RAMPANT TECHPRESS

Tuning Third-party Vendor Oracle Systems

Tuning when you can't touch the code

Mike Ault

Retail Price $12.95 US/$19.95 Canada
ISBN: 0-9740716-3-3

RAMPANT TECHPRESS
eBook

Oracle eBook

Rampant TechPress

Tuning Third-party Vendor Oracle systems

Tuning when you can't touch the code#

#

#

#

#

#

Mike Ault

Notice

While the author & Rampant TechPress makes every effort to ensure the information presented in this white paper is accurate and without error, Rampant TechPress, its authors and its affiliates takes no responsibility for the use of the information, tips, techniques or technologies contained in this white paper. The user of this white paper is solely responsible for the consequences of the utilization of the information, tips, techniques or technologies reported herein. .

Tuning Third-party Vendor Oracle systems

Tuning when you can't touch the code#

By Mike Ault

Published by Rampant TechPress, Kittrell, North Carolina, USA

Series Editor: Don Burleson

Production Editor: Teri Wade

Cover Design: Bryan Hoff

Visit **www.rampant.cc** for information on other *Oracle In-Focus* books.

ISBN: 0-9740716-3-3

Table Of Contents

Introduction

In many Oracle shops today third-party applications are the norm. The major problem for DBAs with these third-party applications is that you are not allowed to alter the source code of the SQL used within the application. Many times the application will generate SQL statements in an ad-hoc manner that further complicates the tuning picture. This paper will attempt to provide insights into how to tune Oracle when you can't touch the code.

Tuning Overview

Everyone who has been in the Oracle DBA profession for any length of time has seen the graph in figure 1. This graph shows the percentage gains, on the average, from tuning various aspects of the Oracle database environment.

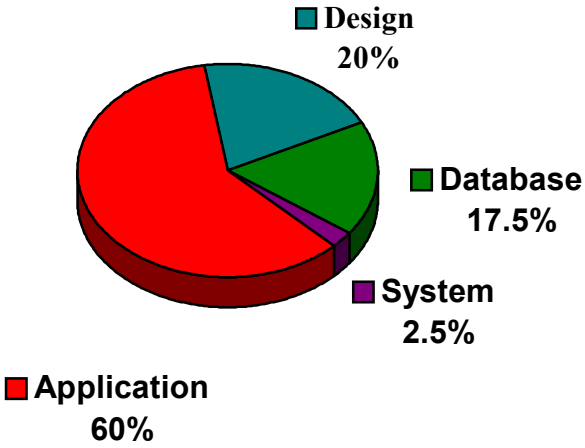


Figure 1: Performance Gains from Tuning

As can be seen from a quick glance at the graph, 80% of tuning gains are realized from proper design and application statement tuning. Unfortunately in a

third-party application such as those provided by SAP, PeopleSoft, Baen, Siebel or Oracle Financials, the DBA is often forced to ignore bad design and SQL since touching the code is forbidden. This leaves us with the 20% of gains that can be reached through the tuning of the database and the system.

However, it should be noted that the graph in figure 1 is not applicable to all cases and carries many unseen qualifications with it. The graph assumes that the system and database have been set up by a reasonably qualified Oracle DBA. Of course this is not always the case and in many locations a qualified Oracle DBA isn't hired until performance problems manifest themselves, this is usually just as the system goes live and a full user load is experienced.

What Can Be Done?

Depending on the Oracle version there are tuning options available to the DBA that don't involve tweaking the SQL. Table 1 shows the main tuning options available by Oracle version.

Oracle Version:	**7.3.x**	**8.0.x**	**8.1.x**
Optimize Internals	X	X	X
Optimizer Modes	X	X	X
Add Resources	X	X	X
Tune Tables and Indexes	X	X	X
Parallel Query	X	X	X
Better Indexes		X	X
Index Only Tables		X	X
Partitioning		X	X
New INI features		X	X
Subpartitioning			X
Outlines			X
Resource Groups			X

Table 1: Tuning Options by Oracle Version

As it should be expected, as the version increases so do the various tuning options available to the DBA. This indicates that the DBA should always press to be on the latest, stable version of Oracle (7.3.4.2, 8.0.6.2.2, 8.1.7.) Let's examine these tuning options and see how they can be applied to your databases. As we cover the options an attempt will be made to show how the option is applied per version as the feature implementations change as Oracle matures.

Optimizing Oracle Internals

When beginning to tune a third-party database where the code can't be touched you should generally begin with making sure that the memory and database internals are optimized for performance. If Oracle doesn't have enough memory, processes or other resources, the other tuning options won't make much difference generally speaking. The options for internals tuning are:

- Database Buffer Tuning
- Database Writer Tuning
- Shared Pool Tuning
- Checkpoints
- Redo Logs
- Rollback Segments
- Sort Area Size

Let's examine each of these areas.

Database Buffer Tuning

When we refer to database buffer tuning we are actually discussing the tuning of the memory used to store data used by Oracle processes. All data that passes to users and then back to the database passes through buffers. If there aren't enough db block buffers there is a significant hit on performance. Likewise if the database base block buffers aren't of the correct size then they can't be efficiently utilized.

Generally it is suggested that the database block buffer size be set to at least 8192 (8k). This size of 8k allows for optimal storage of data and index information on most Oracle platforms. The product of db_block_size and db_block_buffers should be no less than 5-10% of the total physical data size (including indexes) for the system. Usually the product of db_block_size and db_block_buffers will be larger than 5-10% of the physical database size, but this is a good general starting point. Of course the size of the buffer area and other shared global area components, should not exceed 50-60% of the available physical memory or swapping will result.

One gross indicator of database buffer health is called the hit ratio. The hit ratio is expressed as a percent and is calculated using the formula:

(1-(physical reads/(db block gets+consistent gets))) * 100

Traditionally the information for calculating the database block buffers hit ratio is taken from the V$SYSSTATS view. However, in versions 7.3.4 and higher of the database the "physical reads" parameter was altered to include "direct reads" which skews the hit ratio in the downward direction. In Oracle8i the statistic "direct reads" is also recorded so you can subtract the "direct reads" from the "physical reads" to get the correct value with which to calculate hit ratio. However, Oracle has provided the V$BUFFER_POOL_STATISTICS view if the DBA runs the CATPERF.SQL script in the latest releases in which uncontaminated values for "physical reads" are available and this view should be used where it is available.

Hit ratio should generally be as close to 100% as is possible to achieve, however, in some cases artificially high values can be received if nonselective indexes are used in queries. Hit ratio is not the best indicator of performance of the database block buffers.

It is suggested that hit ratio be monitored to give a quick look at performance, however tuning decisions should be made on a more detailed analysis of the buffer area. Using cursors PL/SQL can be used to track hit ratios as is shown in figure 2.

```
CURSOR get_stat(stat IN VARCHAR2) IS
  SELECT name,value FROM v$sysstat
  WHERE name = stat;
Supply the cursor with the variables:

'db block gets‘,'consistent gets‘, 'physical reads‘, ‘direct reads’
h_ratio := (1-(p_reads-d_reads)/(db_gets + con_gets)))*100;
Or use the cursor:

CURSOR get_hratio IS
  SELECT name, (1-(physical_reads/(db_block_gets+consistent_gets)))*100
H_RATIO
  FROM v$buffer_pool_statistics;
```

Notice the cursor returns a pool name as well, in Oracle8 and above multiple buffer pools are allowed.

Figure 2: Example Hit Ratio Calculations

More detailed information about the database blcok buffers is contained in the V$BH view. The V$BH view of the X$BH table is available in newer versions of Oracle. In earlier versions the view had to be created using the CATPARR.SQL script.

The X$BH view contains information on the buffers in the database block buffers and their states. The state information contained in X$BH should be utilized to get a true picture of what is happening with the database block buffers. An

example select, from: "ORACLE Performance Tuning Tips & Techniques", Richard Niemiec, Oracle Press, is shown in figure 3.

```
CREATE VIEW BLOCK_STATUS AS
SELECT DECODE(state, 0, 'FREE',
       1, DECODE(lrba_seq,0, 'AVAILABLE', 'BEING USED'),
       3, 'BEING USED', state) "BLOCK STATUS",
       COUNT(*) "COUNT"
 FROM x$bh
 GROUP BY
 decode(state,0,'FREE',1,decode(lrba_seq,0,'AVAILABLE',
     'BEING USED'),3,'BEING USED',state);
```

Figure 3: Example X$BH Select

If 10-25% buffers are free after 2 hours of use, good. If your database doesn't show at least 10-25% of the database block buffers free, then you should consider increasing the value of DB_BLOCK_BUFFERS in 10-25% increments. An alternative select using the X$BH from NOTE:1019635.6 on Metalink is shown in figure 4.

```
create view buffer_status2 as select
decode(greatest(class,10),10,decode(class,1,'Data',2
           ,'Sort',4,'Header',to_char(class)),'Rollback') "Class",
       sum(decode(bitand(flag,1),1,0,1)) "Not Dirty",
 sum(decode(bitand(flag,1),1,1,0)) "Dirty",
       sum(dirty_queue) "On Dirty",count(*) "Total"
from x$bh
group by decode(greatest(class,10),10,decode(class,1,'Data',2
         ,'Sort',4,'Header',to_char(class)),'Rollback')
/
```

Figure 4: Example Select Against X$BH From Metalink

One thing to note about the scripts in Figures 3 and 4 is that they must be run from the SYS user, both create views that can then be used by other users with appropriate grants.

Another source of information about possible database block buffer problems is the V$WAITSTAT view that summarizes the counts of the various wait conditions occurring in the database. Figure 5 shows an example select against this view.

```
SELECT
       class,"COUNT"
  FROM
       v$waitstat
  WHERE
       class = 'data block';
```

Figure 5: Example V$WAITSTAT Select

It must be stated that data block waits by themselves do not indicate that an increase in database block buffers is required. Data block waits can also be caused by improperly set INITRANS and FREELISTS on heavily used tables. However, in my experience a major portion of data block waits are directly attributable to insufficient database block buffers in systems where a significant number of data block waits are experienced (100 waits is not significant, 10000 are.) If you have high hit ratios (in the high 90's) and experience data block waits with the V$BH view showing 10-25% free buffers, then the waits are probably due to INITRANS and FREELISTS, otherwise they point at insufficient database block buffers.

Using the techniques discussed the DBA should be able to properly tune the size of the DB_BLOCK_BUFFERS parameter to ensure adequate memory is available for the databases data needs. As with virtually all other tuning aspects, the setting for DB_BLOCK_BUFFERS will have to adjusted as the amount of data in the database increases or decreases and the user data requirements change.

Database Writer Tuning

Database writer tuning involves two basic areas, first, how often writes are accomplished and how much is written in each write and second, how many writer processes are designated to service the database output requirements. The V$SYSSTAT view should also be used to calculate the value for the average length of the dirty write queue, values larger than 100 show need for more DB_BLOCK_BUFFERS or DB_WRITERS or a need to increase the size of the DB_BLOCK_WRITE_BATCH (which becomes an undocumented parameter beginning with Oracle8.)

Figure 6 shows a select taken from "Oracle Performance Tuning" , Mark Gurry and Peter Corrigan, O'Reilly Press.

```
SELECT
DECODE (name, 'summed dirty write queue length', value)/
DECODE (name, 'write requests', value) "Write Request Length"
FROM v$sysstat
WHERE name IN ( 'summed dirty queue length', 'write requests') and
value>0;
```

Figure 6: Example Select for Dirty Queue Length

The parameters that govern the behavior and number of database writer processes are shown in table 2.

Parameter	Description
In Oracle 7:	
DB_WRITERS (2 x #disks)	Sets number of DBWR processes
DB_BLOCK_BUFFERS	Sets number of buffers
DB_BLOCK_CHECKPOINT_BATCH	Number of blocks written per batch during checkpoint (Obsolete in 8i)
_DB_BLOCK_WRITE_BATCH	Sets number of buffers written per IO
_DB_BLOCK_MAX_SCAN_CNT	Sets number of blocks scanned before a write is triggered
DISK_ASYNC_IO	Allows asynchronous IO
DB_FILE_SIMULTANEOUS_WRITES	Number of simultaneous writes to a file
In Oracle 8.0:	
DBWR_IO_SLAVES (2 x #disks)	Same as DB_WRITERS
DB_FILE_DIRECT_IO_COUNT	Number of blocks assigned to BU and REC buffers as well as direct IO buffers
In Oracle8i:	
DB_WRITER_PROCESSES (2 x #disks)	Same as DB_WRITERS
DBWR_IO_SLAVES	Sets number of slave DBWR processes
DB_FILE_DIRECT_IO_COUNT	Number of blocks assigned to BU and REC buffers as well as direct IO buffers
DB_BLOCK_LRU_LATCHES	Sets number of LRU latches
DB_BLOCK_MAX_DIRTY_TARGET	Sets target limit of dirty buffers
Many more "_" parameters	

Table 2: Initialization Parameters for DBWR Tuning (Duplicate parameters removed)

Whether you use DB_WRITERS, DBWR_IO_SLAVES or DB_WRITER_PROCESSES usually you won't need more than 2 processes per disk used for Oracle. Generally speaking if you exceed twice your number of CPUs for the number of DBWR processes you will get diminishing returns. In Oracle8i if you have multiple DB_WRITER_PROCESSES you can't have multiple DBWR_IO_SLAVES. You must also have at least one DBWR_BLOCK_LRU_LATCH for each DBWR process. If you set DBWR_IO_SLAVES in Oracle8i then the values for ARCH_IO_SLAVES and LGWR_IO_SLAVES are set to 4 each and DB_WRITER_PORCESSES is set to 1 silently.

DB_BLOCK_BUFFERS has already been discussed.

The undocumented parameters (those preceded by an "_" underscore probably shouldn't be reset. In some cases reducing the value of _DB_BLOCK_WRITE_BATCH may reduce waits for the DBWR processes.

DB_BLOCK_CHECKPOINT_BATCH sets the number of blocks the database writer process(es) write with each checkpoint write. A small value allows threading of other writes but causes longer checkpoint times. A large value gets checkpoints completed faster but holds up other writes. If you set this value to high Oracle will silently set it to the value of the database writer write batch.

DB_BLOCK_MAX_DIRTY_TARGET specifies the number of buffers that are allowed to be dirty before DBRW will write them all out to disk. This limits the required time for instance recovery after a crash but low values will cause DBRW to perform extra work.

DB_FILE_SIMULTANEOUS_WRITES should be set to 4 times the number of disks in your stripe sets. When striping is not used set it to 4.

DISK_ASYNC_IO is only used when asynchronous writes are not stable on your system. Generally DISK_ASYNC_IO defaults to TRUE only set it to false if the previously mentioned condition is true. If you must set DISK_ASYNC_IO to FALSE, configure multiple DBRW or DBRW_IO_SLAVES to simulate asynchronous IO.

One indication of DBWR problems is excessive BUFFER WAITS from V$WAITSTAT. You can check this with a look at buffer waits from Gurry and Corrigan:

```
SELECT name, value FROM v$sysstat
WHERE name='free buffer waits';
```

Shared Pool Tuning

Perhaps one of the least understood areas of Oracle Shared Global Area optimization is tuning the shared pool. The generally accepted tuning methodology involves throwing memory into the pool until the problem goes under. In this section of the paper we will examine the shared pool and define a method for tuning the shared pool that uses measurement, not guesswork to drive the tuning methodologies.

What is the shared pool?

Many people know that the shared pool is a part of the Oracle shared global area (SGA) but little else, what exactly is the shared pool? The shared pool contains several key Oracle performance related memory areas. If the shared pool is

improperly sized then overall database performance will suffer, sometimes dramatically. Figure 7 diagrams the shared pool structure located inside the various Oracle SGAs.

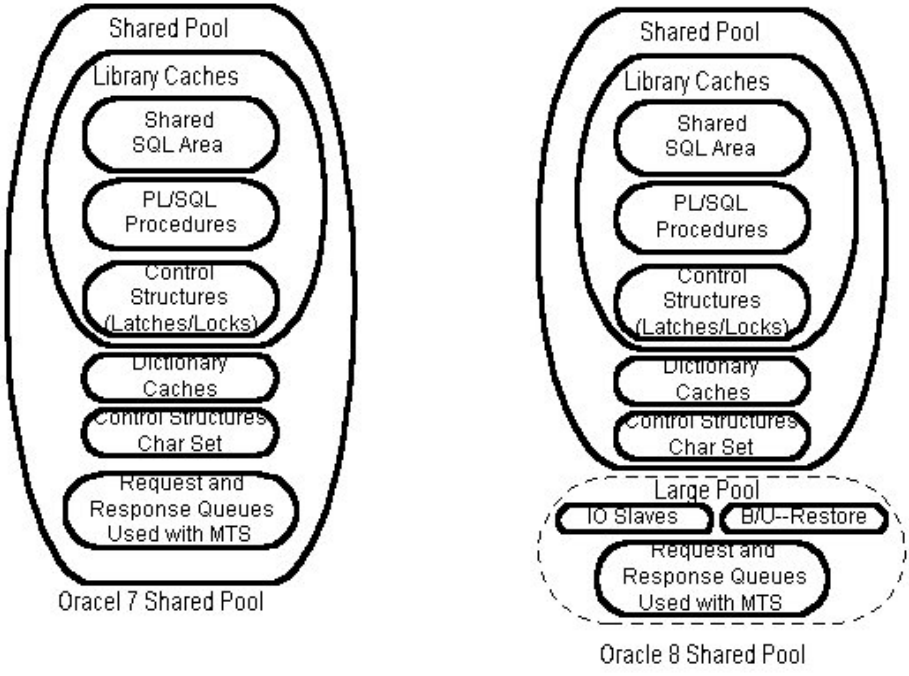


Figure 7: Oracle 7 and Oracle 8 Shared Pool Structures

As you can see from examining the structures pictured in Figure 7, the shared pool is separated into many substructures. The substructures of the shared pool fall into two broad areas, the fixed size areas that for a given database at a given point in time stay relatively constant in size and the variable size areas that grow and shrink according to user and program requirements.

In Figure 7 the areas inside the library caches substructure are variable in size while those outside the library caches (with the exception of the request and response queues used with MTS) stay relatively fixed in size. The sizes are determined based on an Oracle internal algorithm that ratios out the fixed areas based on overall shared pool size, a few of the intialization parameters and empirical determinations from previous versions. In early versions of Oracle (notably 6.2 and lower versions) the dictionary caches could be sized individually allowing a finer control of this aspect of the shared pool. With Oracle 7 the internal algorithm for sizing the data dictionary caches took control from the DBA.

The shared pool is used for objects that can be shared among all users such as table definitions, reusable SQL (although non-reusable SQL is also stored there), PL/SQL packages, procedures and functions. Cursor information is also stored in the shared pool. At a minimum the shared pool must be sized to accommodate the needs of the fixed areas plus a small amount of memory reserved for use in parsing SQL and PL/SQL statements or ORA-07445 errors will result.

Monitoring and Tuning the Shared Pool

Let me begin this section by stating that the default values for the shared pool size initialization parameters are almost always too small by at least a factor of four. Unless your database is limited to the basic scott/tiger type schema and your overall physical data size is less than a couple of hundred megabytes, even the "large" parameters are far too small. What parameters control the size of the shared pool? Essentially only one, SHARED_POOL_SIZE. The other shared pool parameters control how the variable space areas in the shared pool are parsed out, but not overall shared pool size. In Oracle8 a new area, the large pool, controlled by the LARGE_POOL_SIZE parameter is also present. Generally speaking I suggest you start at a shared pool size of 40 megabytes and move up from there. The large pool size will depend on the number of concurrent users, number of multi-threaded server servers and dispatchers and the sort requirements for the application. Sizes of larger than 140-200 megabytes rarely result in performance improvement. The major problem with the shared pool is over population resulting in too many SQL areas to be efficiently managed. Usually when you exceed 5000-7000 SQL areas performance in the shared pool tends to degrade.

What should be monitored to determine if the shared pool is too small? For this you need to wade into the data dictionary tables, specifically the V$SGASTAT and V$SQLAREA views. Figure 8 shows a report that shows how much of the shared pool is in use at any given time the script is run.

```
REM Script to report on shared pool usage
REM
column shared_pool_used  format 9,999.99
column shared_pool_size  format 9,999.99
column shared_pool_avail format 9,999.99
column shared_pool_pct   format 999.99
@title80 'Shared Pool Summary'
spool rep_out\&db\shared_pool
select
  least(max(b.value)/(1024*1024),sum(a.bytes)/(1024*1024))
shared_pool_used,
  max(b.value)/(1024*1024) shared_pool_size,
  greatest(max(b.value)/(1024*1024),sum(a.bytes)/(1024*1024))-
(sum(a.bytes)/(1024*1024)) shared_pool_avail,
  ((sum(a.bytes)/(1024*1024))/(max(b.value)/(1024*1024)))*100
avail_pool_pct
   from v$sgastat a, v$parameter b
```

```
 where (a.pool='shared pool'
 and a.name not in ('free memory'))
 and
 b.name='shared_pool_size';
 spool off
ttitle off
```

Figure 8: Example Script to Show SGA Usage

The script in Figure 8 should be run periodically during times of normal and high usage of your database. The results will be similar to Figure 9. If your shared_pool_pct figures stay in the high nineties then you may need to increase the size of your shared pool, however, this isn't always the case.

```
Date: 11/18/98                                                    Page:    1
Time: 04:16 PM              Shared Pool Summary             SYSTEM
                              ORTEST1 database

SHARED_POOL_USED SHARED_POOL_SIZE SHARED_POOL_AVAIL SHARED_POOL_PCT
---------------- ---------------- ----------------- ---------------
            3.66            38.15             34.49            9.60
```

Figure 9: Example Output From Script In Figure 8.

To often all that is monitored is how much of the shared pool is filled, no one looks how is it filled; with good reusable SQL or bad throw away SQL. You must examine how the space is being used before you can decide whether the shared pool should be increased in size, decreased in size or perhaps a periodic flush schedule set up with the size remaining the same. So how can we determine what is in the shared pool and whether it is being properly reused or not? Let's look at a few more reports.

The first report we will examine shows how individual users are utilizing the shared pool. Before we can run the report a summary view of the V$SQLAREA view must be created, I unimaginatively call this view the SQL_SUMMARY view. The code for the SQL_SUMMARY view is shown in Figure 10.

```
rem FUNCTION: Creates summary of v_$sqlarea and dba_users for use in
rem           sqlmem.sql and sqlsummary.sql reports
rem
rem
create or replace view sql_summary as
select
username, sharable_mem, persistent_mem, runtime_mem
from
sys.v_$sqlarea a, dba_users b
where
a.parsing_user_id = b.user_id;
rem
```

Figure 10: Example SQL Script to Create A View to Monitor Pool Usage By User

Once the SQL_SUMMARY view is created the script in Figure 11 is run to generate a summary report of SQL areas used by user. This shows the distribution of SQL areas and may show you that some users are hogging a disproportionate amount of the shared pool area. Usually, a user that is hogging a large volume of the shared pool is not using good SQL coding techniques which is generating a large number of non-reusable SQL areas.

```
rem
rem FUNCTION: Generate a summary of SQL Area Memory Usage
rem FUNCTION: uses the sqlsummary view.
rem           showing user SQL memory usage
rem
rem sqlsum.sql
rem
column areas                              heading Used|Areas
column sharable   format 999,999,999      heading Shared|Bytes
column persistent format 999,999,999      heading Persistent|Bytes
column runtime    format 999,999,999      heading Runtime|Bytes
column username   format a15              heading "User"
column mem_sum    format 999,999,999  heading Mem|Sum
start title80 "Users SQL Area Memory Use"
spool rep_out\&db\sqlsum
set pages 59 lines 80
break on report
compute sum of sharable on report
compute sum of persistent on report
compute sum of runtime on report
compute sum of mem_sum on report
select
username,
sum(sharable_mem) Sharable,
sum( persistent_mem) Persistent,
sum( runtime_mem) Runtime ,
count(*) Areas,
sum(sharable_mem+persistent_mem+runtime_mem) Mem_sum
from
sql_summary
group by username
order by 2;
spool off
```

```
pause Press enter to continue
clear columns
clear breaks
set pages 22 lines 80
ttitle off
```

Figure 11: Example SQL Script To Report On SQL Area Usage By User

Example output from the script in Figure11 is shown in Figure 12. In the example report no one user is really hogging the SQL area. If you have a particular user that is hogging SQL areas, the report in Figure 12 will show you what SQL areas they have and what is in them. This report on the actual SQL area contents can then be used to help teach the user how to better construct reusable SQL statements.

```
Date: 11/18/98                                                    Page:   1
Time: 04:18 PM              Users SQL Area Memory Use             SYSTEM
                                 ORTEST1 database

                       Shared   Persistent      Runtime      Used
Mem
User                    Bytes        Bytes        Bytes     Areas
Sum
--------------- ------------ ------------ ------------ --------- ---------
---
GRAPHICS_DBA           67,226        4,640       30,512        10
102,378
SYS                   830,929       47,244      153,652        80
1,031,825
SYSTEM              2,364,314       37,848      526,228        63
2,928,390
                ------------ ------------ ------------ --------- ---------
---
sum                 3,262,469       89,732      710,392       153
4,062,593

3 rows selected.
```

Figure 12: Example Output From Figure 11

In the example output we see that SYSTEM user holds the most SQL areas and our application DBA user, GRAPHICS_DBA holds the least. Since these reports where run on my small Oracle 8.0.5 database this is normal, however, usually the application owner will hold the largest section of memory in a well designed system, followed by ad-hoc users using properly designed SQL. In a situation where users aren't using properly designed SQL statements the ad-hoc users will usually have the largest number of SQL areas and show the most memory usage. Again, the script in Figure 13 shows the actual in memory SQL areas for a specific user. Figure 14 shows the example output from a report run against GRAPHICS_USER using the script in Figure 13.

```
rem
rem FUNCTION: Generate a report of SQL Area Memory Usage
rem           showing SQL Text and memory catagories
rem
rem sqlmem.sql
rem
column sql_text      format a60   heading Text word_wrapped
column sharable_mem               heading Shared|Bytes
column persistent_mem             heading Persistent|Bytes
column loads                      heading Loads
column users         format a15   heading "User"
column executions                       heading "Executions"
column users_executing            heading "Used By"
start title132 "Users SQL Area Memory Use"
spool rep_out\&db\sqlmem
set long 2000 pages 59 lines 132
break on users
compute sum of sharable_mem on users
compute sum of persistent_mem on users
compute sum of runtime_mem on users
select
username users, sql_text, Executions, loads, users_executing,
sharable_mem, persistent_mem
from
sys.v_$sqlarea a, dba_users b
where
a.parsing_user_id = b.user_id
and b.username like upper('%&user_name%')
order by 3 desc,1;
spool off
pause Press enter to continue
clear columns
clear computes
clear breaks
set pages 22 lines 80
```

Figure 13: Example Script To Show Active SQL Areas For a User

```
Date:                                                                                                    11/18/98
Page:   1
Time:  04:19  PM                                                                  Users  SQL  Area  Memory  Use
SYSTEM
                                                  ORTEST1 database


Shared Persistent
User           Text                                                                  Executions  Loads Used
By  Bytes      Bytes
-------------- ------------------------------------------------------------------ ---------- ------ -----
-- ------ ----------
GRAPHICS_DBA     BEGIN dbms_lob.read (:1, :2, :3, :4); END;                             2121       1
0  10251        488
                 alter session set nls_language= 'AMERICAN' nls_territory=                 7       1
0   3975        408
                 'AMERICA' nls_currency= '$' nls_iso_currency= 'AMERICA'
                 nls_numeric_characters= '.,' nls_calENDar= 'GREGORIAN'
                 nls_date_format= 'DD-MON-YY' nls_date_language= 'AMERICAN'
                 nls_sort= 'BINARY'
                   BEGIN :1 := dbms_lob.getLength (:2); END;                               6       1
0   9290        448
                   SELECT TO_CHAR(image_seq.nextval) FROM dual                           6       1
0   6532        484
```

```
                SELECT graphic_blob FROM internal_graphics WHERE                        2       1
0   5863        468
                graphic_id=10
                SELECT RPAD(TO_CHAR(graphic_id),5)||':                                  1       1
0   7101        472
                '||RPAD(graphic_desc,30)||' : '||RPAD(graphic_type,10) FROM
                internal_graphics ORDER BY graphic_id
                SELECT graphic_blob FROM internal_graphics WHERE                        1       1
0   6099        468
                graphic_id=12
                SELECT graphic_blob FROM internal_graphics WHERE                        1       1
0   6079        468
                graphic_id=32
                SELECT graphic_blob FROM internal_graphics WHERE                        1       1
0   6074        468
                graphic_id=4
                SELECT graphic_blob FROM internal_graphics WHERE                        1       1
0   5962        468
                graphic_id=8
***************
------ ---------
sum
67226      4640
```

Figure 14: Report Output Example For a Users SQL Area

One warning about the script in figure 13, the report it generates can run to several hundred pages for a user with a large number of SQL areas. What things should you watch for in a user's SQL areas? First, watch for the non-use of bind variables, bind variable usage is shown by the inclusion of variables such as ":1" or ":B" in the SQL text. Notice that in the example report in Figure 8 the first four statements use bind variables, and, consequently are reusable. Non-bind usage means hard coded values such as 'Missing' or '10' are used. Notice that for most of the rest of the statements in the report no bind variables are used even though many of the SQL statements are nearly identical. This is one of the leading causes of shared pool misuse and results in useful SQL being drown in tons of non-reusable garbage SQL.

The problem with non-reusable SQL is that it must still be looked at by any new SQL inserted into the pool (actually it's hash value is scanned). While a hash value scan may seem a small cost item, if your shared pool contains tens of thousands of SQL areas this can be a performance bottleneck. How can we determine, without running the report in Figure 13 for each of possibly hundreds of users, if we have garbage SQL in the shared pool?

The script in Figure 15 shows a view that provides details on individual users SQL area reuse. The view can be tailored to your environment if the limit on reuse (currently set at 1) is too restrictive. For example, in a recent tuning assignment resetting the value to 12 resulting in nearly 70 percent of the SQL being rejected as garbage SQL, in DSS or data warehouse systems where rollups are performed by the month, bi-monthly or weekly values of 12, 24 or 52 might be advisable. Figure 16 shows a report script that uses the view created in Figure 15.

```
REM
REM View to sort SQL into GOOD and GARBAGE
REM
CREATE OR REPLACE VIEW sql_garbage AS
SELECT
  b.username users,
  SUM(a.sharable_mem+a.persistent_mem) Garbage,
  TO_NUMBER(null) good
FROM
   sys.v_$sqlarea a, dba_users b
WHERE
  (a.parsing_user_id = b.user_id and a.executions<=1)
GROUP BY b.username
UNION
SELECT DISTINCT
  b.username users,
  TO_NUMBER(null) garbage,
  SUM(c.sharable_mem+c.persistent_mem) Good
FROM
   dba_users b, sys.v_$sqlarea c
WHERE
  (b.user_id=c.parsing_user_id and c.executions>1)
GROUP BY b.username;
```

Figure 15: Example Script to Create the SQL_GARBAGE View

```
REM
REM Report on SQL Area Reuse by user
REM
column garbage                format 9,999,999,999 heading 'Non-Shared
SQL'
column good           format 9,999,999,999 heading 'Shared SQL'
column good_percent   format 999.99     heading 'Percent Shared'
set feedback off
break on report
compute sum of garbage on report
compute sum of good on report
compute avg of good_percent on report
@title80 'Shared Pool Utilization'
spool rep_out\&db\sql_garbage
select
       a.users,
       a.garbage,
       b.good,
       (b.good/(b.good+a.garbage))*100 good_percent
from
       sql_garbage a, sql_garbage b
where
       a.users=b.users
and
       a.garbage is not null
and
       b.good is not null
/
spool off
set feedback off
clear columns
```

```
clear breaks
clear computes
```

Figure 16: Example Report Script For SQL Reuse Statistics

The report script in Figure 16 shows at a glance (well, maybe a long glance for a system with hundreds of users) which users aren't making good use of reusable SQL. An example report output is shown in Figure 17.

```
Date: 11/18/98                                                    Page:    1
Time: 04:16 PM            Shared Pool Utilization           SYSTEM
                              ORTEST1 databas

USERS                          Non-Shared SQL     Shared SQL Percent
Shared
------------------------------ -------------- -------------- -------------
-
GRAPHICS_DBA                          27,117         38,207
58.49
SYS                                  302,997        575,176
65.50
SYSTEM                             1,504,740        635,861
29.70
                                   -------------- -------------- -------------
-
avg
51.23
sum                                1,834,854      1,249,244
```

Figure 17: Example Report From Showing SQL Reuse Statistics

Notice in Figure 17 that the GRAPHICS_DBA user only shows 58.49% shared SQL use based on memory footprints. From the report in Figure 14 we would expect a low reuse value for GRAPHICS_DBA. The low reuse value for the SYSTEM user is due to its use as a monitoring user, the monitoring SQL is designed to be used once per day or so and was not built with reuse in mind.

Putting it All In Perspective

So what have we seen so far? We have examined reports that show both gross and detailed shared pool usage and whether or not shared areas are being reused. What can we do with this data? Ideally we will use the results to size our shared pool properly. Let's set out a few general guidelines for shared pool sizing:

Guideline 1: If gross usage of the shared pool in a non-ad-hoc environment exceeds 95% (rises to 95% or greater and stays there) establish a shared pool size large enough to hold the fixed size portions, pin reusable packages and procedures. Increase shared pool by 20% increments until usage drops below 90% on the average.

Guideline 2: If the shared pool shows a mixed ad-hoc and reuse environment establish a shared pool size large enough to hold the fixed size portions, pin reusable packages and establish a comfort level above this required level of pool fill. Establish a routine flush cycle to filter non-reusable code from the pool.

Guideline 3: If the shared pool shows that no reusable SQL is being used establish a shared pool large enough to hold the fixed size portions plus a few megabytes (usually not more than 40) and allow the shared pool modified least recently used (LRU) algorithm to manage the pool.

In guidelines 1, 2 and 3, start at around 40 megabytes for a standard size system. Notice in guideline 2 it is stated that a routine flush cycle should be instituted. This flies in the face of what Oracle Support pushes in their shared pool white papers, however, they work from the assumption that proper SQL is being generated and you want to reuse the SQL present in the shared pool. In a mixed environment where there is a mixture of reusable and non-reusable SQL the non-reusable SQL will act as a drag against the other SQL (I call this shared pool thrashing) unless it is periodically removed by flushing. Figure 18 shows a PL/SQL package which can be used by the DBMS_JOB job queues to periodically flush the shared pool only when it exceeds a specified percent full.

```
CREATE OR REPLACE PROCEDURE flush_it(
    p_free IN NUMBER, num_runs IN NUMBER) IS
--
CURSOR get_share IS
SELECT
  LEAST(MAX(b.value)/(1024*1024),SUM(a.bytes)/(1024*1024))
   FROM v$sgastat a, v$parameter b
 WHERE (a.pool='shared pool'
 AND a.name <> ('free memory'))
 AND b.name = 'shared_pool_size';
--
CURSOR get_var IS
 SELECT  value/(1024*1024)
 FROM v$parameter
 WHERE name = 'shared_pool_size';
--
-- Following cursors from Steve Adams Nice_flush
--
  CURSOR reused_cursors IS
    SELECT address || ',' || hash_value
    FROM sys.v_$sqlarea
    WHERE executions > num_runs;
  cursor_string varchar2(30);
--
  CURSOR cached_sequences IS
    SELECT  sequence_owner, sequence_name
    FROM  sys.dba_sequences
    WHERE cache_size > 0;
  sequence_owner varchar2(30);
  sequence_name varchar2(30);
--
```

```
  CURSOR candidate_objects IS
    SELECT kglnaobj, decode(kglobtyp, 6, 'Q', 'P')
    FROM sys.x_$kglob
    WHERE inst_id = userenv('Instance') AND
      kglnaown = 'SYS' AND kglobtyp in (6, 7, 8, 9);
  object_name varchar2(128);
  object_type char(1);
--
-- end of Steve Adams Cursors
--
  todays_date  DATE;
  mem_ratio    NUMBER;
  share_mem    NUMBER;
  variable_mem         NUMBER;
  cur          INTEGER;
  sql_com      VARCHAR2(60);
  row_proc     NUMBER;
--
BEGIN
 OPEN get_share;
 OPEN get_var;
 FETCH get_share INTO share_mem;
 FETCH get_var INTO variable_mem;
 mem_ratio:=share_mem/variable_mem;
 IF mem_ratio>p_free/100 THEN
 --
 -- Following keep sections from Steve Adams nice_flush
 --
 BEGIN
  OPEN reused_cursors;
  LOOP
    FETCH reused_cursors INTO cursor_string;
    EXIT WHEN reused_cursors%notfound;
    sys.dbms_shared_pool.keep(cursor_string, 'C');
  END LOOP;
 END;
 BEGIN
  OPEN cached_sequences;
  LOOP
    FETCH cached_sequences INTO sequence_owner, sequence_name;
    EXIT WHEN cached_sequences%notfound;
    sys.dbms_shared_pool.keep(sequence_owner || '.' || sequence_name,
'Q');
  END LOOP;
 END;
 BEGIN
  OPEN candidate_objects;
  LOOP
    FETCH candidate_objects INTO object_name, object_type;
    EXIT WHEN candidate_objects%notfound;
    sys.dbms_shared_pool.keep('SYS.' || object_name, object_type);
  END LOOP;
 END;
 --
 -- end of Steve Adams section
 --
  cur:=DBMS_SQL.OPEN_CURSOR;
  sql_com:='ALTER SYSTEM FLUSH SHARED_POOL';
```

```
  DBMS_SQL.PARSE(cur,sql_com,dbms_sql.v7);
  row_proc:=DBMS_SQL.EXECUTE(cur);
  DBMS_SQL.CLOSE_CURSOR(cur);
 END IF;
END flush_it;
```

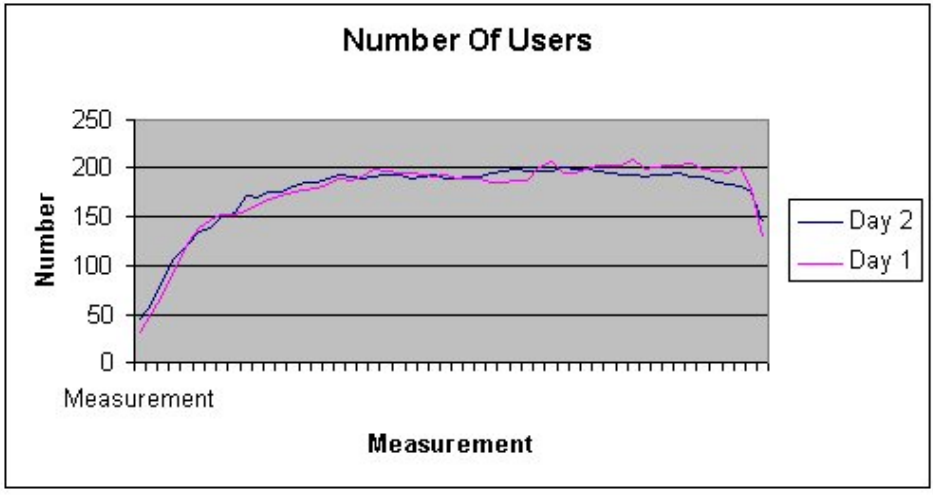
Figure 18: Example Script to Run a Shared Pool Flush Routine

The command set to perform a flush on a once every 30 minute cycle when the pool reaches 95% full would be:

```
VARIABLE x NUMBER;
BEGIN
dbms_job.submit(
:X,'BEGIN flush_it(95); END;',SYSDATE,'SYSDATE+(30/1440)');
END;
/
COMMIT;
```

(Always commit after assigning a job or the job will not be run and queued)

There is always a discussion as to whether this really does help performance so I set up a test on a production instance where on day 1 I did no automated flushing and on day 2 I instituted the automated flushing. Figure 19 shows the graphs of performance indicators, flush cycles and users.



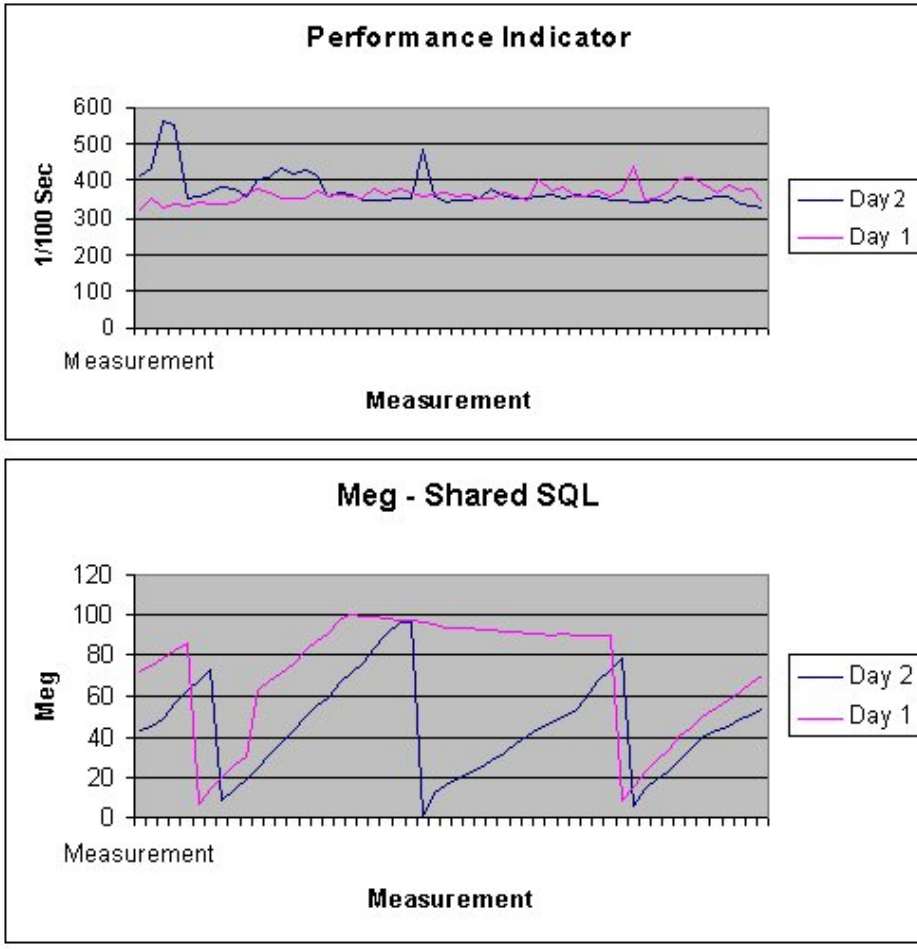


Figure 19: Graphs Showing Effects of Flushing

The thing to notice about the graphs in Figure 19 is the overall trend of the performance indicator between day 1 and day 2. On day 1 (the day with an initial flush as indicated by the steep plunge on the pool utilization graph followed by the buildup to maximum and the flattening of the graph) the performance indicator shows an upward trend. The performance indicator is a measure of how long the database takes to do a specific set of tasks (from the Q Diagnostic tool from Savant Corporation). Therefore an increase in the performance indicator indicates a net decrease in performance. On day 2 the overall trend is downward with the average value less than the average value from day 1. Overall the flushing improved the performance as indicated by the performance indicator by

10 to 20 percent. Depending on the environment I have seen improvements of up to 40-50 percent.

One thing that made the analysis difficult was that on day 2 there were several large batch jobs run which weren't run on day 1. The results still show that flushing has a positive effect on performance when the database is a mixed SQL environment with a large percentage of non-reusable SQL areas.

Guideline 3 also brings up an interesting point, you may already have over allocated the shared pool, in this case guideline 3 may result in you decreasing the size of the shared pool. In this situation the shared pool has become a cesspool filled with nothing but garbage SQL. After allocating enough memory for dictionary objects and other fixed areas and ensuring that the standard packages and such are pinned, you should only maintain a few megabytes above and beyond this level of memory for SQL statements. Since none of the code is being reused you want to reduce the hash search overhead as much as possible, you do this by reducing the size of the available SQL area memory so as few a number of statements are kept as possible.

What to Pin

In all of the guidelines stated so far I mention that the memory is usually allocated above and beyond that needed for fixed size areas and pinned objects. How do you determine what to pin? Generally speaking any package, procedure, function or cursor that is frequently used by your application should be pinned into the shared pool when the database is started. I suggest adding a "null" startup function to every in house generated package it essentially looks like Figure 20.

```
FUNCTION start_up
RETURN number IS
Ret NUMBER:=1;
BEGIN
Ret:=0
RETURN ret;
END start_up;
```

Figure 20: Example Null Startup Function

The purpose of the null startup function is to provide a touch point to pull the entire package into the shared pool. This allows you to create a startup SQL procedure that pulls all of the application packages into the pool and pins them using the DBMS_SHARED_POOL package. The DBMS_SHARED_POOL package may have to be built in earlier releases of Oracle. The DBMS_SHARED_POOL package is built using the DBMSPOOL.SQL and PRVTPOOL.PLB scripts located in (UNIX) $ORACLE_HOME/rdbms/admin or (NT) x:\orant\rdbms\admin (where x: is the home drive for your install).

How do you determine what packages, procedures of functions to pin? Actually, Oracle has made this easy by providing the V$DB_OBJECT_CACHE view that shows all objects in the pool, and, more importantly, how they are being utilized. The script in Figure 21 provides a list of objects that have been loaded more than once and have executions greater than one. Some example output from this script is shown in figure 22. A rule of thumb is that if an object is being frequently executed and frequently reloaded it should be pinned into the shared pool.

```
rem
rem FUNCTION: Report Stored Object Statistics
rem
column owner           format a11            heading Schema
column name            format a30            heading Object|Name
column namespace                                    heading Name|Space
column type                                  heading Object|Type
column kept            format a4             heading Kept
column sharable_mem    format 999,999        heading Shared|Memory
column executions      format 999,999        heading Executes
set lines 132 pages 47 feedback off
@title132 'Oracle Objects Report'
break on owner on namespace on type
spool rep_out/&db/o_stat
select
       OWNER,
       NAMESPACE,
       TYPE,
       NAME,
       SHARABLE_MEM,
       LOADS,
       EXECUTIONS,
       LOCKS,
       PINS,
       KEPT
from
       v$db_object_cache
where
       type not in (
'NOT LOADED','NON-EXISTENT','VIEW','TABLE','SEQUENCE')
       and executions>0 and loads>1 and kept='NO'
order by owner,namespace,type,executions desc;
spool off
set lines 80 pages 22 feedback on
clear columns
clear breaks
ttitle off
```

Figure 21: Script to Show Objects Which Should Be Kept

The output from the script in Figure 21 is shown in Figure 22. Notice the objects with high executions.

```
Date:                                                                                    11/20/98
Page:   1
Time:  09:59  AM                                                          Oracle  Objects  Report
AULTM
                                                  AGCD database

       Name            Object         Object                          Shared
Schema Space            Type            Name                             Memory     LOADS Executes
LOCKS       PINS Kept
------ --------------- -------------- --------------------------- -------- --------- -------- ----
------ ---------- ----
SYS     BODY              PACKAGE BODY   DBMS_EXPORT_EXTENSION            6,957          1     1,338
1          0 NO
                                        DBMS_SQL                        11,016          1        50
1          0 NO
                                        DBMS_SYS_SQL                    21,428          1        50
1          0 NO
                                        DBMS_DEFER_IMPORT_INTERNAL       4,070          1        50
1          0 NO
                                        STANDARD                        26,796          1        50
1          0 NO
                                        DBMS_APPLICATION_INFO            4,585          1         8
1          0 NO
                                        DBMS_OUTPUT                      8,799          1         1
1          0 NO
       TABLE/PROCEDURE  PACKAGE          DBMS_EXPORT_EXTENSION           12,269          1     1,355
1          0 NO
                                        DBMS_DEFER_IMPORT_INTERNAL      10,662          1        51
1          0 NO
                                        DBMS_SQL                         6,960          1        50
1          0 NO
                                        STANDARD                       118,556          1        50
1          0 NO
                                        DBMS_SYS_SQL                     7,472          1        50
1          0 NO
                                        DBMS_APPLICATION_INFO           11,569          1         9
1          0 NO
                                        DBMS_OUTPUT                     13,391          1         1
1          0 NO
```

Figure 22: Example Output From the Script In Figure 21.

Unfortunately in my active instance I already have the objects pinned that are required, but the example report in Figure 22 taken from one of my less active instances still shows the concept. Note that you only have to pin the package, not the package and package body.

Guideline 4: Determine usage patterns of packages, procedures, functions and cursors and pin those that are frequently used.

The Shared Pool and MTS

The use of the multi-threaded server option (MTS) in Oracle requires a sometimes dramatic increase in the size of the shared pool. This increase in the size of the shared pool caused by MTS is due to the addition of the user global areas required for sorting and message queues. If you are using MTS you should monitor the V$SGASTAT values for MTS related memory areas and adjust the shared pool memory allocations accordingly.

Note that in Oracle 8 you should make use of the large pool feature to pull the user global areas (UGA) and multi-threaded server queues out of the shared pool

area if MTS is being used. This prevents the fragmentation problems that have been reported in shared pools when MTS is used without allocating the large pool. The parallel query option (PQO) in Oracle8 also makes use of the large pool area, depending on the number of users and degree of parallel, the large pool may require over 200 megabytes by itself in a PQO environment.

Large Pool Sizing

Sizing the large pool can be complex. The large pool, if configured must be at least 600 kilobytes in size. Usually for most MTS applications 600k is enough. However, if PQO is also used in your Oracle8 environment then the size of the large pool will increase dramatically. The V$SGASTAT dynamic performance view has a new column in Oracle8, POOL. The POOL column in the V$SGASTAT view is used to contain the pool area where that particular type of object is being stored. By issuing a summation select against the V$SGASTAT view a DBA can quickly determine the size of the large pool area currently being used.

```
SELECT name, SUM(bytes) FROM V$SGASTAT WHERE pool='LARGE POOL' GROUP BY
ROLLUP(name);
```

The above select should be used when an "ORA-04031:Unable to allocate 16084 bytes of shared memory ("large pool", "unknown object", "large pool hea", "PX large pool") " error is received during operation with a large pool configured (the number of bytes specified may differ). When the above select is run, the resulting summary number of bytes will indicate the current size of the pool and show how close you are to your maximum as specified in the initialization parameter LARGE_POOL_SIZE. Generally increasing the large_pool by up to 100% will eliminate the ORA-04031 errors.

Oracle8i provides for automated sizing of the large pool. If PARALLEL_AUTOMATIC_TUNING is set to TRUE or if PARALLEL_MAX_SERVERS is set to a non-zero value then the LARGE_POOL_SIZE will be calculated, however, it can be over-ridden with a manually specified entry in the initialization file. Indeed, if an ORA-27102: Out of Memory error is received when you set either of these parameters (or both) you must either manually set LARGE_POOL_SIZE or reduce the value for PARALLEL_MAX_SERVERS. The following formula determines the set point for the LARGE_POOL_SIZE if it is not manually set:

```
(DOP^2*(4I-1)+2*DOP*3+4*DOP(I-1))*PEMS*USERS
```

Where

- DOP – Degree of Parallel calculated from #CPU/NODE * #NODES
- I – Number of threads/CPU

- PEMS – Parallel execution message size – set with PARALLEL_EXECUTION_MESSAGE_SIZE initialization parameter, usually defaults to 2k or 4k but can be larger.
- USERS – Number of concurrent users using parallel query

For a 2k PEMS with 4 concurrent users for a steadily increasing value for DOP the memory size is a quadratic function ranging from around 4 meg for 10 CPUs to 120 meg with 70 CPUs. This memory requirement is demonstrated in Figure 23.

Figure 23: Example Chart for 2k PEMS and 4 Concurrent Users Showing Memory Requirements as Number of CPUs Increases

On my NT4.0 Oracle8i, 8.1.3 test system I have 2 CPUs, set at 2 threads per cpu (DOP of 4) and then 4 threads per cpu (DOP of 8), message buffer of 4k and I performed multiple tests increasing the PARALLEL_MAX_SERVERS initialization parameter to see what the resulting increase in LARGE_POOL_SIZE would be, the results were:

```
PARALLEL_MAX_SERVERS   DOP 4 LARGE_POOL_SIZE          DOP 8 LARGE_POOL_SIZE
4                                685,024 bytes
685,024 bytes
   857,056 bytes                                                857,056 bytes
16                            1,151,968 bytes
1,545,184 bytes
```

Notice that for a small number of CPUs the large pool size increase from an increase in parallel max servers isn't affected by changes in the number of parallel threads until the value of threads is large in respect to the number of CPUs.

For non-PQO systems a general rule of thumb is 5K of memory for each MTS user for the large pool area.

Guideline 5: In Oracle7when using MTS increase the shared pool size to accommodate MTS messaging and queuing as well as UGA requirements. In Oracle8 use the Large Pool to prevent MTS from effecting the shared pool areas.

A Matter Of Hashing

We have discussed hashing in prior sections, essentially each SQL statement is hashed and this hash value is then used to compare to already stored SQL areas, if a matching hash is found the statements are compared. The hash is only calculated based on the first 200 or so characters in the SQL statement, so extremely long SQL statements can result in multiple hashes being the same even though the stored SQL is different (if the first 100 or so characters in each

statement are identical). This is another argument for using stored procedures and functions to perform operations and for the use of bind variables. In 8.0 the hash value is calculated on the first 100 and last 100 characters reducing the chances of multiple identical hash values for different SQL statements. In 8i the hash is calculated on the entire SQL text so multiple identical hashes should never occur.

If the number of large, nearly identical statements is high, then the number of times the parser has to compare a new SQL statement to existing SQL statements with the same hash value increases. This results in a higher statement overhead and poorer performance. You should identify these large statements and encourage users to re-write them using bind variables or to proceduralize them using PL/SQL. The report in Figure 24 will show if you have a problem with multiple statements being hashed to the same value.

```
Rem:
rem: FUNCTION: Shows by user who has possible
rem:           SQL reuse problems
rem:
column total_hash                              heading 'Total Hash|Values'
column same_hash                                       heading 'SQL With|Same
Hash'
column u_hash_ratio    format 999.999          heading 'SQL Sharing|Hash'
start title80 'Shared Hash Value Report'
spool rep_out\&&db\shared_hash.lst
break on report
compute sum of total_hash on report
compute sum of same_hash on report
select
       a.username,
       count(b.hash_value) total_hash,
       count(b.hash_value)-count(unique(b.hash_value)) same_hash,
(count(unique(b.hash_value))/count(b.hash_value))*100 u_hash_ratio
from
       dba_users a,
       v$sqlarea b
where
       a.user_id=b.parsing_user_id
group by
       a.username;
clear computes
```

Figure 24: Example Script to Report on Hashing Problems

The script in Figure 24 produces a report similar to that shown in Figure 25. The report in Figure 25 shows which users are generating SQL that hashes to the same values. Once you have a user isolated you can then run the script in Figure 26 to find the bad SQL statements.

```
Date: 11/20/98                                                  Page:    1
Time: 11:40 AM            Shared Hash Value Report              AULTM
                               DCARS database

                                Total Hash  SQL With SQL Sharing
USERNAME                            Values Same Hash        Hash
------------------------------ ---------- --------- -----------
AULTM                                  129         0     100.000
DCARS                                 6484        58      99.105
MCNAIRT                                 20         0     100.000
PASSMAP                                  2         0     100.000
QDBA                                   109         0     100.000
RCAPS                                  270         0     100.000
RCOM                                   342         7      97.953
REPORTS1                                28         0     100.000
SECURITY_ADMIN                          46         0     100.000
SYS                                    134         0     100.000
                               ---------- ---------
sum                                   7564        65
```

Figure 25: Hash Report

A quick glance at the report in Figure 25 shows that we need to look at the DCARS user to correct hashing problems they might be having and improve the reuse of SQL in the shared pool. However, look at the number of hash areas this user has accumulated, 6,484, if I run the report from Figure 13 it will out weigh the paper version of the Oracle documentation set. A faster way to find the hash values would be to do a self join and filter out the hash values that are duplicate. Sounds easy enough, but remember, the V$ tables have no rowids so you can't use the classic methods, you have to find another column that will be different when the HASH_VALUE column in V$SQLAREA is the same. Look at the select in Figure 26.

```
select distinct a.hash_value from v$sqlarea a, v$sqlarea b, dba_users c
where a.hash_value=b.hash_value and
a.parsing_user_id = c.user_id
and c.username='DCARS' and  ← change to user you are concerned about
a.FIRST_LOAD_TIME != b.FIRST_LOAD_TIME
```

Figure 26: Example Select To Determine Duplicate Hash Values

Figure 27 has an example output from the above select.

```
DCARS:column hash_value format 99999999999
DCARS:set echo on
DCARS: select distinct a.hash_value from v$sqlarea a, v$sqlarea b,
  2  dba_users c
  3  where a.hash_value=b.hash_value and
  4  a.parsing_user_id = c.user_id
  5  and c.username='DCARS' and
  6* a.FIRST_LOAD_TIME != b.FIRST_LOAD_TIME
```

```
  HASH_VALUE
------------
 -1595172473
 -1478772040
 -1344554312
  -941902153
  -807684425
  -507978165
  -270812489
   441376718
   784076104
   979296206
  1765990350
  1945885214
```

Figure 26: Example Hash Select Output

Once you have the hash value you can pull the problem SQL statements from either V$SQLAREA or V$SQLTEXT very easily, look at Figure 27.

```
DCARS:select sql_text from v$sqlarea where hash_value='441376718';

SQL_TEXT
------------------------------------------------------------------------
SELECT   region_code,   region_dealer_num,   consolidated_dealer_num,
dealer_name,   dealer_status_code,   dealer_type_code,   mach_credit_code,
parts_credit_code  FROM dealer  WHERE   region_code = '32' AND
region_dealer_num = '6433'

SELECT   region_code,   region_dealer_num,   consolidated_dealer_num,
dealer_name,   dealer_status_code,   dealer_type_code,   mach_credit_code,
parts_credit_code  FROM dealer  WHERE   region_code = '56' AND
region_dealer_num = '6273'
```

Figure 27: Example of Statements With Identical Hash Values But Different SQL

Long statements require special care to see that bind variables are used to prevent this problem with hashing. Another help for long statements is to use views to store values at an intermediate state thus reducing the size of the variable portion of the SQL. Notice in the example select in Figure 27 that the only difference between the two identically hashed statements is that the “region_code” and “region_dealer_num” comparison values are different, if bind variables had been used in these statements there would only have been one entry instead of two.

Guideline 6: Use bind variables, PL/SQL (procedures or functions) and views to reduce the size of large SQL statements to prevent hashing problems.

Monitoring Library and Data Dictionary Caches

I've spent most of this article looking at the shared SQL area of the shared pool. Let's wrap up with a high level look at the library and data dictionary caches. The library cache area is monitored via the V$LIBRARYCACHE view and contains the SQL area, PL/SQL area, table, index and cluster cache areas. The data dictionary caches contain cache area for all data dictionary related definitions.

The script in Figure 28 creates a report on the library caches. The items of particular interest in the report generated by the script in Figure 28 (shown in Figure 29) are the various ratios.

```
rem
rem Title: libcache.sql
rem
rem FUNCTION: Generate a library cache report
rem
column namespace                        heading "Library Object"
column gets             format 9,999,999 heading "Gets"
column gethitratio      format 999.99   heading "Get Hit%"
column pins             format 9,999,999 heading "Pins"
column pinhitratio      format 999.99   heading "Pin Hit%"
column reloads          format 99,999   heading "Reloads"
column invalidations    format 99,999   heading  "Invalid"
column db format a10
set pages 58 lines 80
start title80 "Library Caches Report"
define output = rep_out\&db\lib_cache
spool &output
select
        namespace,
        gets,
        gethitratio*100 gethitratio,
        pins,
        pinhitratio*100 pinhitratio,
        RELOADS,
        INVALIDATIONS
from
        v$librarycache
/
spool off
pause Press enter to continue
set pages 22 lines 80
ttitle off
undef output
```

Figure 28: Example Script To Monitor The Library Caches

Look at the example output from the script in Figure 28 in Figure 29. In Figure 29 we see that all Get Hit% (gethitratio in the view) except for indexes are greater than 80-90 percent. This is the desired state, the value for indexes is low

because of the few accesses of that type of object. Notice that the Pin Hit% is also greater than 90% (except for indexes) this is also to be desired. The other goals of tuning this area are to reduce reloads to as small a value as possible (this is done by proper sizing and pinning) and to reduce invalidations. Invalidations happen when for one reason or another an object becomes unusable. However, if you must use flushing of the shared pool reloads and invalidations may occur as objects are swapped in and out of the shared pool. Proper pinning can reduce the number of objects reloaded and invalidated.

Guideline 7: In a system where there is no flushing increase the shared pool size in 20% increments to reduce reloads and invalidations and increase hit ratios.

```
Date: 11/21/98                                              Page:   1
Time: 02:51 PM            Library Caches Report             SYSTEM
                             ORTEST1 database

Library Object        Gets Get Hit%       Pins Pin Hit% Reloads Invalid
--------------- ---------- -------- ---------- -------- ------- -------
SQL AREA            46,044    99.17     99,139    99.36      24      16
TABLE/PROCEDURE      1,824    84.59      6,935    93.21       3       0
BODY                   166    93.98        171    91.23       0       0
TRIGGER                  0   100.00          0   100.00       0       0
INDEX                   27      .00         27      .00       0       0
CLUSTER                373    98.12        373    97.59       0       0
OBJECT                   0   100.00          0   100.00       0       0
PIPE                     0   100.00          0   100.00       0       0
```

Figure 29: Example Of The Output From Library Caches Report

The data dictionary caches used to be individually tunable through several initialization parameters, now they are internally controlled. The script in Figure 30 should be used to monitor the overall hit ratio for the data dictionary caches.

```
rem
rem title:    ddcache.sql
rem FUNCTION: report on the v$rowcache table
rem HISTORY:  created sept 1995 MRA
rem
start title80 "DD Cache Hit Ratio"
spool rep_out\&db\ddcache
SELECT (SUM(getmisses)/SUM(gets)) RATIO
FROM V$ROWCACHE
/
spool off
pause Press enter to continue
ttitle off
```

Figure 30: Script to Monitor the Data Dictionary Caches

The output from the script in Figure 30 is shown in Figure 31.

```
Date: 11/21/98                                              Page:    1
Time: 02:59 PM              DD Cache Hit Ratio              SYSTEM
                             ORTEST1 database

    RATIO
---------
.01273172
```

Figure 31: Example Output From Data Dictionary Script

The ratio reported from the script in Figure 30 should always be less than 1. The ratio corresponds to the number of times out of 100 that the database engine sought something from the cache and missed. A dictionary cache miss is more expensive than a data block buffer miss so if your ratio gets near 1 increase the size of the shared pool since the internal algorithm isn't allocating enough memory to the data dictionary caches.

Guideline 8: In any shared pool, if the overall data dictionary cache miss ratio exceeds 1 percent, increase the size of the shared pool.

In Summary

In section of the tuning paper we have discussed ways to monitor for what objects should be pinned, discussed multi-threaded server , looked at hashing problems and their resolution as well as examined classic library and data dictionary cache tuning. Including the guidelines from last months article we have established 8 guidelines for tuning the Oracle shared pool:

Guideline 1: If gross usage of the shared pool in a non-ad-hoc environment exceeds 95% (rises to 95% or greater and stays there) establish a shared pool size large enough to hold the fixed size portions, pin reusable packages and procedures. Gradually increase shared pool by 20% increments until usage drops below 90% on the average.

Guideline 2: If the shared pool shows a mixed ad-hoc and reuse environment, establish a shared pool size large enough to hold the fixed size portions, pin reusable packages and establish a comfort level above this required level of pool fill. Establish a routine flush cycle to filter non-reusable code from the pool.

Guideline 3: If the shared pool shows that no reusable SQL is being used establish a shared pool large enough to hold the fixed size portions plus a few

megabytes (usually not more than 40) and allow the shared pool modified least recently used (LRU) algorithm to manage the pool. (also see guideline 8)

Guideline 4: Determine usage patterns of packages, procedures, functions and cursors and pin those that are frequently used.

Guideline 5: In Oracle7when using MTS increase the shared pool size to accommodate MTS messaging and queuing as well as UGA requirements. In Oracle8 use the Large Pool to prevent MTS from effecting the shared pool areas.

Guideline 6: Use bind variables, PL/SQL (procedures or functions) and views to reduce the size of large SQL statements to prevent hashing problems.

Guideline 7: In a system where there is no flushing increase the shared pool size in 20% increments to reduce reloads and invalidations and increase object cache hit ratios.

Guideline 8: In any shared pool, if the overall data dictionary cache miss ratio exceeds 1 percent, increase the size of the shared pool.

Using these guidelines and the scripts and techniques you should be well on the way towards a well tuned and well performing shared pool.

Tuning Checkpoints

Checkpoints provide for concurrency in an Oracle database. Checkpoints write out timestamp and SCN information as well as dirty blocks to the database files.Pre-7.3.4 the checkpoint process was optional, now it is required.

Checkpoints provide for rolling forward after a system crash. Data is applied from the time of the last checkpoint forward from the redo entries. Checkpoints also provide for reuse of redo logs. When a redo log is filled the LGWR process automatically switches to the next available log. All data in the now inactive log is written to disk by an automatic checkpoint. This frees the log for reuse or for archiving.

Checkpoints occur when a redo log is filled, when the INIT.ORA parameter LOG_CHECKPOINT_INTERVAL ORACLE7 is reached (Total bytes written to a redo log), or the elapsed time has reached the INIT.ORA parameter LOG_CHECKPOINT_TIMEOUT expressed in seconds or every three seconds, or when an ALTER SYSTEM command is issued with the CHECKPOINT option specified.

While frequent checkpoints will reduce recovery time, they will also decrease performance. Infrequent checkpoints will increase performance but increase

required recovery times. To reduce checkpoints to only happen on log switches, set LOG_CHECKPOINT_INTERVAL to larger than your redo log size, and set LOG_CHECKPOINT_TIMEOUT to zero.

If checkpoints still cause performance problems, set the INIT.ORA parameter CHECKPOINT_PROCESS to TRUE to start the CKPT process running. This will free the DBWR from checkpoint duty and increase performance. The INIT.ORA parameter PROCESSES may also have to be increased. Note that on Oracle8 and greater the checkpoint process is not optional and is started along with the other Oracle instance processes.

Another new option with Oracle8i is the concept of fast-start checkpointing. In order to configure fast-start checkpointing you set the initialization parameter FAST_START_IO_TARGET. The FAST_START_IO_TARGET parameter sets the number of IO operations that Oracle will attempt to limit itself to before writing a checkpoint. This feature is only available with Oracle 8i Enterprise Edition.

Other initialization parameters that control checkpointing are:

- LOG_BUFFER_SIZE – should be set such that there aren't large numbers of small writes and the overall write tiome isn't too long, usually not more than 1 megabyte.
- LOG_SMALL_ENTRY_MAX_SIZE (Gone in 8i) sets the size in bytes for the largest copy to the redo buffers that occurs under the redo allocation latch. Decreasing the size of this parameter will reduce contention for the redo allocation latch.
- LOG_SIMULTANEOUS_COPIES (Gone to "_" in 8i) set to twice the number of CPUs to reduce contention for the redo copy latches by increasing the number of latches.
- LOG_ENTRY_PREBUILD_THRESHOLD(Gone to "_" in 8.0, gone in 8i) sets the number of bytes of redo to gather before copying to the log buffer. For multi-CPU systems increasing this value can be beneficial.

Tuning Redo Logs

To tune redo logs you should:

- Actually tune LGWR process to optimize log writes
- LGWR writes when log buffers 1/3 full, or on COMMIT
- Tune redo log size based on transaction size, too small a size results in frequent inefficient IO, too large results in too long a write

- Be sure logs are not in contention with each other or other files

To Determine average transaction size as far as redo buffer writes:

```
(redo size + redo wastage)
---------------------------------
Redo writes
```

Use data from V$SYSSTAT. Size your log buffers to near this size, error on too much rather than too little.

Size actual redo logs such that they switch every thirty minutes, or based on the amount of data you can afford to lose (loss of the active redo log results in loss of its data.)
Redo logs maintain a complete history of data and database changing transactions. Redo logs are critical for recovery and operation of the Oracle database system. Unfortunately redo logs are another structure that is difficult to tune before an application system goes active. The majority of tuning efforts with redo logs deal with two important issues:

1. Minimize the impact of the redo log/archive log/checkpoint processes on database performance.
2. Maximize recoverability of the database

At times these two goals may be in opposition since by maximizing recoverability (by reducing time to recovery for example) you will cause a performance impact. I am afraid you will have to balance these two goals while dealing with redo log tuning, however one thing to remember is that you will (hopefully) spend much more time dealing with an operational database than you will recovering a database so in the greater scheme of things perhaps optimizing for performance is the major goal you should attempt to reach.

Redo Log Sizing

The size of a redo log depends on the transaction volume within your database. Unfortunately there are no magic formulae to apply that will give you a size value, it is completely empirically derived. Oracle requires at least two groups with one redo log member per group for Oracle to start. If you have archive logging enabled this should b pushed to a minimum of three groups of one redo log member each. I prefer a minimum of five groups of two mirrored members each for archive logging.

Redo logs should be sized so that should you loose the online redo log a minimal amount of data is lost. What is a minimal amount of data? Your guess is as good as mine is, however you need to ask your users (or managers) how much data

can they afford to lose? The value they give you for data will probably relate to a time interval such as “we can lose an hours worth of data but no more”. If you are given a time interval then you need to size the logs such that a log switch happens approximately at that interval during normal usage.

Log switch information is contained in the various versions of the v$log_hist or v$log_history views. Log switch information is also contained in the alert log. Once you establish how much data you can afford to lose (based on a time interval) monitor your views or alert log to find how often log switches are happening and adjust the size up or down to meet your requirement. Figure 32 shows a script to generate log switch statistics.

```
REM NAME   :log_hist.sql
REM PURPOSE:Provide info on logs for last 24 hour since last log switch
REM USE    : From SQLPLUS
REM Limitations        : None
REM
COLUMN thread#         FORMAT 999     HEADING 'Thrd#'
COLUMN sequence#       FORMAT 99999   HEADING 'Seq#'
COLUMN first_change#                  HEADING 'Low#'
COLUMN next_change#                   HEADING 'High#'
COLUMN first_time                     HEADING 'Accessed'
SET LINES 80
@title80 "Log History Report"
SPOOL rep_out\&db\log_hist
REM
SELECT thread#, sequence#,
   first_change#,next_change#,
   TO_CHAR(a.first_time,'dd-mon-yyyy hh24:mi:ss') first_time
FROM
       v$log_history a
WHERE
       a.first_time >
       (SELECT b.first_time-1
       FROM v$log_history b WHERE b.next_change# =
       (SELECT MAX(c.next_change#) FROM v$log_history c));
SPOOL OFF
SET LINES 80
CLEAR COLUMNS
TTITLE OFF
PAUSE Press enter to continue
```

Figure 32: Script to Generate redo Log Switch Information

The above script will provide log switch information, the output from the script is shown in Figure 33.

```
Date: 04/02/99                                          Page:   1
Time: 09:58 AM            Log History Report              SYSTEM
                               DMDB database

Thrd#   Seq#       Low#      High# Accessed
----- ------ ---------- ---------- --------------------
    1     71      66879      66977 30-mar-1999 11:13:04
    1     72      66977      67066 30-mar-1999 11:13:28
    1     73      67066      67160 30-mar-1999 11:13:43
    1     74      67160      67229 30-mar-1999 11:13:53
    1     75      67229      67303 30-mar-1999 11:14:02
        . . .
    1    248     104705     104716 30-mar-1999 16:04:57
    1    249     104716     104723 30-mar-1999 16:13:46
    1    250     104723     105257 30-mar-1999 16:13:47
    1    251     105257     105963 30-mar-1999 16:28:36
181 rows selected.
Press enter to continue
```

Figure 33: Output From Redo Log Switch Script

Of course, without knowing the current size of the redo logs the above information does us little good, the script in figure 34 will document the size of your redo logs and the location of their files.

```
REM NAME:     log_file.sql
REM FUNCTION: Report on Redo Logs Physical files
REM
COLUMN group#          FORMAT 999999
COLUMN member          FORMAT a40
COLUMN meg             FORMAT 9,999
REM
SET LINES 80 PAGES 60 FEEDBACK OFF VERIFY OFF
START title80 'Redo Log Physical Files'
BREAK ON group#
SPOOL rep_out\&db\rdo_file
REM
SELECT
a.group#,a.member,b.bytes,b.bytes/(1024*1024) meg
FROM
sys.v_$logfile a,
sys.v_$log b
WHERE
a.group#=b.group#
ORDER BY
group#;
SPOOL OFF
CLEAR COLUMNS
CLEAR BREAKS
TTITLE OFF
SET PAGES 22 FEEDBACK ON VERIFY ON
PAUSE Press enter to continue
```

Figure 34: Redo Log Physical File Report

The output from the above script is shown in figure 35.

```
Date: 04/02/99                                          Page:   1
Time: 10:10 AM          Redo Log Physical Files          SYSTEM
                              DMDB database
 GROUP# MEMBER                                      BYTES    MEG
------- ------------------------------------ ---------- ------
      1 C:\ORACLE1\ORTEST1\REDO\LOG4DMDB.ORA     1048576      1
      2 D:\ORACLE2\ORTEST1\REDO\LOG3DMDB.ORA     1048576      1
      3 E:\ORACLE3\ORTEST1\REDO\LOG2DMDB.ORA     1048576      1
      4 F:\ORACLE4\ORTEST1\REDO\LOG1DMDB.ORA     1048576      1
Press enter to continue
```

Figure 35: Example Output of Redo Log File Report

Based on our desire to maximize performance and meet recoverability guidelines (only lose a maximum of an hours data) we need to increase the size of the above redo logs since they are switching about every ten seconds.

Another item to adjust that deals with redo logs is the size of the log buffer. The log buffer is written to in a circular fashion and as the buffer fills (actually at about a third full) the LGWR process starts to write it out to the redo log. Too small a log buffer setting and you will incur excessive IO to the redo logs and work the LGWR to death, too large a value and the writes are delayed. I usually suggest no larger a size than 1 megabyte for the log_buffer parameter and that the size be either equal to or an equal divisor of the actual redo log size. Unless you have very small redo logs the default value for log_buffers supplied by Oracle is too small.

Tuning Rollback Segments

It is difficult if not impossible to proactively tune rollback segments. The reason for this difficulty in the tuning of rollback segments is that they depend on the size of transactions for their sizing information and you usually won't know the size of a transaction until the transaction is running on a production level system. However, once transactions are running in a quasi-production size environment the sizing of rollback segments is made much easier. The views DBA_ROLLBACK_SEGS and V$ROLLSTAT along with V$ROLLNAME are used in an active environment to aid in the sizing estimations. The views shown in Figure 36 help parse the large amount of information in these views into more manageable chunks.

```
REM
REM FUNCTION: create views required for rbk1 and rbk2 reports.
REM
rem exit
CREATE OR REPLACE VIEW rollback1 AS
SELECT
       d.segment_name,extents,
```

```
        optsize,shrinks,
        aveshrink,aveactive,
        d.status
FROM
v$rollname n,
v$rollstat s,
dba_rollback_segs d
WHERE
d.segment_id=n.usn(+)
and d.segment_id=s.usn(+);

CREATE OR REPLACE VIEW rollback2 AS
SELECT
d.segment_name,extents,
xacts,hwmsize,rssize,
waits,wraps,extends
FROM
v$rollname n,
v$rollstat s,
dba_rollback_segs d
WHERE
d.segment_id=n.usn(+)
and d.segment_id=s.usn(+);
```

Figure 36: Views to Parse out Rollback Data

Once the views in figure 36 are created, two simple reports give us the information to derive a best guess estimate of rollback sizing parameters. These reports are shown in figure 37.

```
REM NAME                : RBK1.SQL
REM FUNCTION            : REPORT ON ROLLBACK SEGMENT STORAGE
REM FUNCTION            : USES THE ROLLBACK1 VIEW
REM USE                 : FROM SQLPLUS
REM Limitations         : None
REM
COLUMN hwmsize          FORMAT 9999999999    HEADING 'LARGEST TRANS'
COLUMN tablespace_name  FORMAT a10           HEADING 'TABLESPACE'
COLUMN segment_name     FORMAT A10           HEADING 'ROLLBACK'
COLUMN optsize          FORMAT 9999999999   HEADING 'OPTL|SIZE'
COLUMN shrinks          FORMAT 9999           HEADING 'SHRINKS'
COLUMN aveshrink        FORMAT 9999999999   HEADING 'AVE|SHRINK'
COLUMN aveactive        FORMAT 9999999999   HEADING 'AVE|TRANS'
COLUMN waits            FORMAT 99999              HEADING 'WAITS'
COLUMN wraps            FORMAT 99999              HEADING 'WRAPS'
COLUMN extends          FORMAT 9999              HEADING 'EXTENDS'
rem
BREAK ON REPORT
COMPUTE AVG OF AVESHRINK ON REPORT
COMPUTE AVG OF AVEACTIVE ON REPORT
COMPUTE AVG OF SHRINKS ON REPORT
COMPUTE AVG OF WAITS ON REPORT
COMPUTE AVG OF WRAPS ON REPORT
COMPUTE AVG OF EXTENDS ON REPORT
COMPUTE AVG OF HWMSIZE ON REPORT
SET FEEDBACK OFF VERIFY OFF LINES 132 PAGES 58
```

```
@title132 "ROLLBACK SEGMENT STORAGE"
SPOOL rep_out\&db\rbk1
rem
SELECT * FROM rollback1 ORDER BY segment_name;
SPOOL OFF
CLEAR COLUMNS
TTITLE OFF
SET FEEDBACK ON VERIFY ON LINES 80 PAGES 22
PAUSE Press enter to continue
TTITLE OFF
SET FEEDBACK ON VERIFY ON LINES 80 PAGES 22
PAUSE Press enter to continue

REM NAME        : RBK2.SQL
REM FUNCTION    : REPORT ON ROLLBACK SEGMENT STATISTICS
REM FUNCTION    : USES THE ROLLBACK2 VIEW
REM USE         : FROM SQLPLUS
REM Limitations : None
REM
COLUMN segment_name     FORMAT A10              HEADING 'ROLLBACK'
COLUMN extents          FORMAT 9,999            HEADING 'EXTENTS'
COLUMN xacts            FORMAT 9,999            HEADING 'TRANS'
COLUMN hwmsize          FORMAT 9,999,999,999    HEADING 'LARGEST TRANS'
COLUMN rssize           FORMAT 9,999,999,999    HEADING 'CUR SIZE'
COLUMN waits            FORMAT 999              HEADING 'WAITS'
COLUMN wraps            FORMAT 999              HEADING 'WRAPS'
COLUMN extends          FORMAT 999              HEADING 'EXTENDS'
REM
SET FEEDBACK OFF VERIFY OFF lines 80 pages 58
REM
@title80 "ROLLBACK SEGMENT STATISTICS"
SPOOL rep_out\&db\rbk2
REM
SELECT * FROM rollback2 ORDER BY segment_name;
SPOOL OFF
SET LINES 80 PAGES 20 FEEDBACK ON VERIFY ON
TTITLE OFF
CLEAR COLUMNS
PAUSE Press enter to continue
```

Figure 37: Reports Using ROLLBACK1 and ROLLBACK2

The output from the reports in figure 37 will resemble those shown in figure 38.

```
Date: 12/07/98                                                          Page:
1
Time: 03:38 PM              Rollback Segment Report
DBAUTIL
                              ORTEST1 database
       Tablespace Rollback    Initial      Next    Minimum  Maximum  Current  Optimal
Owner  Name       Segment Name Extent     Extent    Extents  Extents  Extents  Setting
------ ---------- ------------ ---------- ---------- -------- -------- -------- -------
PUBLIC PSRBS2     A09           10485760  10485760        2      121        2
                  A10           10485760  10485760        2      121        2
SYS    PSRBS2     A05              20480     20480        4      249      120 2457600
                  R06            2097152   2097152        4      249        4 8388608
                  R07            2097152   2097152        4      249        4 8388608
                  R09            2097152   2097152        4      249        4 8388608
                  RBSBIG       104857600  52428800        4      120        4
                  R10            2097152   2097152        4      249        4 8388608
                  R08            2097152   2097152        4      249        4 8388608
       SYSTEM     SYSTEM           53248     53248        2      249        4

Date: 12/08/98                                                  Page:   1
Time: 05:15 PM          ROLLBACK SEGMENT STATISTICS             DBAUTIL
                              ORTEST1 database

ROLLBACK  EXTENTS  TRANS  LARGEST TRANS       CUR SIZE WAITS WRAPS EXTENDS
--------- ------- ------ -------------- -------------- ----- ----- -------
A05             4      0      8,433,664      8,433,664     3     5       0
A09             7      0     73,396,224     73,396,224     4     5       0
A10             3      0     31,453,184     31,453,184     1     3       0
R06             4      0     33,746,944      8,433,664     8    34      23
R07             4      0     65,388,544      8,433,664     3     6       0
R08             4      1    103,358,464      8,433,664     0     3       0
R09             4      0     21,090,304      8,433,664     9    21       6
R10             4      0     23,199,744      8,433,664     2    17       7
RBSBIG          4      0    262,139,904    262,139,904     1     0       0
SYSTEM          4      0        241,664        241,664     0     0       0
```

Figure 38: Example Rollback Segment Report

As you can see from looking at the report the ORTEST instance rollback segments are a bit of a mess. We have both public (Owner PUBLIC) and private (Owner SYS) rollback segments, segments that have extent sizes from 20k to 100 megabytes and a mix of segments with and without OPTIMAL set. The second report shows that there are numerous waits, wraps and extends.

In sizing rollback segments your goal should be to reduce waits, wraps and extends (and thus shrinks) to a minimum. The act of extending or shrinking cause recursive SQL which is a performance robber as well as dynamic extension which is another performance robber. By properly sizing rollback segments there should be no shrinks and no waits.

I have found that sizing rollback segments such that the initial and next extents are sized for the average transaction and the optimal is set at the average of the largest transaction waits and shrinks are reduced or eliminated.

Unless you are using a shared server environment I do not suggest using public rollback segments at all. All rollback segments should be sized identically except for a special rollback segment used for large transactions (in this case RBSBIG). Notice how the large rollback segment is mixed in with the smaller rollback segments, this is not a good practice. The large rollback segment should be placed in its own tablespace that has been optimized for its use.

In many cases, large transactions such as batch loads, updates or deletes can be "chunked" to reduce impact on rollback segments and thus prevent the frustration of running out of extents, space, snapshot too old or all three during a large transaction.

Tuning Oracle Sorts

Sorts are done when Oracle performs operations that retrieve information and require the information retrieved to be an ordered set, in other words, sorted. Sorts are done when the following operations are performed:

- Index creation
- Group by or Order by statements
- Use of the distinct operator
- Join operations
- Union, Intersect and Minus set operators.

Each of these operations requires a sort. There is one main indicator that your sorts are going to disk and therefore your sort area in memory is too small. This area is defined by the initialization parameters SORT_AREA_SIZE and SORT_AREA_RETAINED_SIZE in both ORACLE7 and ORACLE8.

The primary indicator is the sorts (disk) statistic shown in Figure 15.42. If this parameter exceeds 10% of the sum of sorts(memory) and sorts(disk) increase the SORT_AREA_SIZE parameter. Large values for this parameter can induce paging and swapping, so be careful you don't over allocate. In ORACLE 7 this area is allocated either directly from memory to each user or, if the multi-threaded server is used (MTS) a section of the shared pool is allocated to each user. In ORACLE8 an extra shared area called the LARGE POOL is used (if it has been initialized).

Under ORACLE7 the SORT_AREA_SIZE parameter controls the maximum sort area. The sort area will be dynamically reallocated down to the size specified by the initialization parameter SORT_AREA_RETAINED_SIZE.

In ORACLE7.2 and later the initialization parameters SORT_DIRECT_WRITES, SORT_WRITE_BUFFER_SIZE and SORT_WRITE_BUFFERS control how needed disk sorts are optimized. By specifying SORT_DIRECT_WRITES to TRUE you an improve your sort times by several fold because this forces writes direct to disk rather than using the buffers. The SORT_WRITE_BUFFER_SIZE parameter should be set such that SORT_WRITE_BUFFERS * SORT_WRITE_BUFFER_SIZE is as large as you dare have it be on your system and still not get swapping. The SORT_WRITE_BUFFERS is a value from 2 to 8 and the SORT_WRITE_BUFFER_SIZE is set between 32 and 64k bytes. Therefore the maximum size this can be will be 8*64k = 512k or half a megabyte.

Some additional sort parameters are SORT_READ_FAC and SORT_SPACEMAP_SIZE. The SORT_READ_FAC parameter assists with sort merges. Set this to between 25-100% of the value of the _DB_BLOCK_MULTIBLOCK_READ_COUNT parameter. The SORT_SPACEMAP_SIZE parameter if set correctly helps with actions such as index builds. The suggested setting is:

```
((total sort bytes/(SORT_AREA_SIZE)) + 64
```

Where:

total sort bytes = (number of records in sort) * (average row length + (2 * No_of_columns))

However setting it higher temporarily isn't harmful and can speed the index build appreciably.

For standard sorts you should set the SORT_AREA_SIZE to the average sort size for your database. The temporary tablespaces initial and next default storage parameter should be set to the value of SORT_AREA_SIZE. For use with parallel query sorts a temporary tablespace should be spread (striped) across as many disks as the degree of parallelism.

On versions that support temporary tablespace specification a temporary tablespace should be used for the target for disk sorts. A temporary tablespace (one created or altered to be TEMPORARY in nature) has greatly reduced space management overhead and thus can speed sorts. Another tip for the tablespaces used for sorting is that it should be striped over as many drives as possible to speed access to the sort sets.

The initialization parameter SORT_MULTIBLOCK_READ_COUNT does for sorts what DB_MULTIBLOCK_READ_COUNT does for full table scans, it forces Oracle to read at least that amount of data specified per merge read pass.

The views that are used to help in the sort tuning process are V$SORT_SEGMENT and V$SORT_USAGE. These views are not populated unless dosk sorts occur. The V$SORT_SEGMENT view contains a single line for each sort operation that gives detailed information about segment size in blocks. If you are getting excessive disk sorts you should query this view to calculate the best possible sort area size. An example query to give average sort area size is shown in Figure 39.

```
REM
REM FUNCTION: Generate a summary of Disk Sort Area Usage
REM
REM disksort.sql
REM
COLUMN value NEW_VALUE bs NOPRINT
SELECT value FROM v$parameter WHERE name='db block size';
START title80 "Instance Disk Area Average Sizes"
SPOOL rep_out\&&db\disk_sort
SELECT
       Tablespace_name,
       COUNT(*) areas,
       (SUM(total_blocks)/COUNT(*))*&&bs avg_sort_bytes
FROM v$sort_segment
GROUP BY tablespace_name;
SPOOL OFF
```

Figure 39: Example Report On Disk Sort Sizes

Optimizer Modes

Essentially there are two optimizer modes: RULE or CHOOSE. CHOOSE must be used if:

- HINTS used
- Mode set to FIRST_ROWS or ALL_ROWS
- New features are to be used

However, CHOOSE has its drawbacks:

- Must use frequent ANALYZE
- Must Use histograms with skewed data

Tuning the Multi-part Oracle8 Buffer Cache

In Oracle8 and Oracle8i the database block buffer has been split into three possible areas, the default, keep and recycle buffer pool areas. It is not required that these three pools be used, only one, the default pool configured with the DB_BLOCK_BUFFERS initialization parameter must be present, the others are "sub" pool to this main pool. How are the various pools used?

Use of the Default Pool

If a table, index or cluster is created with specifying that the KEEP or RECYCLE pool be used for its data, then it is placed in the default pool when it is accessed. This is standard Oracle7 behavior and if no special action is taken to use the other pools then this is also standard Oracle8 and Oracle8I behavior. The initialization parameters DB_BLOCK_BUFFERS and DB_BLOCK_LRU_LATCHES must be set if multiple pools are to be used:

```
DB_BLOCK_BUFFERS = 2000
DB_BLOCK_LRU_LATCHES = 10
```

Use of The KEEP Pool

The KEEP database buffer pool is configured using the BUFFER_POOL_KEEP initialization parameter which looks like so:

```
BUFFER_POOL_KEEP = '100,2'
```

The two specified parameters are the number of buffers from the default pool to assign to the keep pool and the number of LRU (least recently used) latches to assign to the keep pool. The minimum number of buffers assigned to the pool is 50 times the number of assigned latches. The keep pool, as its name implies, is used to store object data that shouldn't be aged out of the buffer pool such as look up information and specific performance enhancing indexes. The objects are assigned to the keep pool through either their creation statement or by specifically assigning them to the pool using the ALTER command. Any blocks already in the default pool are not affected by the ALTER command, only subsequently accessed blocks.

The keep pool should be sized such that it can hold all the blocks from all of the tables created with the buffer pool set to KEEP.

Use of the RECYCLE Pool

The RECYCLE database buffer pool is configured using the BUFFER_POOL_RECYCLE initialization parameter which looks like so:

```
BUFFER_POOL_RECYCLE = '1000,5'
```

The two specified parameters are the number of buffers from the default pool to assign to the recycle pool and the number of LRU (least recently used) latches to assign to the keep pool. The minimum number of buffers assigned to the pool is 50 times the number of assigned latches. The recycle pool, as its name implies, is used to store object data that should be aged out of the buffer pool rapidly such as searchable LOB information. The objects are assigned to the recycle pool through either their creation statement or by specifically assigning them to the pool using the ALTER command. Any blocks already in the default pool are not affected by the ALTER command, only subsequently accessed blocks.

As long as the recycle pool shows low block contention it is sized correctly.

With the above setpoints for the default, keep and recycle pools the default pool would end up with 900 buffers and 3 lru latches.

Tuning the Three Pools

Since the classic method of tuning the shared pool is not available in Oracle8i we must examine new methods to achieve the same ends. This involves looking at what Oracle has provided for tuning the new pools. A new script, **catperf.sql** offers several new views for tuning the Oracle buffer pools. These views are:

- V$BUFFER_POOL **--** Provides static information on pool configuration
- V$BUFFER_POOL_STATISTICS -- Provides Pool related statistics
- V$DBWR_WRITE_HISTOGRAM -- Provides summary information on DBWR write activities
- V$DBWR_WRITE_LOG -- Provides write information for each buffer area.

Of the four new views the V$BUFFER_POOL_STATISTICS view seems the most useful for tuning the buffer pool. The V$BUFFER_POOL_STATISTICS view contains statistics such as **buffer_busy_waits, free_buffer_inspected, dirty_buffers_inspected** and physical write related data.

If a buffer pool shows excessive numbers of dirty_buffers_inspected and high amounts of buffer_busy_waits then it probably needs to be increased in size.

When configuring LRU latches and DBWR processes remember that the latches are assigned to the pools sequentially and to the DBRW processes round robin. The number of LRU processes should be equal to or a multiple of the value of DBWR processes to ensure that the DBRW load is balanced across the processes.

Adding Resources

If all possible tuning has been accomplished then add resources. Resource are either memory, additional CPUs or more disks. However, you should analyze the system to see what will give you the greatest return on investment. If you aren't seeing memory contention then it makes no sense to add memory. If you aren't CPU bound adding CPUs probably won't help (unless you are going to parallel query or multiple DBWR processes.) Likewise if you aren't seeing disk contention then additional disks probably won't buy you much. However, if you are seeing redo log contention or IO contention due to having to share disks between multiple files, then performance gains may be realized by spreading the offending files across multiple disk arrays even if the existing disks aren't IO bound.

All things considered, memory will give the most tuning benefit. On the average memory is 14,000 times faster than disk. Anytime an operation can be moved into memory performance will be improved. Proper caching of indexes and tables, proper sort area sizing and proper sizing of cache and pool areas are critical to proper performance.

It has been said that parallel query is the often sought "make_database_faster" initialization parameter, but only for a properly designed set of tables and indexes. Multiple CPUs will help if you use parallel query since having parallel threads running against insufficient CPUs will make problems worse. Parallel query and multiple processes (DBWR, LGWR, etc) not much help with single CPU system.

Use proper striping (RAID1/0 is usually best performer, RAID5 most dependable) when laying out your tables and indexes. Also consider partitioning and in Oracle8i subpartitioning. Table 3 shows the various RAID levels and what files should be placed on them.

RAID	Type of Raid	Control File	Database File	Redo Log File	Archive Log File
0	Striping	Avoid	OK	Avoid	Avoid
1	Shadowing	Recommended	OK	Recommended	Recom-mended
0+1	Striping and Shadowing	OK	Recom-mended	Avoid	Avoid

RAID	Type of Raid	Control File	Database File	Redo Log File	Archive Log File
3	Striping with static parity	OK	OK	Avoid	Avoid
5	Striping with rotating parity	OK	Recommended if RAID0-1 not available	Avoid	Avoid

Table 3: RAID Recommendations (From Metalink NOTE:45635.1)

Tuning Tables and Indexes

The biggest thing you can do to tune tables is to ensure that tables are properly sized and spread on the disk array. In recent articles the concept of using fixed extent sizes for all objects in a specific tablespace has been discussed. This is an excellent concept for reducing fragmentation problems when used properly. You must still perform sizing on the tables to be sure that you place the table in the proper sizing model. With modern disk arrays and use of RAID, many of the old table structure rules no longer apply, however, even on RAID Oracle blocks are Oracle blocks. Therefore it is still wise to place the fixed length, fixed size, or fixed value columns first and place the variable size or updated columns last in the table order when the table is built. This allows optimal use of the PCTFREE area.

Just like tables, indexes need to be properly sized, ordered and spread on disk array. Indexes like large block sizes since this allows more optimal use of available block space. The order of columns in an index should match the order of columns in the query WHERE clause. Unless the leading column in the index matches the leading column in the where clause an index may not be used. In Oracle8i with query rewrite you can get away with miss-ordered columns but it is not suggested unless done to reduce the indexes clustering factor.

Table Rebuilds

Tables should be rebuilt for the following reasons:

- Excessive numbers of extents
- Excessive amounts of row chaining

- To partition the table
- To move the table to a different tablespace

Even though many suggest that multiple extents aren't harmful to a table or index, if you get excessive extents it makes doing table and index maintenance more difficult due to the excessive calls to FET$ and UET$ tables.

Row chaining occurs when an update forces a row to grow beyond the available free space left in the block for updates. Row chaining results in doubling of your IO to retrieve specific values.

Unless a table is created as a partitioned table from the start you cannot add partitions or make it partitioned later without rebuilding the table entirely.

The REBUILD command can also be used to move a table from one tablespace to another.

Rebuilding Indexes

Indexes should be rebuilt when:

- Index has too many levels
- Index is too broad
- Index clustering factor too high

A Balanced tree is defined as a tree structure in which any path from the root page to any leaf page will traverse the same number of levels. Figure 40 shows the basic structure of a B*Tree index.

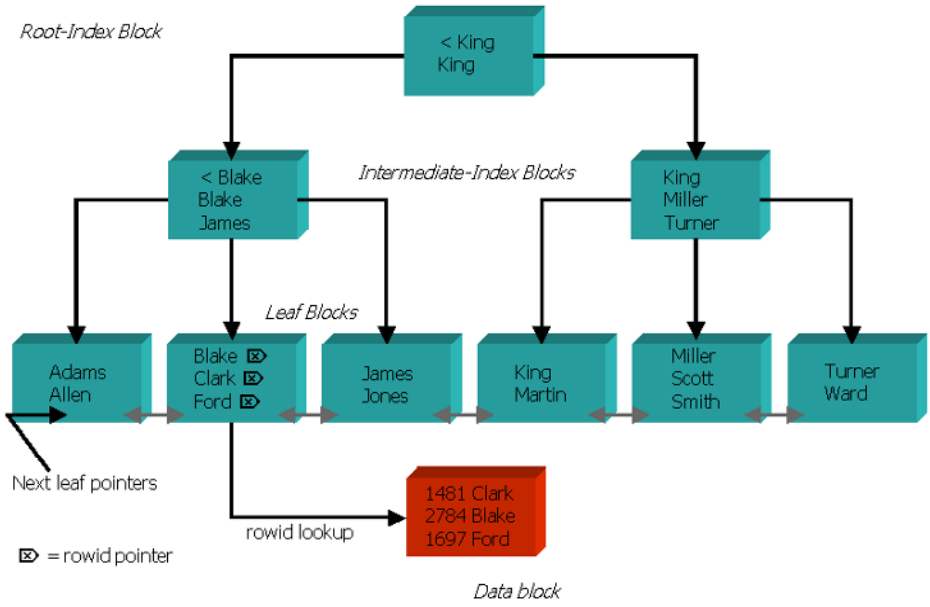


Figure 40: Basic B*Tree structure

When the number of levels gets too deep or the width of the final level gets too broad the performance of the index degrades. This also goes hand-in-hand with a poor clustering factor.

The index clustering factor (CF) determines the price of accessing data via the rowids retrieved from the index. The CF tells you how many data blocks are read in a full index scan. You can determine the actual number of data blocks read by multiplying the CF by the percentage of data to be read. The CF can range between the number of blocks containing data to the number of rows in the tableA high clustering factor can either be implicit in the index design, in which case there is nothing you can do other than reorder the index columns, or it can be caused by an index that has been frequently updated causing block splits. Figures 41 and 42 show indexs with good and bad clustering factors.

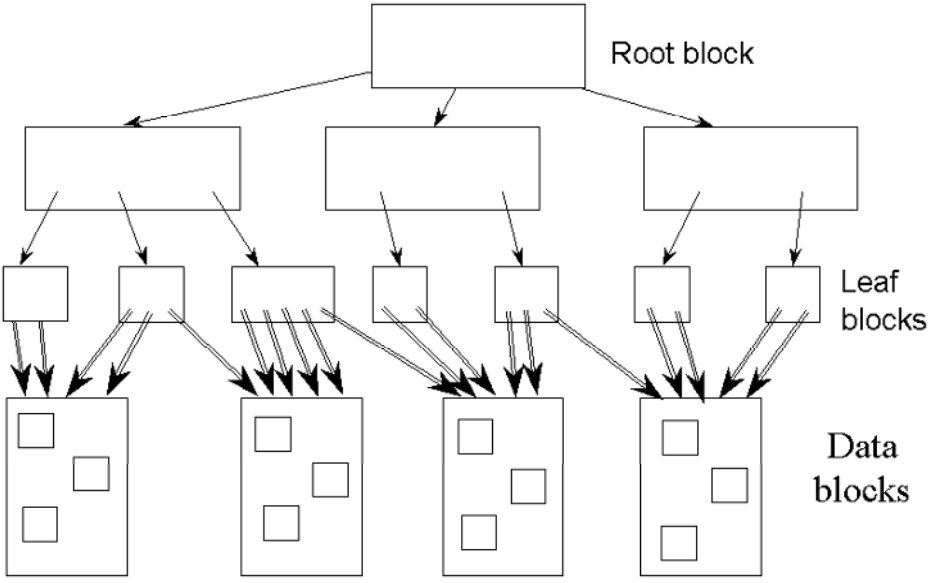


Figure 41: Index with Good (low) Clustering Factor

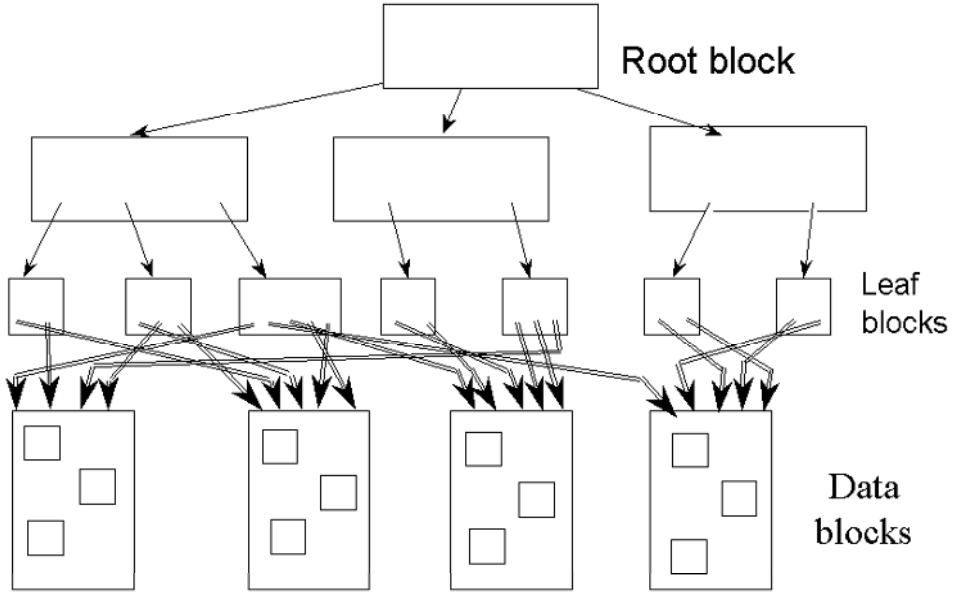


Figure 42: Index with a Bad (high) Clustering Factor

Adjusting Index Cost in Oracle8

In Oracle 8i the total cost of the index is adjusted using the following formula:

Adjusted Cost = Cost * OPTIMIZER_INDEX_COST_ADJ / 100

This adjustment bypasses the Oracle assumption of a low buffer hit ratio used in the data access calculation. In order to ensure that this adjustment is the proper thing to do make sure there is adequate memory allocated to Oracle and that the index is cached.
If you assume that most of the data will be placed in the buffer cache and remain there, you can set **OPTIMIZER_INDEX_COST_ADJ** to a value less than 100, reflecting the percentage of time data will be found in the buffer.

If you set the **OPTIMIZER_INDEX_COST_ADJ** to 10, the price of accessing any index is 10% of the previously calculated price, presuming that data will be found in the buffer cache 90% of the time on average.

However, you must be very careful. It is very tempting to explicitly set a low value for OPTIMIZER_INDEX_COST_ADJ and force index usage in all casesyou must be sure that your buffer cache can support data remaining in the buffer so that the assumption will be correct. You must also be sure that the buffer cache remains in memory and is not paged out (*i.e., make sure you have adequate physical memory!*)

Bitmapped Index Usage*

A bitmapped index is used for low-cardinality data such as sex, race, hair color, and so on. If a column to be indexed has a selectivity of greater than 30 to 40% of the total data then it is probably a good candidate for bitmap indexing.

Bitmap indexing is not suggested for high-cardinality, high-update, or high-delete-type data as in these type situations bitmap indexes may have to be frequently rebuilt.

There are three things to consider when choosing an index method:

- Performance
- Storage
- Maintainability

The major advantages for using bitmapped indexes are: performance impact for certain queries, and their relatively small storage requirements. Note, however, that bitmapped indexes are not applicable to every query and

bitmapped indexes, like B-tree indexes, can impact the performance of insert, update, and delete statements.

Bitmapped indexes can provide very impressive performance improvements. Under test conditions the execution times of certain queries improved by several orders of magnitude. The queries that benefit the most from bitmapped indexes have the following characteristics:

- The WHERE-clause contains multiple predicates on low-cardinality columns.
- The individual predicates on these low-cardinality columns select a large number of rows.
- Bitmapped indexes have been created on some or all of these low-cardinality columns.
- The tables being queried contain many rows.

An advantage of bitmapped indexes is that multiple bitmapped indexes can be used to evaluate the conditions on a single table. Thus, bitmapped indexes are very useful for complex ad hoc queries that contain lengthy WHERE-clauses involving low cardinality data.

Bitmapped indexes incur a small storage cost and have a significant storage savings over B-tree indexes. A bitmapped index can require 100 times less space than a B-tree index for a low-cardinality column.

Remember that a strict comparison of the relative sizes of B-tree and bitmapped indexes is not an accurate measure for selecting bitmapped over B-tree indexes. Because of the performance characteristics of bitmapped indexes and B-tree indexes, you should continue to maintain B-tree indexes on your high-cardinality data. Bitmapped indexes should be considered primarily for your low-cardinality data.

The storage savings are so large because bitmapped indexes replace multiple-column B-tree indexes. In addition, single bit values replace possibly long columnar type data. When using only B-tree indexes, you must anticipate the columns that will commonly be accessed together in a single query and then create a multicolumn B-tree index on those columns. Not only does this B-tree index require a large amount of space, but it will also be ordered; that is, a B-tree index on (MARITAL_STATUS, RACE, SEX) is useless for a query that only accesses RACE and SEX. To completely index the database, you are forced to create indexes on the other permutations of these columns. In addition to an index on (MARITAL_STATUS, RACE, SEX), there is a need for indexes on (RACE, SEX, MARITAL_STATUS), (SEX, MARITAL_STATUS, RACE), etc. For

the simple case of three low-cardinality columns, there are six possible concatenated B-tree indexes.
What this means is that you are forced to decide between disk space and performance when determining which multiple-column B-tree indexes to create.

With bitmapped indexes, the problems associated with multiple-column B-tree indexes is solved because bitmapped indexes can be efficiently combined during query execution. Three small single-column bitmapped indexes are a sufficient functional replacement for six three-column B-tree indexes. Note that while the bitmapped indexes may not be quite as efficient during execution as the appropriate concatenated B-tree indexes, the space savings provided by bitmapped indexes can often more than justify their utilization.

The net storage savings will depend upon a database's current usage of B-tree indexes:

- A database that relies on single-column B-tree indexes on high-cardinality columns will not observe significant space savings (but should see significant performance increases).
- A database containing a significant number of concatenated B-tree indexes could reduce its index storage usage by 50% or more, while maintaining similar performance characteristics.
- A database that lacks concatenated B-tree indexes because of storage constraints will be able to use bitmapped indexes and increase performance with minimal storage costs.

Bitmapped indexes are best for read-only or light OLTP environments. Because there is no effective method for locking a single bit, row-level locking is not available for bitmapped indexes. Instead, locking for bitmapped indexes is effectively at the block level which can impact heavy OLTP environments. Note also that like other types of indexes, updating bitmapped indexes is a costly operation.

Although bitmapped indexes are not appropriate for databases with a heavy load of insert, update, and delete operations, their effectiveness in a data warehousing environment is not diminished. In such environments, data is usually maintained via bulk inserts and updates. For these bulk operations, rebuilding or refreshing the bitmapped indexes is an efficient operation. The storage savings and performance gains provided by bitmapped indexes can provide tremendous benefits to data warehouse users.

In preliminary testing of bitmapped indexes, certain queries ran up to 100 times faster. The bitmapped indexes on low-cardinality columns were also about ten times smaller than B-tree indexes on the same columns. In these tests, the

queries containing multiple predicates on low-cardinality data experienced the most significant speedups. Queries that did not conform to this characteristic were not assisted by bitmapped indexes. Bitmapped composite indexes cannot exceed 30 columns.

The Initialization parameters of concern when dealing with bitmap indexes are:

- CREATE_BITMAP_AREA_SIZE -- Determines the amount of memory allocated for bitmap creation. Default is 8MB. If cardinality is small, this value can be reduced significantly.
- BITMAP_MERGE_AREA_SIZE -- Amount of memory to use for merging bitmap strings. Default value is 1MB. Larger value can improve performance since the bitmap segments must be pre-sorted before being merged into a single bitmap.

Some perfromance characteristics for bitmap indexes are:

Larger block sizes can improve the storing and retrieving of bitmap information. This means more efficient and thus faster operations involving bitmaps.

To compress storage further, use the NOT NULL constraint on bitmap index columns. This is because nulls do exist in bitmap indexes, therefore they can be used to support IS NULL and IS NOT NULL conditions.

Another consideration with bitmap indexes is that an ALTER TABLE command that modifies a bitmap index column may *invalidate* the index structure.

The final thing you should remember about bitmap indexes is that they are not considered by the RBO. In order to use bitmap indexes you must use cost based (CBO) optimization.

Function Based Indexes

New to oracle8i is the concept of a function based index. In previous releases of Oracle if we wanted to have a column that was always searched uppercase (for example a last name that could have mixed case like McClellum) we had to place the returned value with its mixed case letters in one column and add a second column that was upper-cased to index and use in searches. This doubling of columns required for this type of searching lead to doubling of size requirements for some application fields. The cases where more complex such as SOUNDEX and other functions would also have required use of a second column. This is not the case with Oracle8i, now functions and user-defined functions as well as methods can be used in indexes. Let's look at a simple example using the UPPER function.

```
CREATE INDEX tele_dba.up1_clientsv81
ON tele_dba.clientsv81(UPPER(customer_name))
TABLESPACE tele_index
STORAGE (INITIAL 1M NEXT 1M PCTINCREASE 0);
```

In many applications a column may store a numeric value that translates to a minimal set of text values, for example a user code that designates functions such as 'Manager', 'Clerk', or 'General User'. In previous versions of Oracle you would have had to perform a join between a lookup table and the main table to search for all 'Manager' records. With function indexes the DECODE function can be used to eliminate this type of join.

```
CREATE INDEX tele_dba.dec_clientsv81
ON tele_dba.clientsv81(DECODE(user_code,
1,'MANAGER',2,'CLERK',3,'GENERAL USER'))
TABLESPACE tele_index
STORAGE (INITIAL 1M NEXT 1M PCTINCREASE 0);
```

A query against the clientsv8i table that would use the above index would look like:

```
SELECT customer_name FROM tele_dba.clientsv8i
WHERE DECODE(user_code,
1,'MANAGER',2,'CLERK',3,'GENERAL USER')='MANAGER';
```

The explain plan for the above query shows that the index will be used to execute the query:

```
SQL> SET AUTOTRACE ON EXPLAIN
SQL> SELECT customer_name FROM tele_dba.clientsv8i
  2  WHERE DECODE(user_code,
  3* 1,'MANAGER',2,'CLERK',3,'GENERAL USER') = 'MANAGER'

no rows selected

Execution Plan
----------------------------------------------------------------------------
0   SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=526)
1 0 TABLE ACCESS (BY INDEX ROWID)OF 'CLIENTSV8I'(Cost=1 Card=1 Bytes=526)
2 1 INDEX (RANGE SCAN) OF 'DEC_CLIENTSV8I' (NON-UNIQUE) (Cost=1 Card=1)
```

The table using function based indexes must be analyzed, the intialization parameter ENABLE_QUERY_REWRITE set to true, and the optimizer mode set to CHOOSE or the function based indexes will not be used. The RULE based optimizer cannot use function based indexes.

If the function based index is built using a user defined function, any alteration or invalidation of the user function will invalidate the index. Any user built functions must not contain aggregate functions and must be deterministic in nature. A deterministic function is one that is built using the DETERMINISTIC key word in

the CREATE FUNCTION, CREATE PACKAGE or CREATE TYPE commands. A deterministic function is defined as one that always returns the same set value given the same input no matter where the function is executed from within your application.

As of 8.1.5 the validity of the DETERMINISTIC key word usage is not verified and it is left up to the programmer to ensure it is used properly. A function based index cannot be created on a LOB, REF or nested table column or against an object type that contains a LOB, REF or nested table. Let's look at an example of a user defined type (UDT) method.

```
CREATE TYPE room_t AS OBJECT(
lngth NUMBER,
width NUMBER,
MEMBER FUNCTION SQUARE_FOOT
RETURN NUMBER DETERMINISTIC);
/
CREATE TYPE BODY room_t AS
  MEMBER FUNCTION SQUARE_FOOT
  RETURN NUMBER IS
  area NUMBER;
  BEGIN
   AREA:=lngth*width;
    RETURN area
  END;
END;
/
CREATE TABLE rooms OF room_t
TABLESPACE test_data
STORAGE (INITIAL 100K NEXT 100K PCTINCREASE 0);

CREATE INDEX area_idx ON rooms r (r.square_foot());
```

Note: the above example is based on the examples given in the oracle manuals, when tested on 8.1.3 the DETERMINISTIC keyword caused an error, dropping the DETERMINISTIC keyword allowed the type to be created , however, the attempted index creation failed on the alias specification. In 8.1.3 the key word is REPEATABLE instead of DETERMINISTIC, however, even when specified with the REPEATABLE keyword the attempt to create the index failed on the alias.

A function based index is allowed to be either a normal B*tree index or it can also be mapped into a bitmapped format.

Reverse Key Indexes

New in oracle8 are reversed key indexes. A reversed key index prevents unbalancing of the b*-tree and the resulting hot blocking which will happen if the b*-tree becomes unbalanced. Generally, unbalanced b*trees are caused by high volume insert activity in a parallel server where the key value is only slowly changing such as with an integer generated from a sequence or a data value. A

reverse key index works by reversing the order of the bytes in the key value, of course the ROWID value is not altered, just the key value. The only way to create a reverse key index is to use the CREATE INDEX command, an index that is not reverse key cannot be altered or rebuilt into a reverse key index, however, a reverse key index can be rebuilt to be a normal index.

One of the major limitations of reverse key indexes are that they cannot be used in an index range scan since reversing the index key value randomly distributes the blocks across the index leaf nodes. A reverse key index can only use the fetch-by-key or full-index(table)scans methods of access. Let's look at an example.

```
CREATE INDEX rpk_po ON tele_dba.po(po_num) REVERSE
TABLESPACE tele_index
STORAGE (INITIAL 1M NEXT 1M PCTINCREASE 0);
```

The above index would reverse the values for the po_num column is it creates the index. This would assure random distribution of the values across the index leaf-nodes. But what if we then determine that the benefits of the reverse key do not out weigh the draw backs? We can use the ALTER command to rebuild the index as a noreverse index:

```
ALTER INDEX rpk_po REBUILD NOREVERSE;
ALTER INDEX rpk_po RENAME TO pk_po;
```

While the manuals only discuss the benefits of the reverse key index in the realm of Oracle Parallel Server, if you experience performance problems after a bulk load of data, dropping and recreating the indexes involved as reverse key indexes may help if the table will continue to be loaded in a bulk fashion.

Index Organized Tables

Index organized tables have been around since Oracle8.0. If neither the HASH or INDEX ORGANIZED options are used with the create table command then a table is created as a standard hash table. If the INDEX ORGANIZED option is specified, the table is created as a B-tree organized table identical to a standard Oracle index created on similar columns. Index organized tables do not have rowids.

Index organized tables have the option of allowing overflow storage of values that exceed optimal index row size as well as allowing compression to be used to reduce storage requirements. Overflow parameters can include columns to overflow as well as the percent threshold value to begin overflow. An index organized table must have a primary key. Index organized tables are best suited for use with queries based on primary key values. Index organized tables can be partitioned in Oracle8i as long as they do not contain LOB or nested table types.

The pcthreshold value specifies the amount of space reserved in an index block for row data, if the row data length exceeds this value then the row(s) are stored in the area specified by the OVERFLOW clause. If no overflow clause is specified rows that are too long are rejected. The INCLUDING COLUMN clause allows you to specify at which column to break the record if an overflow occurs. For example:

```
CREATE TABLE test8
  ( doc_code CHAR(5),
    doc_type INTEGER,
    doc_desc VARCHAR(512),
       CONSTRAINT pk_docindex PRIMARY KEY (doc_code,doc_type) )
  ORGANIZATION INDEX TABLESPACE data_tbs1
  PCTTHRESHOLD 20 INCLUDING doc_type
  OVERFLOW TABLESPACE data_tbs2
/
```

In the above example the IOT **test8** has three columns, the first two of which make up the key value. The third column in **test8** is a description column containing variable length text. The PCTHRESHOLD is set at 20 and if the threshold is reached the overflow goes into an overflow storage in the **data_tbs2** tablespace with any values of **doc_desc** that won't fit in the index block. Note that you will the best performance from IOTs when the complete value is stored in the IOT structure, otherwise you end up with an index and table lookup as you would with a standard index-table setup.

Partitioned Tables and Indexes

In Oracle8 we have true table and index partitioning where the system maintains range partitioning, maintains indexes and all operations are supported against the partitioned tables. Partitions are good because:

- Each partition is treated logically as its own object. It can be dropped, split or taken offine without affecting other partitions in the same object.
- Rows inside partitions can be managed separately from rows in other partitions in the same object. This is supported by the extended partition syntax.
- Maintenance can be performed on individual partitions in an object, this is all known as partition independence.
- Storage values (initial, next, ext) can be different between individual partitions or can be inherited.
- Partitions can be loaded without affecting other partitions.

A partitioned table in Oracle8 is range partitioned, for example on month, day, year or some other integer or numeric value. This makes partitioning of tables ideal for the time-based data that is the main-stay of data warehousing.

So an accounts payable table would become:

```
CREATE TABLE acct_pay_99 (acct_no NUMBER, acct_bill_amt NUMBER, bill_date
DATE, paid_date DATE, penalty_amount NUMBER, chk_number NUMBER)
STORAGE (INITIAL 40K NEXT 40K PCTINCREASE 0)
PARTITION BY RANGE (paid_date)
(
PARTITION acct_pay_jan99
   VALUES LESS THAN (TO_DATE('01-feb-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_feb99
   VALUES LESS THAN (TO_DATE('01-mar-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_mar99
   VALUES LESS THAN (TO_DATE('01-apr-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_apr99
   VALUES LESS THAN (TO_DATE('01-may-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_may99
   VALUES LESS THAN (TO_DATE('01-jun-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_jun99
   VALUES LESS THAN (TO_DATE('01-jul-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_jul99
   VALUES LESS THAN (TO_DATE('01-aug-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_aug99
   VALUES LESS THAN (TO_DATE('01-sep-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_sep99
   VALUES LESS THAN (TO_DATE('01-oct-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_oct99
   VALUES LESS THAN (TO_DATE('01-nov-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_nov99
   VALUES LESS THAN (TO_DATE('01-dec-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay11,
PARTITION acct_pay_dec99
   VALUES LESS THAN (TO_DATE('01-jan-2000','DD-mon-YYYY'))
   TABLESPACE acct_pay12,
PARTITION acct_pay_2000
   VALUES LESS THAN (MAXVALUE))
   TABLESPACE acct_pay_max
/
```

The above command results in a partitioned table that can be treated as a single table for all inserts, updates and deletes or, if desired, the individual partitions can be addressed. In addition the indexes created will be by default local indexes

that are automatically partitioned the same way as the base table. Be sure to specify tablespaces for the index partitions or they will be placed with the table partitions.

In the example the paid_date is the partition key which can have up to 16 columns included. Deciding the partition key can be the most vital aspect of creating a successful partitioned table. I suggest using the UTLSIDX.SQL script series to determine the best combination of key values. The UTLSIDX.SQL script series is documented in the script headers for UTLSIDX.SQL, UTLOIDXS.SQL and UTLDIDXS.SQL script SQL files. Essentially you want to determine how many key values or concatenated key values there will be and how many rows will correspond to each key value set. In many cases it will be important to balance rows in each partition so that IO is balanced. However in other cases you may want hard separation based on the data ranges and you don't really care about the number of records in each partition, this needs to be determined on a warehouse-by-warehouse basis.

Partitioned Indexes

An index can be range partitioned *unless*:

- It is a cluster index
- It is defined on a clustered table

Oracle supports three types of partitioned indexes:

- Local Prefixed
- Local Non-Prefixed
- Global Prefixed

A local index is defined as an index that is *equi-partitioned* with the underlying base table, i.e., all keys in a given index partition refer only to rows stored in the single related table partitionLocal index partitions are automatically maintained as table partitions are inserted, dropped or split.

A global partitioned index is defined as an index in which the keys in a given index partition may refer to rows in *more than one* underlying table partition.Generally not equi-partitioned with the tableGlobal partitioned indexes offer better performance via index partition “pruning”In Oracle, global indexes *must* be prefixed.

A partitioned index is said to be *local prefixed* if it is partitioned based on the value of the *left-most* column(s) in the key. A partitioned index is said to be *local non-prefixed* if it is partitioned based on the value of any column(s) *other than* the left-most index column(s). A partitioned index is said to be *global prefixed* if it is partitioned based on the value of the *left-most* column(s) in the key, which *differs* from the table partition key.

Global indexes can offer performance benefits as a result of partition pruning but they can potentially reduce availability by preventing partition-level maintenance operations. Conversely, local indexes improve maintainability and availability but can be more I/O intensive to scan.

Parallel Query

Parallel query was first offered in Oracle version 7. However, in Oracle 7 the implementation was rather weak and sometimes generated questionable results. Parallel Query in Oracle8 and 8i is more stable and offers a great performance boost to specific types of SQL activities.

Parallel query splits the query into multiple segments and then assigns a segment of the query or operation to each available parallel query slave based on the settings for degree of parallel. This allows for maximal usage of multiple CPUs.

Parallel query works best if table and indexes are partitioned. The value for degree of parallel is set at database, table or index level. There are numerous tuning options available for parallel query in 8 and 8i allowing a very fine degree of control.

Oracle8 Enhanced Parallel DML

To use parallel anything in Oracle8 the parallel server parameters must be set properly in the initialization file, these parameters are:

- COMPATIBLE Set this to at least 8.0
- CPU_COUNT this should be set to the number of CPUs on your server, if it isn't set it manually.

- DML_LOCKS set to 200 as a start for a parallel system.
- ENQUEUE_RESOURCES set this to DML_LOCKS+20
- OPTIMIZER_PERCENT_PARALLEL this defaults to 0 favoring serial plans, set to 100 to force all possible parallel operations or somewhere in between to be on the fence.
- PARALLEL_MIN_SERVERS set to the minimum number of parallel server slaves to start up.
- PARALLEL_MAX_SERVERS set to the maximum number of parallel slaves to start, twice the number of CPUs times the number of concurrent users is a good beginning.
- SHARED_POOL_SIZE set to at least ((3*msgbuffer_size)*(CPUs*2)*PARALLEL_MAX_SERVERS) bytes + 40 megabytes
- ALWAYS_ANTI_JOIN Set this to HASH or NOT IN operations will be serial.
- SORT_DIRECT_WRITES Set this to AUTO

DML, data manipulation language, what we know as INSERT, UPDATE and DELETE as well as SELECT can use parallel processing, the list of parallel operations supported in Oracle8 is:

- Table scan
- NOT IN processing
- GROUP BY processing
- SELECT DISTINCT
- AGGREGATION
- ORDER BY
- CREATE TABLE x AS SELECT FROM y;
- INDEX maintenance
- INSERT INTO x ... SELECT ... FROM y
- Enabling constraints (index builds)
- Star transformation

In some of the above operations the table has to be partitioned to take full advantage of the parallel capability. In some releases of Oracle8 you have to explicitly turn on parallel DML using the ALTER SESSION command:

```
ALTER SESSION ENABLE PARALLEL DML;
```

Remember that the COMPATIBLE parameter must be set to at least 8.0.0 to get parallel DML. Also, parallel anything doesn't make sense if all you have is one CPU. Make sure that your CPU_COUNT variable is set correctly, this should be automatic but problems have been reported on some platforms.

Oracle8 supports parallel inserts, updates, and deletes into partitioned tables. It also supports parallel inserts into non-partitioned tables. The parallel insert operation on a non-partitioned table is similar to the direct path load operation that is available in Oracle7. It improves performance by formatting and writing disk blocks directly into the datafiles, bypassing the buffer cache and space management bottlenecks. In this case, each parallel insert process inserts data into a segment above the high watermark of the table. After the transaction commits, the high watermark is moved beyond the new segments.

To use parallel DML, it must be enabled prior to execution of the insert, update, or delete operation. Normally, parallel DML operations are done in batch programs or within an application that executes a bulk insert, update, or delete. New hints are available to specify the parallelism of DML statements.

I suggest using explain plan and tkprof to verify that operations you suspect are parallel are actually parallel. If you find for some reason Oracle isn't doing parallel processing for an operation which you feel should be parallel, use the parallel hints to force parallel processing:

- PARALLEL
- NOPARALLEL
- APPEND
- NOAPPEND
- PARALLEL_INDEX

An example would be:

```
SELECT /*+ FULL(clients) PARALLEL(clients,5,3)*/ client_id, client_name,
client_address FROM clients;
```

By using hints the developer and tuning DBA can exercise a high level of control over how a statement is processed using the parallel query option. The

initialization parameters for use with Oracle Parallel Query in all versions is shown in Table 4.

PARAMETER	DESCRIPTION
In Oracle7	
PARALLEL_MAX_SERVERS	The maximum number of parallel query slaves
PARALLEL_MIN_SERVERS	The minimum number of parallel query slaves
PARALLEL_MIN_PERCENT	Sets the minimum percent of query slaves which must be available.
PARALLEL_SERVER_IDLE_TIME	Max allowed idle time in minutes before a slave is terminated.
In Oracle8	
OPTIMIZER_PERCENT_PARALLEL	If set to non-zero allows optimizer to look at DOP when calculating cost. Low favors indexes, high favors tables.
PARALLEL_ADAPTIVE_MULTI_ USER	If set, reduces DOP based on user load
PARALLEL_BROADCAST_ ENABLED	Improves parallel performance in hash and merge joins.
PARALLEL_EXECUTION_ MESSAGE_SIZE	Specifies the size of the parallel execution messages.
PARALLEL_MIN_MESSAGE_POOL	Minimum amount of shared_pool allocated to parallel query if no large pool configured.
2 UNDOCUMENTED PARAMETERS	
In Oracle8i	
PARALLEL_AUTOMATIC_TUNING	Provides for fully automatic tuning of parallel query processing, overrides PARALLEL_ADAPTIVE_MULTI_USER if set.
PARALLEL_THREADS_PER_CPU	Sets number of threads or processes a CPU can handle during a parallel query operation.
11 UNDOCUMENTED PARAMETERS	

Table 4: Parallel Query Initialization Parameters

Managing Multiple Buffer Pools in Oracle8

In Oracle8 and Oracle8i the database block buffer has been split into three possible areas, the default, keep and recycle buffer pool areas. It is not required that these three pools be used, only one, the default pool configured with the DB_BLOCK_BUFFERS initialization parameter must be present, the others are "sub" pool to this main pool. How are the various pools used?

Use of the Default Pool

If a table, index or cluster is created with specifying that the KEEP or RECYCLE pool be used for its data, then it is placed in the default pool when it is accessed. This is standard Oracle7 behavior and if no special action is taken to use the other pools then this is also standard Oracle8 and Oracle8I behavior. The initialization parameters DB_BLOCK_BUFFERS and DB_BLOCK_LRU_LATCHES must be set if multiple pools are to be used:

```
DB_BLOCK_BUFFERS = 2000
DB_BLOCK_LRU_LATCHES = 10
```

Use of The KEEP Pool

The KEEP database buffer pool is configured using the BUFFER_POOL_KEEP initialization parameter which looks like so:

```
BUFFER_POOL_KEEP = '100,2'
```

The two specified parameters are the number of buffers from the default pool to assign to the keep pool and the number of LRU (least recently used) latches to assign to the keep pool. The minimum number of buffers assigned to the pool is 50 times the number of assigned latches. The keep pool, as its name implies, is used to store object data that shouldn't be aged out of the buffer pool such as look up information and specific performance enhancing indexes. The objects are assigned to the keep pool through either their creation statement or by specifically assigning them to the pool using the ALTER command. Any blocks already in the default pool are not affected by the ALTER command, only subsequently accessed blocks.

The keep pool should be sized such that it can hold all the blocks from all of the tables created with the buffer pool set to KEEP.

Use of the RECYCLE Pool

The RECYCLE database buffer pool is configured using the BUFFER_POOL_RECYCLE initialization parameter which looks like so:

```
BUFFER_POOL_RECYCLE = '1000,5'
```

The two specified parameters are the number of buffers from the default pool to assign to the recycle pool and the number of LRU (least recently used) latches to assign to the keep pool. The minimum number of buffers assigned to the pool is 50 times the number of assigned latches. The recycle pool, as its name implies, is used to store object data that should be aged out of the buffer pool rapidly such as searchable LOB information. The objects are assigned to the recycle pool through either their creation statement or by specifically assigning them to the pool using the ALTER command. Any blocks already in the default pool are not affected by the ALTER command, only subsequently accessed blocks.

As long as the recycle pool shows low block contention it is sized correctly.

With the above setpoints for the default, keep and recycle pools the default pool would end up with 900 buffers and 3 lru latches.

Sizing the Default Pool

The default pool holds both the keep and recycle pools, it must be sized according to the following formula as a minimum:

> Default = (keep + recycle + (total_of_non-keep_or_recycle_object_sizes/100))/DB_BLOCK_SIZE

Each object not explicitly assigned to the keep or recycle pools will be placed into the default pool when it is accessed. As a general rule of thumb the data currently in use will be equal to approximately 5 to 10 percent of the physical database objects such as tables, clusters and indexes. I suggest to start at five percent and move up from there.

Sizing the Keep Pool

The keep buffer pool should be sized to the total size of the data objects that are explicitly signed to the pool. Remember, the keep pool is designed to hold objects that would have been cached in earlier versions of Oracle. Generally speaking small indexes, lookup tables, small active data tables are good candidates for the keep pool. To size the pool you must have a good estimate of the size of the objects you want to keep.

Sizing the Recycle Pool

Probably the most difficult pool to size will be the recycle pool. The reason the recycle pool is difficult to size is that it is designed to hold transient data objects (such as chunks of LOB data items.) I would suggest to size the recycle pool according to the following formula:

> Recycle = (SUM(size_non_lob_object(1-n)/20) + (lob_chunk_size_i(1-n) * No_simul_accesses_i))

The first part of this formula is for non-lob objects that might be searched in large pieces such as partitioned tables. If you can find the partition size then exclude the divide by 20 and just use the partition size.

The second half of the formula addresses LOB (BLOB, CLOB, NCLOB) type objects that will be accessed in chunks such as for searching or comparing using piece-wise logic. The specified chunk size for each assigned object times the number of expected different simultaneous accesses is used to derive the area size required.

The sum of the above two numbers should give a size for the recycle pool.

Tuning the Three Pools

Since the classic method of tuning the shared pool is not available in Oracle8i we must examine new methods to achieve the same ends. This involves looking at what Oracle has provided for tuning the new pools. A new script, **catperf.sql** offers several new views for tuning the Oracle buffer pools. These views are:

- V$BUFFER_POOL **--** Provides static information on pool configuration
- V$BUFFER_POOL_STATISTICS -- Provides Pool related statistics
- V$DBWR_WRITE_HISTOGRAM -- Provides summary information on DBWR write activities
- V$DBWR_WRITE_LOG -- Provides write information for each buffer area.

Of the four new views the V$BUFFER_POOL_STATISTICS view seems the most useful for tuning the buffer pool. The V$BUFFER_POOL_STATISTICS view contains statistics such as **buffer_busy_waits, free_buffer_inspected, dirty_buffers_inspected** and physical write related data for each of the pool areas.

If a buffer pool shows excessive numbers of dirty_buffers_inspected and high amounts of buffer_busy_waits then it probably needs to be increased in size.

When configuring LRU latches and DBWR processes remember that the latches are assigned to the pools sequentially and to the DBRW processes round robin. The number of LRU processes should be equal to or a multiple of the value of DBWR processes to ensure that the DBRW load is balanced across the processes.

Using Outlines in Oracle8i

In versions of Oracle prior to Oracle8i the only way to stabilize an execution plan was to ensure that tables where analyzed frequently and that the relative ratios of rows in the tables involved stayed relatively stable. Neither of these options in pre-Oracle8i for stabilizing execution plans worked 100 percent of the time. In Oracle8i a new feature known as OUTLINEs has been added.

New in Oracle8i is the OUTLINE capability. An outline allows the DBA to tune a SQL statement and then store the optimizer plan for the statement in what is known as an OUTLINE. From that point forward whenever an identical SQL statement to the one in the OUTLINE is used, it will use the optimizer instructions contained in the OUTLINE.

This storing of plan outlines for SQL statements is known as plan stability and insures that changes in the Oracle environment don't affect the way a SQL statement is optimized by the cost based optimizer. If you wish, Oracle will define plans for all issued SQL statements at the time they are executed and this stored plan will be reused until altered or dropped. Generally I do not suggest using the automatic outline feature as it can lead to poor plans being reused by the optimizer. It makes more sense to monitor for high cost statements and tune them as required, storing an outline for them only once they have been properly tuned.

The use of OUTLINES also facilitates the tuning of systems where the code cannot be changed. This is accomplished through the concept of “stealth hints” that is, hints that are applied at parse time but are otherwise invisible. An example use of this technique would be where there are two indexes that, due to the way cost is figured, are not being used properly for a specific query. By dropping the offending index, creating an outline with the proper index being used and then recreating the index that was dropped you can force the use of a specific index without changing the code.

As with the storage of SQL in the shared pool, storage of outlines depends on the statement being reissued in an identical fashion each time it is used. If even one space is out of place the stored outline is not reused. Therefore your queries should be stored as PL/SQL procedures, functions or packages (or perhaps Java routines) and bind variables should always be used. This allows reuse of the stored image of the SQL as well as reuse of stored outlines.

Remember that to be useful over the life of an application the outlines will have to be periodically verified by checking SQL statement performance. If performance of SQL statements degrades the stored outline may have to be dropped and regenerated after the SQL is retuned.

Creation of a OUTLINE object

Outlines are created using the CREATE OUTLINE command, the syntax for this command is:

```
CREATE [OR REPLACE] OUTLINE outline_name
[FOR CATEGORY category_name]
ON statement;
```

Where:

- Outline_name -- is a unique name for the outline
- [FOR CATEGORY category_name] – This optional clause allows more than one outline to be associated with a single query by specifying multiple catagories each named uniquely.
- ON statement – This specifies the statement for which the outline is prepared.

An example would be:

```
CREATE OR REPLACE OUTLINE get_tables
ON
SELECT
a.owner,
a.table_name,
a.tablespace_name,
SUM(b.bytes),
COUNT(b.table_name) extents
FROM
       dba_tables a,
       dba_extents b
WHERE
       a.owner=b.owner
       AND a.table_name=b.table_name
GROUP BY
       a.owner, a.table_name, a.tablespace_name;
```

Assuming the above select is a part of a stored PL/SQL procedure or perhaps part of a view, the stored outline will now be used each time an exactly matching SQL statement is issued.

Altering a OUTLINE

Outlines are altered using the ALTER OUTLINE or CREATE OR REPLACE form of the CREATE command. The format of the command is identical whether it is used for initial creation or replacement of an existing outline. For example, what if we want to add SUM(b.blocks) to the previous example?

```
CREATE OR REPLACE OUTLINE get_tables
ON
SELECT
a.owner,
a.table_name,
a.tablespace_name,
SUM(b.bytes),
COUNT(b.table_name) extents,
SUM(b.blocks)
FROM
       dba_tables a,
       dba_extents b
WHERE
       a.owner=b.owner
       AND a.table_name=b.table_name
GROUP BY
       a.owner, a.table_name, a.tablespace_name;
```

The above example has the effect of altering the stored outline *get_tables* to include any changes brought about by inclusion of the SUM(b.blocks) in the SELECT list. But what if we want to rename the outline or change a category name? The ALTER OUTLINE command has the format:

```
ALTER OUTLINE outline_name
[REBUILD]
[RENAME TO new_outline_name]
[CHANGE CATEGORY TO new_category_name]
```

The ALTER OUTLINE command allows us to rebuild the outline for an existing outline_name as well as rename the outline or change its category. The benefit of using the ALTER OUTLINE command is that we do not have to respecify the complete SQL statement as we would have to using the CREATE OR REPLACE command.

Dropping an OUTLINE

Outlines are dropped using the DROP OUTLINE command the syntax for this command is:

```
DROP OUTLINE outline_name;
```

Use of the OUTLN_PKG To Manage SQL Stored Outlines

The OUTLN_PKG package provides for the management of stored outlines. A stored outline is an execution plan for a specific SQL statement. A stored outline permits the optimizer to stabilize a SQL statements execution plan giving repeatable execution plans even when data and statistics change.

The DBA should take care to who they grant execute on the OUTLN_PKG, by default it is not granted to the public user group nor is a public synonym created.

The following sections show the packages in the OUTLN_PKG.

DROP_UNUSED

The drop_unused procedure is used to drop outlines that have not been used in the compilation of SQL statements. The drop_unused procedure has no arguments.

```
SQL> EXECUTE OUTLN_PKG.DROP_UNUSED;

PL/SQL procedure successfully executed.
```

To determine if a SQL statement OUTLINE is unused, perform a select against the DBA_OUTLINES view:

```
SQL> desc dba_outlines;
 Name                              Null?    Type
 --------------------------------- -------- ----
 NAME                                       VARCHAR2(30)
 OWNER                                      VARCHAR2(30)
 CATEGORY                                   VARCHAR2(30)
 USED                                       VARCHAR2(9)
 TIMESTAMP                                  DATE
 VERSION                                    VARCHAR2(64)
 SQL_TEXT                                   LONG

SQL> set long 1000
SQL> select * from dba_outlines where used='UNUSED';
```

```
NAME         OWNER  CATEGORY USED   TIMESTAMP VERSION    SQL_TEXT
------------ ------ -------- ------ --------- ---------- ----------------------
TEST_OUTLINE SYSTEM TEST     UNUSED 08-MAY-99 8.1.3.0.0  select a.table_name,
                                                         b.tablespace_name,
                                                         c.file_name from
                                                         dba_tables a,
                                                         dba_tablespaces b,
                                                         dba_data_files c
                                                         where
                                                         a.tablespace_name =
                                                         b.tablespace_name
                                                         and b.tablespace_name
                                                         = c.tablespace_name
                                                         and c.file_id =
                                                         (select
                                                         min(d.file_id) from
                                                         dba_data_files d
                                                         where
                                                         c.tablespace_name =
                                                         d.tablespace_name)


1 row selected.

SQL> execute sys.outln_pkg.drop_unused;

PL/SQL procedure successfully completed.

SQL> select * from dba_outlines where used='UNUSED';

no rows selected
```

Remember, the procedure drops all unused outlines so use it carefully.

DROP_BY_CAT

The drop_by_cat procedure drops all outlines that belong to a specific category. The procedure drop_by_cat has one input variable, cat, a VARCHAR 2 that corresponds to the name of the category you want to drop.

```
SQL> create outline test_outline for category test on
  2  select a.table_name, b.tablespace_name, c.file_name from
  3  dba_tables a, dba_tablespaces b, dba_data_files c
  4  where
  5  a.tablespace_name=b.tablespace_name
  6  and b.tablespace_name=c.tablespace_name
  7  and c.file_id = (select min(d.file_id) from dba_data_files d
  8  where c.tablespace_name=d.tablespace_name)
  9  ;
Operation 180 succeeded.

SQL> select * from dba_outlines where category='TEST';
```

```
NAME         OWNER  CATEGORY USED   TIMESTAMP VERSION    SQL_TEXT
------------ ------ -------- ------ --------- ---------- -------------------------
TEST_OUTLINE SYSTEM TEST     UNUSED 08-MAY-99 8.1.3.0.0  select a.table_name, b.ta
                                                         blespace_name, c.file_nam
                                                         e from
                                                         dba_tables a, dba_tablesp
                                                         aces b, dba_data_files c
                                                         where
                                                         a.tablespace_name=b.table
                                                         space_name
                                                         and b.tablespace_name=c.t
                                                         ablespace_name
                                                         and c.file_id = (select m
                                                         in(d.file_id) from dba_da
                                                         ta_files d
                                                         where c.tablespace_name=d
                                                         .tablespace_name)


1 row selected.

SQL> execute sys.outln_pkg.drop_by_cat('TEST');

PL/SQL procedure successfully completed.

SQL> select * from dba_outlines where category='TEST';

no rows selected
```

UPDATE_BY_CAT

The update_by_cat procedure changes all of the outlines in one category to a new category. If the SQL text in an outline already has an outline in the target category, then it is not merged into the new category. The procedure has two input variables, oldcat VARCHAR2 and newcat VARCHAR2 where oldcat corresponds to the category to be merged and newcat is the new category that oldcat is to be merged with.

```
SQL> create outline test_outline for category test on
  2  select a.table_name, b.tablespace_name, c.file_name from
  3  dba_tables a, dba_tablespaces b, dba_data_files c
  4  where
  5  a.tablespace_name=b.tablespace_name
  6  and b.tablespace_name=c.tablespace_name
  7  and c.file_id = (select min(d.file_id) from dba_data_files d
  8  where c.tablespace_name=d.tablespace_name)
  9  ;

Operation 180 succeeded.

SQL> create outline test_outline2 for category test on
  2  select * from dba_data_files;

Operation 180 succeeded.

SQL> create outline prod_outline1 for category prod on
  2  select owner,table_name from dba_tables;
```

```
Operation 180 succeeded.

SQL> create outline prod_outline2 for category prod on
  2  select * from dba_data_files;

Operation 180 succeeded.

SQL>  select name,category from dba_outlines order by category
NAME           CATEGORY
-------------- --------
PROD_OUTLINE1   PROD
PROD_OUTLINE2   PROD
TEST_OUTLINE2   TEST
TEST_OUTLINE    TEST

4 rows selected.

SQL> execute sys.outln_pkg.update_by_cat('TEST','PROD');

PL/SQL procedure successfully completed.

SQL> select name,category from dba_outlines order by category;
NAME           CATEGORY
-------------- --------
TEST_OUTLINE    PROD
PROD_OUTLINE1   PROD
PROD_OUTLINE2   PROD
TEST_OUTLINE2   TEST

4 rows selected.
```

As a result of the update_by_cat procedure call we moved the TEST_OUTLINE outline into the PROD category, but the TEST_OUTLINE2, since it is a duplicate of PROD_OUTLINE2, was not merged.

Summary

The OUTLN_PKG is a powerful new feature in Oracle. By its capability to add hints to Oracle SQL statements without altering code it allows the DBA greater flexibility in tuning “hands off” systems than ever before.

Using Oracle8i Resource Plans and Groups

New in Oracle8i is the concept of Oracle resource groups. A resource group specification allows you to specify that a specific group of database users can only use a certain percentage of the CPU resources on the system. A resource plan must be developed that defines the various levels within the application and their percentage of CPU resources in a waterfall type structure where each subsequent levels percentages are based on the previous levels.

Creating a Resource Plan

Rather than have a simple CREATE RESOURCE PLAN command, Oracle8i has a series of packages which must be run in a specific order to create a proper resource plan. All resource plans are created in a pending area before being validated and committed to the database. The requirements for a valid resource plan are outlined in the definition of the DBMS_RESOURCE_MANAGER.VALIDATE_PENDING_AREA procedure below.
Resource plans can have up to 32 levels with 32 groups per level allowing the most complex resource plan to be easily grouped. Multiple plans, sub-plans and groups can all be tied together into an application spanning CPU resource utilization rule set. This rule set is known as a set of directives.

An example would be a simple 2-tier plan like that shown in Figure 41.

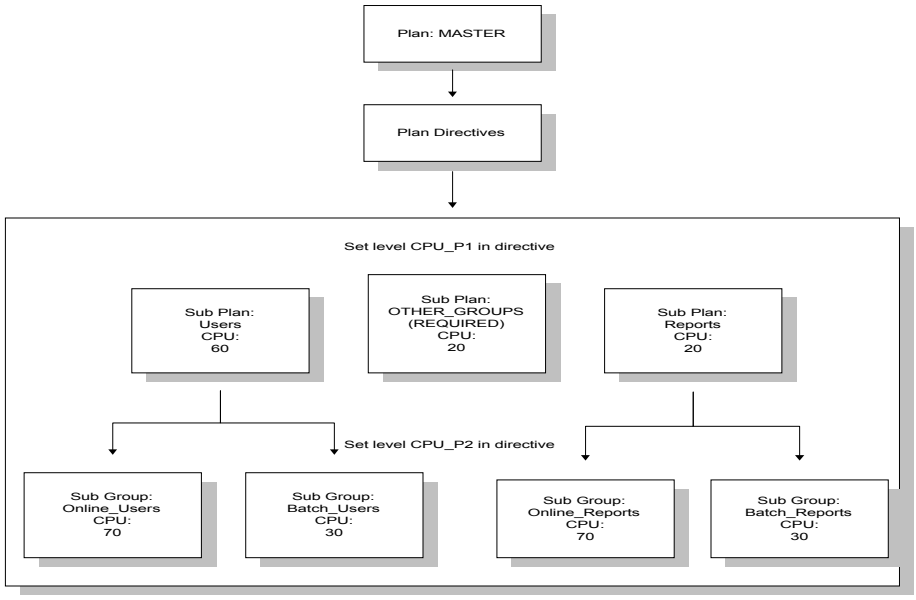


Figure 41 Example Resource Plan

An example of how this portioning out of CPU resources works would be to examine what happens in the plan shown in Figure 41. In figure 41 we have a top level called MASTER which can have up to 100% of the CPU if it requires it. The next level of the plan creates two sub-plans, USERS and REPORTS which will get maximums of 60 and 20 percent of the CPU respectively (we also have the required plan OTHER_GROUPS to which we have assigned 20 percent, if a user is not assigned to a specific group, they get OTHERS). Under USERS we have two groups, ONLINE_USERS and BATCH_USERS. ONLINE_USERS gets 70 percent of USERS 60 percent or an overall percent of CPU of 42 percent while the other sub-group, BATCH_USERS gets 30 percent of the 60 percent for a total overall percent of 18.

The steps for creating a resource plan, its directives and its groups is shown in Figure 42.

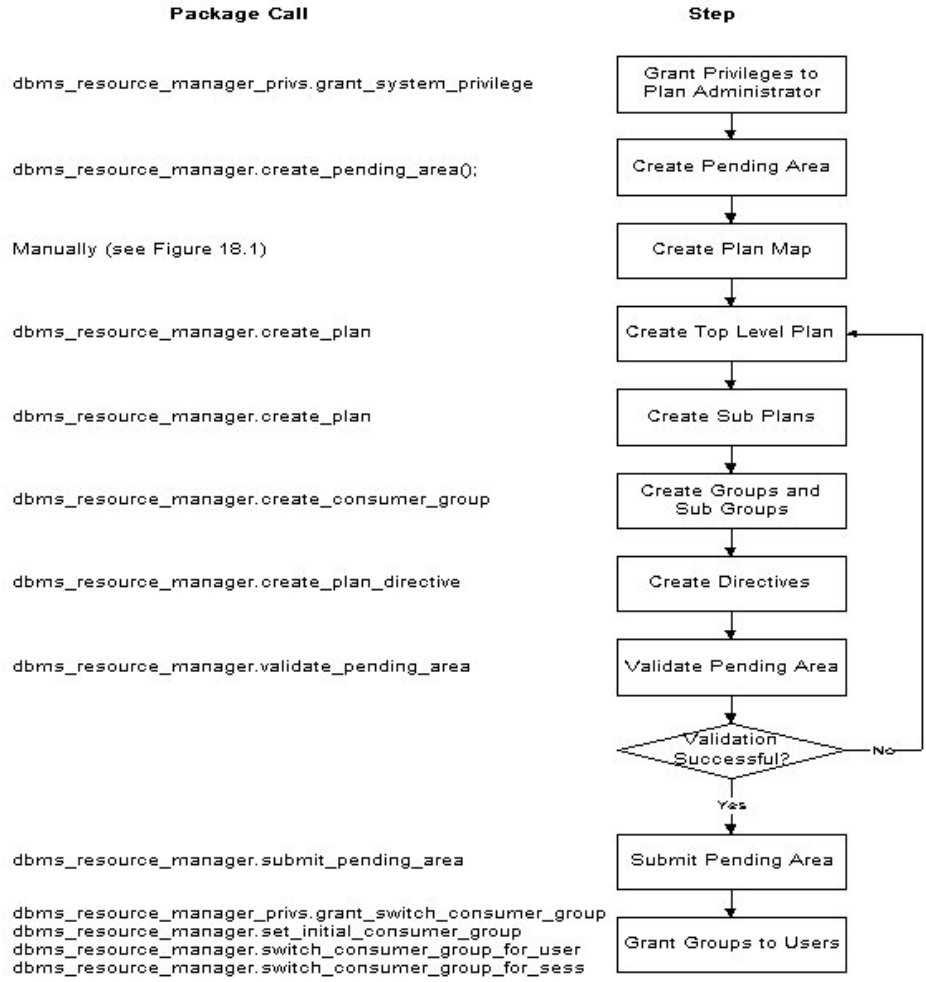


Figure 42 Steps to Create a Resource Plan

One thing to notice about Figure 42 is that the last step shows several possible packages which can be run to assign or change the assignment of resource groups. The first package listed, DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SWITCH_CONSUMER_GROUP must be run the first time a user is assigned to a resource group or you won't be able to assign the user to the group. After the user has been given the SWITCH_CONSUMER_GROUP system privilege you don't have to re-run the package. Figure 3 shows the code to create the resource plan in Figure 41. Figure 44 shows the results from running the source in figure 43.

```
set echo on
spool test_resource_plan.doc
-- Grant system privilege to plan administrator
--
execute
dbms_resource_manager_privs.grant_system_privilege('SYSTEM','ADMINISTER_RE
SOURCE_MANAGER',TRUE);
--
--connect to plan administrator
--
CONNECT system/system_test@ortest1.world
--
-- Create Plan Pending Area
--
EXECUTE dbms_resource_manager.create_pending_area();
--
-- Create plan
--
execute dbms_resource_manager.create_plan('MASTER','Example Resource
Plan','EMPHASIS');
execute dbms_resource_manager.create_plan('USERS','Example Resource Sub
Plan','EMPHASIS');
execute dbms_resource_manager.create_plan('REPORTS','Example Resource Sub
Plan','EMPHASIS');
--
--Create tiers of groups in plan
--
EXECUTE dbms_resource_manager.create_consumer_group('ONLINE_USERS','3rd
level group','ROUND-ROBIN');
EXECUTE dbms_resource_manager.create_consumer_group('BATCH_USERS','3rd
level group','ROUND-ROBIN');
EXECUTE dbms_resource_manager.create_consumer_group('ONLINE_REPORTS','2rd
level group','ROUND-ROBIN');
EXECUTE dbms_resource_manager.create_consumer_group('BATCH_REPORTS','2rd
level group','ROUND-ROBIN');
--
-- Create plan directives
--
EXECUTE dbms_resource_manager.create_plan_directive('MASTER', 'USERS',
0,60,0,0,0,0,0,0,NULL);
EXECUTE dbms_resource_manager.create_plan_directive('MASTER', 'REPORTS',
0,20,0,0,0,0,0,0,NULL);
EXECUTE
dbms_resource_manager.create_plan_directive('MASTER','OTHER_GROUPS',
0,20,0,0,0,0,0,0,NULL);
EXECUTE dbms_resource_manager.create_plan_directive('USERS',
'ONLINE_USERS',  0,0,70,0,0,0,0,0,NULL);
EXECUTE dbms_resource_manager.create_plan_directive('USERS',
'BATCH_USERS',   0,0,30,0,0,0,0,0,NULL);
EXECUTE
dbms_resource_manager.create_plan_directive('REPORTS','ONLINE_REPORTS',0,0
,70,0,0,0,0,0,NULL);
EXECUTE
dbms_resource_manager.create_plan_directive('REPORTS','BATCH_REPORTS',
0,0,30,0,0,0,0,0,NULL);
--
-- Verify Plan
--
```

```
EXECUTE dbms_resource_manager.validate_pending_area;
--
-- Submit Plan
--
EXECUTE dbms_resource_manager.submit_pending_area;
spool off
set echo off
```

Figure 43 Script to create example resource plan

Notice how the script in figure 3 follows the chart in Figure 2. These are the proper steps to create a resource plan. Figure 4 shows the results from running the script in Figure 3.

```
SQL> -- Grant system privilege to plan administrator
SQL> --
SQL> execute
dbms_resource_manager_privs.grant_system_privilege('SYSTEM','ADMINISTER_RE
SOURCE_MANAGER',TRUE);

PL/SQL procedure successfully completed.

SQL> --
SQL> --connect to plan administrator
SQL> --
SQL> CONNECT system/system_test@ortest1.world
Connected.
SQL> --
SQL> -- Create Plan Pending Area
SQL> --
SQL> EXECUTE dbms_resource_manager.create_pending_area();

PL/SQL procedure successfully completed.

SQL> --
SQL> -- Create plan
SQL> --
SQL> execute dbms_resource_manager.create_plan('MASTER','Example Resource
Plan','EMPHASIS');

PL/SQL procedure successfully completed.

SQL> execute dbms_resource_manager.create_plan('USERS','Example Resource
Sub Plan','EMPHASIS');

PL/SQL procedure successfully completed.

SQL> execute dbms_resource_manager.create_plan('REPORTS','Example Resource
Sub Plan','EMPHASIS');

PL/SQL procedure successfully completed.

SQL> --
SQL> --Create tiers of groups in plan
SQL> --
```

```
SQL> EXECUTE
dbms_resource_manager.create_consumer_group('ONLINE_USERS','3rd level
group','ROUND-ROBIN');

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_consumer_group('BATCH_USERS','3rd level
group','ROUND-ROBIN');

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_consumer_group('ONLINE_REPORTS','2rd level
group','ROUND-ROBIN');

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_consumer_group('BATCH_REPORTS','2rd level
group','ROUND-ROBIN');

PL/SQL procedure successfully completed.

SQL> --
SQL> -- Create plan directives
SQL> --
SQL> EXECUTE dbms_resource_manager.create_plan_directive('MASTER',
'USERS',           0,60,0,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE dbms_resource_manager.create_plan_directive('MASTER',
'REPORTS',         0,20,0,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_plan_directive('MASTER','OTHER_GROUPS',
0,20,0,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE dbms_resource_manager.create_plan_directive('USERS',
'ONLINE_USERS',  0,0,70,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE dbms_resource_manager.create_plan_directive('USERS',
'BATCH_USERS',   0,0,30,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_plan_directive('REPORTS','ONLINE_REPORTS',0,0
,70,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.
```

```
SQL> EXECUTE
dbms_resource_manager.create_plan_directive('REPORTS','BATCH_REPORTS',
0,0,30,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> --
SQL> -- Verify Plan
SQL> --
SQL> EXECUTE dbms_resource_manager.validate_pending_area;

PL/SQL procedure successfully completed.

SQL> --
SQL> -- Submit Plan
SQL> --
SQL> EXECUTE dbms_resource_manager.submit_pending_area;

PL/SQL procedure successfully completed.

SQL> spool off
```

Figure 44 Example run of script to create example resource plan

The other operations allowed against the components of the resource plan are alter and drop. Let's look at a quick drop example in Figure 45.

```
EXECUTE dbms_resource_manager.delete_plan('MASTER');
EXECUTE dbms_resource_manager.delete_plan('USERS');
EXECUTE dbms_resource_manager.delete_plan('REPORTS');
--
--delete tiers of groups in plan
--
EXECUTE dbms_resource_manager.delete_consumer_group('ONLINE_USERS');
EXECUTE dbms_resource_manager.delete_consumer_group('BATCH_USERS');
EXECUTE dbms_resource_manager.delete_consumer_group('ONLINE_REPORTS');
EXECUTE dbms_resource_manager.delete_consumer_group('BATCH_REPORTS');
```

Figure 45 Example Drop Procedure

Notice how you must drop all parts of the plan, this is because Oracle allows Orphan groups and plans to exist. As you can tell from looking at the scripts the DBMS_RESOURCE_MANAGER and DBMS_RESOURCE_MANAGER_PRIVS packages are critical to implementing Oracle resource groups. Let's examine these packages.

DBMS_RESOURCE_MANAGER Package

The DBMS_RESOURCE_MANAGER package is used to administer the new resource plan and consumer group options in Oracle8i. The package contains

several procedures that are used to create, modify, drop and grant access to resource plans, groups, directives and pending areas. The invoker must have the ADMINISTER_RESOURCE_MANAGER system privilege to execute these procedures. The procedures to grant and revoke this privilege are in the package DBMS_RESOURCE_MANAGER_PRIVS. The procedures in DBMS_RESOURCE_MANAGER are listed in table 5.

Procedure	Purpose
CREATE_PLAN	Creates entries that define resource plans.
UPDATE_PLAN	Updates entries that define resource plans.
DELETE_PLAN	Deletes the specified plan as well as all the plan directives it refers to.
DELETE_PLAN_CASCADE	Deletes the specified plan as well as all its descendants (plan directives, subplans, consumer groups).
CREATE_CONSUMER_GROUP	Creates entries that define resource consumer groups.
UPDATE_CONSUMER_GROUP	Updates entries that define resource consumer groups.
DELETE_CONSUMER_GROUP	Deletes entries that define resource consumer groups.
CREATE_PLAN_DIRECTIVE	Creates resource plan directives.
UPDATE_PLAN_DIRECTIVE	Updates resource plan directives.
DELETE_PLAN_DIRECTIVE	Deletes resource plan directives.
CREATE_PENDING_AREA	Creates a work area for changes to resource manager objects.
VALIDATE_PENDING_AREA	Validates pending changes for the resource manager.
CLEAR_PENDING_AREA	Clears the work area for the resource manager.
SUBMIT_PENDING_AREA	Submits pending changes for the resource manager.
SET_INITIAL_CONSUMER_ GROUP	Assigns the initial resource consumer group for a user.
SWITCH_CONSUMER_ GROUP_FOR_SESS	Changes the resource consumer group of a specific session.
SWITCH_CONSUMER_ GROUP_FOR_USER	Changes the resource consumer group for all sessions with a given user name.

Table 5 DBMS_RESOURCE_MANAGER_PACKAGES

DBMS_RESOURCE_MANGER Procedure Syntax

The calling syntax for all of the DBMS_RESOURCE_MANAGER packages follow.

Syntax for the CREATE_PLAN Procedure:

```
DBMS_RESOURCE_MANAGER.CREATE_PLAN (
   plan                        IN VARCHAR2,
   comment                     IN VARCHAR2,
   cpu_mth                     IN VARCHAR2 DEFAULT 'EMPHASIS',
   max_active_sess_target_mth IN VARCHAR2 DEFAULT
                                  'MAX_ACTIVE_SESS_ABSOLUTE',
   parallel_degree_limit_mth  IN VARCHAR2 DEFAULT
                             'PARALLEL_DEGREE_LIMIT_ABSOLUTE');
```

Where:

- Plan - the plan name
- Comment - any text comment you want associated with the plan name
- Cpu_mth - one of EMPHASIS or ROUND-ROBIN
- max_active_sess_target_mth - allocation method for max. active sessions
- parallel_degree_limit_mth - allocation method for degree of parallelism

Syntax for the UPDATE_PLAN Procedure:

```
DBMS_RESOURCE_MANAGER.UPDATE_PLAN (
   plan                           IN VARCHAR2,
   new_comment                    IN VARCHAR2 DEFAULT NULL,
   new_cpu_mth                    IN VARCHAR2 DEFAULT NULL,
   new_max_active_sess_target_mth IN VARCHAR2 DEFAULT NULL,
   new_parallel_degree_limit_mth  IN VARCHAR2 DEFAULT NULL);
```

Where:

- plan - name of resource plan
- new_comment - new user's comment
- new_cpu_mth - name of new allocation method for CPU resources
- new_max_active_sess_target_mth - name of new method for max. active sessions
- new_parallel_degree_limit_mth - name of new method for degree of parallelism

Syntax for the DELETE_PLAN Procedure:

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN (
   plan IN VARCHAR2);
```

Where:

- Plan - Name of resource plan to delete.

Syntax for the DELETE_PLAN Procedure:

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN_CASCADE (
   plan IN VARCHAR2);
```

Where:

- Plan - Name of plan.

Syntax for the CREATE_RESOURCE_GROUP Procedure:

```
DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP (
   consumer_group IN VARCHAR2,
   comment        IN VARCHAR2,
   cpu_mth        IN VARCHAR2 DEFAULT 'ROUND-ROBIN');
```

Where:

- consumer_group - Name of consumer group.
- Comment - User's comment.
- cpu_mth - Name of CPU resource allocation method.

Syntax for the UPDATE_RESOURCE_GROUP Procedure:

```
DBMS_RESOURCE_MANAGER.UPDATE_CONSUMER_GROUP (
   consumer_group IN VARCHAR2,
   new_comment    IN VARCHAR2 DEFAULT NULL,
   new_cpu_mth    IN VARCHAR2 DEFAULT NULL);
```

Where:

- plan - name of resource plan
- new_comment - new user's comment
- new_cpu_mth - name of new allocation method for CPU resources
- new_max_active_sess_target_mth - name of new method for max. active sessions

- new_parallel_degree_limit_mth - name of new method for degree of parallelism

Syntax for the DELTE_RESOURCE_GROUP Procedure:

```
DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP (
   consumer_group IN VARCHAR2);
```

Where:

- plan - name of resource plan.

Syntax for the CREATE_PLAN_DIRECTIVE Procedure:

```
DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE (
   plan                      IN VARCHAR2,
   group_or_subplan          IN VARCHAR2,
   comment                   IN VARCHAR2,
   cpu_p1                    IN NUMBER   DEFAULT NULL,
   cpu_p2                    IN NUMBER   DEFAULT NULL,
   cpu_p3                    IN NUMBER   DEFAULT NULL,
   cpu_p4                    IN NUMBER   DEFAULT NULL,
   cpu_p5                    IN NUMBER   DEFAULT NULL,
   cpu_p6                    IN NUMBER   DEFAULT NULL,
   cpu_p7                    IN NUMBER   DEFAULT NULL,
   cpu_p8                    IN NUMBER   DEFAULT NULL,
   max_active_sess_target_p1 IN NUMBER   DEFAULT NULL,
   parallel_degree_limit_p1  IN NUMBER   DEFAULT NULL);
```

Where:

- plan - name of resource plan
- group_or_subplan - name of consumer group or subplan
- comment - comment for the plan directive
- cpu_p1 - first parameter for the CPU resource allocation method
- cpu_p2 - second parameter for the CPU resource allocation method
- cpu_p3 - third parameter for the CPU resource allocation method
- cpu_p4 - fourth parameter for the CPU resource allocation method
- cpu_p5 - fifth parameter for the CPU resource allocation method
- cpu_p6 - sixth parameter for the CPU resource allocation method
- cpu_p7 - seventh parameter for the CPU resource allocation method
- cpu_p8 - eighth parameter for the CPU resource allocation method
- max_active_sess_target_p1 - first parameter for the max. active sessions allocation method

- (RESERVED FOR FUTURE USE)
- parallel_degree_limit_p1 - first parameter for the degree of parallelism allocation method

Syntax for the UPDATE_PLAN_DIRECTIVE Procedure:

```
DBMS_RESOURCE_MANAGER.UPDATE_PLAN_DIRECTIVE (
   plan                          IN VARCHAR2,
   group_or_subplan              IN VARCHAR2,
   new_comment                   IN VARCHAR2 DEFAULT NULL,
   new_cpu_p1                    IN NUMBER   DEFAULT NULL,
   new_cpu_p2                    IN NUMBER   DEFAULT NULL,
   new_cpu_p3                    IN NUMBER   DEFAULT NULL,
   new_cpu_p4                    IN NUMBER   DEFAULT NULL,
   new_cpu_p5                    IN NUMBER   DEFAULT NULL,
   new_cpu_p6                    IN NUMBER   DEFAULT NULL,
   new_cpu_p7                    IN NUMBER   DEFAULT NULL,
   new_cpu_p8                    IN NUMBER   DEFAULT NULL,
   new_max_active_sess_target_p1 IN NUMBER   DEFAULT NULL,
   new_parallel_degree_limit_p1  IN NUMBER   DEFAULT NULL);
```

Where:

- plan - name of resource plan
- group_or_subplan - name of group or subplan
- new_comment - comment for the plan directive
- new_cpu_p1 - first parameter for the CPU allocation method
- new_cpu_p2 - parameter for the CPU allocation method
- new_cpu_p3- parameter for the CPU allocation method
- new_cpu_p4 - parameter for the CPU allocation method
- new_cpu_p5 - parameter for the CPU allocation method
- new_cpu_p6 - parameter for the CPU allocation method
- new_cpu_p7 - parameter for the CPU allocation method
- new_cpu_p8 - parameter for the CPU allocation method
- new_max_active_sess_target_p1 - first parameter for the max. active sessions allocation method
- (RESERVED FOR FUTURE USE)
- new_parallel_degree_limit_p1 - first parameter for the degree of parallelism allocation method

Syntax for the DELETE_PLAN_DIRECTIVE Procedure:

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN_DIRECTIVE (
   plan             IN VARCHAR2,
   group_or_subplan IN VARCHAR2);
```

Where:

- plan - name of resource plan
- group_or_subplan - name of group or subplan.

Syntax for CREATE_PENDING_AREA Procedure:

This procedure lets you make changes to resource manager objects.

All changes to the plan schema must be done within a pending area. The pending area can be thought of as a "scratch" area for plan schema changes. The administrator creates this pending area, makes changes as necessary, possibly validates these changes, and only when the submit is completed do these changes become active.

You may, at any time while the pending area is active, view the current plan schema with your changes by selecting from the appropriate user views.

At any time, you may clear the pending area if you want to stop the current changes. You may also call the VALIDATE procedure to confirm whether the changes you has made are valid. You do not have to do your changes in a given order to maintain a consistent group of entries. These checks are also implicitly done when the pending area is submitted.

Note: Oracle allows "orphan" consumer groups (i.e., consumer groups that have no plan directives that refer to them). This is in anticipation that an administrator may want to create a consumer group that is not currently being used, but will be used in the future. The procedure has no arguments.

```
DBMS_RESOURCE_MANAGER.CREATE_PENDING_AREA;
```

Syntax of the VALIDATE_PENDING_AREA Procedure:

The VALIDATE_PENDING_AREA procedure is used to validate the contents of a pending area before they are submitted. The procedure has no arguments.

```
DBMS_RESOURCE_MANAGER.VALIDATE_PENDING_AREA;
```

Usage Notes For the Validate and Submit Procedures:

The following rules must be adhered to, and they are checked whenever the validate or submit procedures are executed:

1. No plan schema may contain any loops.
2. All plans and consumer groups referred to by plan directives must exist.
3. All plans must have plan directives that refer to either plans or consumer groups.
4. All percentages in any given level must not add up to greater than 100 for the emphasis resource allocation method.
5. No plan may be deleted that is currently being used as a top plan by an active instance.
6. For Oracle8i, the plan directive parameter, parallel_degree_limit_p1, may only appear in plan directives that refer to consumer groups (i.e., not at subplans).
7. There cannot be more than 32 plan directives coming from any given plan (i.e., no plan can have more than 32 children).
8. There cannot be more than 32 consumer groups in any active plan schema.
9. Plans and consumer groups use the same namespace; therefore, no plan can have the same name as any consumer group.
10. There must be a plan directive for OTHER_GROUPS somewhere in any active plan schema.This ensures that a session not covered by the currently active plan is allocated resources as specified by the OTHER_GROUPS directive.

If any of the above rules are broken when checked by the VALIDATE or SUBMIT procedures, then an informative error message is returned. You may then make changes to fix the problem(s) and reissue the validate or submit procedures.

Syntax of the CLEAR_PENDING_AREA Procedure:

The CLEAR_PENDING_AREA procedure clears the pending area without submitting it, all changes or entries are lost. The procedure has no arguments.

```
DBMS_RESOURCE_MANAGER.CLEAR_PENDING_AREA;
```

Syntax of the SUBMIT_PENDING_AREA Procedure:

The SUBMIT_PENDING_AREA procedure submits the contents of the pending area. First the contents are validated and then they are stored as valid in the database. The procedure has no arguments.

```
DBMS_RESOURCE_MANAGER.SUBMIT_PENDING_AREA;
```

Syntax of the SET_INITIAL_CONSUMER_GROUP Procedure:

The SET_INITIAL_CONSUMER_GROUP procedure sets the initial consumer group to which a user will belong. The user must have been granted SWITCH_RESOURCE_GROUP permission before you attempt to run this procedure.

```
DBMS_RESOURCE_MANAGER.SET_INITIAL_CONSUMER_GROUP (
   user           IN VARCHAR2,
   consumer_group IN VARCHAR2);
```

Where:

- User – The user that is to have the resource group set.
- Consumer_group – The resource (or consumer) group to grant to the user.

Syntax of the SWITCH_CONSUMER_GROUP_FOR_SESS Procedure:

The SWITCH_RESOURCE_GROUP_FOR_SESS procedure allows an administrator to switch a user's consumer group for the duration of the current session.

```
DBMS_RESOURCE_MANAGER.SWITCH_CONSUMER_GROUP_FOR_SESS(
  SESSION_ID      IN NUMBER,
  SESSION_SERIAL  IN NUMBER,
  CONSUMER_GROUP  IN VARCHAR2);
```

Where:

- session_id - SID column from the view V$SESSION
- session_serial - SERIAL# column from the view V$SESSION
- consumer_group - name of the consumer group of which to switch.

Syntax of the SWITCH_CONSUMER_GROUP_FOR_USER Procedure:

The SWITCH_CONSUMER_GROUP_FOR_USER switches a user's default consumer group to a new group. This is a permanent change.

```
DBMS_RESOURCE_MANAGER.SWITCH_CONSUMER_GROUP_FOR_USER (
   user           IN VARCHAR2,
   consumer_group IN VARCHAR2);
```

Where:

- user - name of the user
- consumer_group - name of the consumer group to switch to

DBMS_RESOURCE_MANAGER_PRIVS Package

The DBMS_RESOURCE_MANAGER package has a companion package that grants privileges in the realm of the resource consumer option. The companion package is DBMS_RESOURCE_MANAGER_PRIVS. The procedures inside DBMS_RESOURCE_MANAGER_PRIVS are documented in table 6.

Procedure	Purpose
GRANT_SYSTEM_PRIVILEGE	Performs a grant of a system privilege.
REVOKE_SYSTEM_PRIVILEGE	Performs a revoke of a system privilege.
GRANT_SWITCH_CONSUMER_GROUP	Grants the privilege to switch to resource consumer groups.
REVOKE_SWITCH_CONSUMER_GROUP	Revokes the privilege to switch to resource consumer groups.

Table 6 DBMS_RESOURCE_MANAGER_PRIVS Procedures

DBMS_RESOURCE_MANGER_PRIVS Procedure Syntax

The calling syntax for all of the DBMS_RESOURCE_MANAGER_PRIVS packages follows.

Syntax for the GRANT_SYSTEM_PRIVILEGE Procedure:

The GRANT_SYSTEM_PRIVILEGE procedure grants ADMINISTER_RESOURCE_MANAGER privilege to a user. Currently there is only one resource group system grant.

```
DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SYSTEM_PRIVILEGE (
   grantee_name   IN VARCHAR2,
   privilege_name IN VARCHAR2 DEFAULT 'ADMINISTER_RESOURCE_MANAGER',
   admin_option   IN BOOLEAN);
```

Where:

- grantee_name - Name of the user or role to whom privilege is to be granted.
- privilege_name - Name of the privilege to be granted.
- admin_option - TRUE if the grant is with admin_option, FALSE otherwise.

Syntax for the REVOKE_SYSTEM_PRIVILGE Procedure:

The REVOKE_SYSTEM_PRIVILEGE procedure revokes the ADMINISTER_RESOURCE_MANAGER privilege from a user.

```
DBMS_RESOURCE_MANAGER_PRIVS.REVOKE_SYSTEM_PRIVILEGE (
   revokee_name   IN VARCHAR2,
   privilege_name IN VARCHAR2 DEFAULT 'ADMINISTER_RESOURCE_MANAGER');
```

Where:

- revokee_name - Name of the user or role from whom privilege is to be revoked.
- privilege_name - Name of the privilege to be revoked.

Syntax of the GRANT_SWITCH_CONSUMER_GROUP Procedure:

The GRANT_SWITCH_CONSUMER_GROUP procedure grants a user the ability to switch resource groups. This privilege must be granted to a user before their initial resource group can be granted.

```
DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SWITCH_CONSUMER_GROUP (
   grantee_name   IN VARCHAR2,
   consumer_group IN VARCHAR2,
   grant_option   IN BOOLEAN);
```

Where:

- grantee_name - Name of the user or role to whom privilege is to be granted.
- consumer_group - Name of consumer group.

- grant_option - TRUE if grantee should be allowed to grant access, FALSE otherwise.

Usage Notes

1. If you grant permission to switch to a particular consumer group to a user, then that user can immediately switch their current consumer group to the new consumer group.
2. If you grant permission to switch to a particular consumer group to a role, then any users who have been granted that role and have enabled that role can immediately switch their current consumer group to the new consumer group.
3. If you grant permission to switch to a particular consumer group to PUBLIC, then any user can switch to that consumer group.
4. If the grant_option parameter is TRUE, then users granted switch privilege for the consumer group may also grant switch privileges for that consumer group to others.
5. In order to set the initial consumer group of a user, you must grant the switch privilege for that group to the user.

Syntax of the REVOKE_SWITCH_CONSUMER_GROUP Procedure:

The REVOKE_SWITCH_CONSUMER_GROUP procedure revokes the ability of a user to switch their resource group.

```
DBMS_RESOURCE_MANAGER_PRIVS.REVOKE_SWITCH_CONSUMER_GROUP (
   revokee_name   IN VARCHAR2,
   consumer_group IN VARCHAR2);
```

Where:

- revokee_name - Name of user/role from which to revoke access.
- consumer_group - Name of consumer group.

Usage Notes

1. If you revoke a user's switch privilege for a particular consumer group, then any subsequent attempts by that user to switch to that consumer group will fail.
2. If you revoke the initial consumer group from a user, then that user will automatically be part of the DEFAULT_CONSUMER_GROUP (OTHERS) consumer group when logging in.

3. If you revoke the switch privilege for a consumer group from a role, then any users who only had switch privilege for the consumer group via that role will not be subsequently able to switch to that consumer group.
4. If you revoke the switch privilege for a consumer group from PUBLIC, then any users who could previously only use the consumer group via PUBLIC will not be subsequently able to switch to that consumer group.

Section Summary

By carefully planning your resource allocation into plans and resource groups a multi-tier resource allocation plan can be quickly developed. By allocating CPU resources you can be sure that processing power is concentrated where it needs to be such that the CEO isn't waiting on a sub-clerk's process to finish before they get their results.

This section has shown how to use the various DBMS packages to configure and maintain a resource plan with its associated consumer groups.

Presentation Summary

In this presentation we have looked at non-code related Oracle tuning for application where alteration of source code is not allowed. We have looked at physical and internals tuning, indexing options, table and index tuning as well as methods for placing “stealth hints” into code.

This paper contains excerpts from the book: "Oracle Administration and Management", Michael R. Ault, John Wiley and Sons publishing with permission.