The large pool will default to some size if the parameter DBWR_IO_SLAVES is set to some positive value or parallel query is enabled are set. It is recommended that you set the size of the large pool manually yourself if you are using a feature that utilizes it. The default mechanism is typically not the appropriate value for your situation.

Java Pool

The Java pool was added in version 8.1.5 of Oracle to support the running of Java in the database. If you code a stored procedure in Java, Oracle will make use of this chunk of memory when processing that code. The parameter JAVA_POOL_SIZE is used to fix the amount of memory allocated to the Java pool for all session-specific Java code and data.

The Java pool is used in different ways, depending on the mode in which the Oracle server is running. In dedicated server mode, the Java pool includes the shared part of each Java class, which is actually used per session. These are basically the read-only parts (execution vectors, methods, and so on) and are about 4 to 8 KB per class.

Thus, in dedicated server mode (which will most likely be the case for applications using purely Java stored procedures), the total memory required for the Java pool is quite modest and can be determined based on the number of Java classes you will be using. It should be noted that none of the per-session state is stored in the SGA in dedicated server mode, as this information is stored in the UGA and, as you will recall, the UGA is included in the PGA in dedicated server mode.

When connecting to Oracle using a shared server connection, the Java pool includes:

* The shared part of each Java class, *and*
* Some of the UGA used for per-session state of each session, which is allocated from the java_pool within the SGA. The remainder of the UGA will be located as normal in the shared pool, or if the large pool is configured it will be allocated there instead.

As the total size of the Java pool is fixed in Oracle 9i and before, application developers will need to estimate the total requirement of their applications, and multiply this by the number of concurrent sessions they need to support. This number will dictate the overall size of the Java pool. Each Java UGA will grow or shrink as needed, but bear in mind that the pool must be sized such that all UGAs combined must be able to fit in at the same time. In Oracle 10g and above, this parameter may be modified and the Java pool may grow and shrink over time without the database being restarted.

Streams Pool

The streams pool is a new SGA structure starting in Oracle 10g. Streams itself is a new database feature as of Oracle 9iR2 and above. It was designed as a data sharing/replication tool and is Oracle's stated direction going forward for data replication.

NOTE: The going forward statement should not be interpreted as meaning that advanced replication, Oracle's now legacy replication feature, is going away any time soon. Rather, Advanced Replication will continue to be supported in future releases. Also, to learn more about Streams itself, you would want to see the Streams Concepts Guide, available on http://otn.oracle.com/ in the documentation section.

The streams pool (or up to 10% of the shared pool if no streams pool is configured) is used to buffer queue messages used by the streams process as it is moving/copying data from one database to another. Instead of using permanent disk based queues, with the attendant overhead associated with them, Streams uses in memory queues. If these queues fill up, they will spill over to disk eventually. If the Oracle instance with the memory queue fails for some reason, due to an instance failure (software crash), power failure or whatever – these in memory queues are rebuilt from the redo logs.

So, the streams pool will only be important in systems using the Streams database feature and in those environments should be set in order to avoid 'stealing' 10% of the shared pool for this feature.

Automatic SGA memory management

Just as there are two ways to manage PGA memory, there are two ways to manage SGA memory starting in Oracle 10g – manually, by setting all of the necessary 'pool' and 'cache' parameters and automatic – by setting just a few memory parameters and a single SGA_TARGET parameter. By setting the SGA_TARGET, you are allowing the instance to size, and resize various SGA components.

NOTE: In Oracle 9i and before, there was only manual SGA memory management

---

In Oracle 10g, memory related parameters have been classified into one of two areas:

* *Auto-Tuned SGA Parameters*: Current these are db_cache_size, shared_pool_size, large_pool_size and java_pool_size
- *Manual SGA Parameters*: These include log_buffer, streams_pool, db_Nk_cache_size, db_keep_cache_size, and db_recycle_cache_size.

NOTE: In order to use automatic SGA memory management, the parameter statistics_level must be set to TYPICAL or ALL.

Under automatic SGA memory management, the primary parameter for sizing the auto-tuned components is the SGA_TARGET parameter, which may be dynamically sized while the database is up and running, up to the setting of the SGA_MAX_SIZE parameter (which defaults to be equal to the SGA_TARGET, so you plan on increasing the SGA_TARGET, you must have set the SGA_MAX_SIZE larger before starting the database instance). The database will use the SGA_TARGET value, minus the size of any of the other manually sized components such as the db_keep_cache_size, db_recycle_cache_size and so on – and use that amount of memory to size the default buffer pool, the shared pool, large pool and java pool. Dynamically at runtime, the instance will allocate and reallocate memory between those 4 memory areas as needed. Instead of returning an ORA-04031 "Unable to allocate N bytes of shared memory" to a user when the shared pool runs out of memory, the instance could instead choose to shrink the buffer cache by some number of megabytes (a granule size) and increase the shared pool by that amount.

Over time, as the memory needs of the instance are ascertained, the size of the various SGA components would become more or less fixed in size. The database also remembers the sizes of these 4 components across database startup and shutdown so that it doesn't have to start all over again figuring out the right size for your instance each time. It does this via 4 double underscore parameters: __db_cache_size, __java_pool_size, __large_pool_size, and __shared_pool_size. During a normal or immediate shutdown, the database will record these values to the stored parameter file and use them upon startup to set the default sizes of each area.

Additionally, if you know you want a certain minimum value to be used for one of the 4 areas, you may set that parameter in addition to setting the SGA_TARGET. The instance will uses your setting as the lower bound, the smallest that particular area may be.

Memory Structures Wrap-Up

In this section, we have taken a look at the Oracle memory structure. We started at the process and session level, taking a look at the PGA (Process Global Area) and UGA (User Global Area), and their relationship to each other. We have seen how the mode in which you connect to Oracle will dictate how memory is organized. A dedicated server connection implies more memory used in the server process than under shared server connection, but that use of shared server connections implies there will be the need for a significantly larger SGA. Then, we discussed the components of the SGA itself, looking at its main structures. We discovered the differences between the shared pool and the large pool, and looked at why you might want a large pool in order to 'save' your shared pool. We covered the Java pool and how it is used under various conditions. We looked at the block buffer cache and how that can be subdivided into smaller, more focused pools.

Now we are ready to move onto the physical processes that make up the rest of an Oracle instance.