MONITORING AND TUNING ORACLE - CHAPTER 22 PART 2

This material is extracted from the second half of Chapter 22 of *Configuring and Tuning Databases on the Solaris Platform*, by Allan N. Packer, (c) 2002, Sun Microsystems Press. The first half of the chapter appeared in the July edition of Sun BluePrints Online. Chapter 21, *Drill-Down Monitoring of Database Servers,* was presented in the June 2002 edition of Sun BluePrints Online.

Monitoring Oracle with utlbstat/utlestat

In the first part of this chapter we considered ways of managing Oracle behavior, including changing tunable parameters, monitoring error messages, generating explain plans, and calculating the buffer cache hit rate. We then discussed monitoring Oracle with the `utlbstat.sql` and `utlestat.sql` scripts. We continue this theme with buffer busy wait statistics.

Buffer Busy Wait Statistics

The following statistics are useful if buffer busy wait events suggest high contention on buffers. The issues are discussed earlier in this chapter in "Systemwide Wait Events" on page 19.

```
SQL> Rem Buffer busy wait statistics.  If the value for 'buffer busy wait' in
SQL> Rem the wait event statistics is high, then this table will identify
SQL> Rem which class of blocks is having high contention.  If there are high
SQL> Rem 'undo header' waits, then add more rollback segments.  If there are
SQL> Rem high 'segment header' waits, then adding freelists might help. Check
SQL> Rem v$session_wait to get the addresses of the actual blocks having
SQL> Rem contention.
SQL> select * from stats$waitstat
  2    where count != 0
  3    order by count desc;

CLASS                    COUNT              TIME
data block                 288                 0
undo header                 49                 0
undo block                  38                 0
free list                    4                 0
```

Rollback Segments

The `v$rollstat` view presents detailed information on rollback segment behavior. The purpose of rollback segments is described in “Segments” on page 112 of *Configuring and Tuning Databases on the Solaris Platform*.

```
SQL> set lines 159;
SQL> set numwidth 19;
SQL> Rem Waits_for_trans_tbl high implies you should add rollback segments.
SQL> select * from stats$roll;


UNDO_SEG TRN_TBL_GETS TRN_TBL_WAITS UNDO_BYT_WR SEG_SIZE_BYT XACTS SHRINKS  WRAPS
-------- ------------ ------------- ----------- ------------ ----- -------  -----
       0            6             0           0       458752     0       0      0
       1        36385             0    24940942      2762752     0       0    293
       2        35993             0    24685736      2353152     1       0    289
       3        35937             0    24726410      6858752     1       0    290
<< rows deleted >>
```

As transactions proceed, rollback segments gradually grow in size. Oracle reclaims this space periodically by automatically eliminating extents to return a rollback segment to the optimal size (set by the `OPTIMAL` parameter in the `STORAGE` clause of the rollback segment). A substantial number of shrinks can indicate that the optimal rollback segment size is too small. Be aware, though, that setting the `OPTIMAL` parameter too large wastes space.

If waits are more than 5% of gets, increase the number of rollback segments.

Note that I have abbreviated some of the column headings above for formatting purposes.

Modified init.ora Parameters

The following section of the `report.txt` file displays current `init.ora` parameters that have been modified from the defaults.

```
SQL> set lines 79;
SQL>
SQL> column name  format a39 trunc;
SQL> column value format a39 trunc;
SQL> Rem The init.ora parameters currently in effect:
SQL> select name, value from v$parameter where isdefault = ‘FALSE’
  2    order by name;

NAME                                    VALUE
db_block_buffers                         605600
db_block_size                            2048
db_file_multiblock_read_count             4
db_files                                 350
db_name                                    accts
db_writer_processes                       5
dml_locks                                2500
log_buffer                               8097792
log_checkpoint_interval                   999999999
shared_pool_size                         33554432
sort_area_size                            1000000
transactions                             1000
<< rows deleted >>
```

Dictionary Cache Statistics

The dictionary cache stores data dictionary information for all objects in the database. Consequently, it tends to be heavily accessed. Details of dictionary cache statistics can also be queried with the `v$rowcache` view.

```
SQL> column name format a15 trunc;
SQL> column scan_reqs heading 'SCAN_REQ';
SQL> column scan_miss heading 'SCAN_MIS';
SQL> column cur_usage heading 'CUR_USAG';
SQL> set numwidth 8;
SQL> Rem get_miss and scan_miss should be very low compared to requests.
SQL> Rem cur_usage is the number of entries in the cache being used.
SQL> select * from stats$dc
  2   where get_reqs != 0 or scan_reqs != 0 or mod_reqs != 0;

NAME            GET_REQS GET_MISS SCAN_REQ SCAN_MIS MOD_REQS COUNT  CUR_USAG
--------------- -------- -------- -------- -------- -------- -----  --------
dc_tablespaces     39923        0        0        0        0    14        13
dc_free_extents       28        6        6        0       18    42         4
dc_segments           29        0        0        0        6   141       123
dc_rollback_seg      600        0        0        0        0    66        61
dc_used_extents        6        6        0        0        6    88        55
dc_tablespace_q       23        0        0        0       23    23         8
dc_users              56        0        0        0        0    14         6
dc_user_grants        52        0        0        0        0    14         4
dc_objects            18        0        0        0        0   313       301
dc_usernames           3        0        0        0        0    21         4
dc_object_ids         10        0        0        0        0   230       218
dc_profiles            1        0        0        0        0     3         1

12 rows selected.
```

Once the database has been running for a while, look for a high proportion of misses compared to gets. If misses are significant, increase the size of the shared pool (the `shared_pool_size` parameter in `init.ora`).

Tablespace and Database File I/O Activity

The following section of `report.txt` shows I/O activity by tablespace and by database file, including the number of physical reads and writes and the time taken to complete each type of I/O (in hundredths of seconds). If times are shown as `0`, set the `timed_statistics` parameter in `init.ora` to `TRUE`.

Mapping database files to physical disks can be complicated if a volume manager is used to stripe the database files. Nonetheless, information about the I/O activity for each file can be useful in data layout planning (discussed in detail in Chapter 17). For more information about monitoring disk utilization, refer to "STEP 2. Monitoring Disks" on page 289 of *Configuring and*

Tuning Databases on the Solaris Platform.

```
SQL> set lines 157;
SQL> column table_space format a80 trunc;
SQL> set numwidth 10;
SQL> Rem Sum IO operations over tablespaces.
SQL> select
  2    table_space||'      `
  3    table_space,
  4    sum(phys_reads) reads,  sum(phys_blks_rd) blks_read,
  5    sum(phys_rd_time) read_time,  sum(phys_writes) writes,
  6    sum(phys_blks_wr) blks_wrt,  sum(phys_wrt_tim) write_time,
  7    sum(megabytes_size) megabytes
  8   from stats$files
  9   group by table_space
 10   order by table_space;

TABLE_SPACE  READS BLKS_READ READ_TIME WRITES BLKS_WRT WRITE_TIME MEGABYTES
------------ ----- --------- --------- ------ -------- ---------- ---------
SYSTEM         410      1528      2321  28333    28333          0       262
TABLESPACE1    415       415      1610   1331     1331          0        52
TABLESPACE10 22847     22847     62941  49582    49582          0       189
TABLESPACE12 15288     15287     51655      2        2          0       157
TABLESPACE15     2         2         0   5102     5102          0        31
TABLESPACE16     2         2         0      2        2          0       210
TABLESPACE5 291653    291643    916007 225756   225755          0      2310
TABLESPACE6  77249     77249    201654 116744   116746          0      2520
TABLESPACE7 252514    252506    961914 491426   491428          0      3528
<< rows deleted >>

SQL> set lines 196;
SQL> column table_space format a48 trunc;
SQL> column file_nameformat a48 trunc;
SQL> set numwidth 10;
SQL> Rem I/O should be spread evenly across drives. A big difference
between
SQL> Rem phys_reads and phys_blks_rd implies table scans are going on.
SQL> select table_space, file_name,
  2     phys_reads reads, phys_blks_rd blks_read, phys_rd_time read_time,
  3     phys_writes writes, phys_blks_wr blks_wrt, phys_wrt_tim write_time,
  4     megabytes_size megabytes,
  5     round(decode(phys_blks_rd,0,0,phys_rd_time/phys_blks_rd),2) avg_rt,
  6     round(decode(phys_reads,0,0,phys_blks_rd/phys_reads),2) "blocks/rd"
  7   from stats$files order by table_space, file_name;

 TABLE_SPACE    FILE_NAME               READS  BLKS_READ  READ_TIME  WRITES
BLKS_WRT  WRITE_TIME  MEGABYTES  AVG_RT  blocks/rd
-------------- ----------------------- -----  ---------  ---------  ------
--------  ----------  ---------  ------  ---------
SYSTEM          /home2/dbfiles/sys001     410       1528       2321   28333
   28333           0        262    1.52       3.73
TABLESPACE1     /home3/dbfiles/wdi001     415        415       1610    1331
    1331           0         52    3.88          1
TABLESPACE10   /home2/dbfiles/iord2001   22847     22847      62941   49582
   49582           0        189    2.75          1
TABLESPACE12   /home2/dbfiles/icust001   15288     15287      51655       2
       2           0        157    3.38          1
TABLESPACE15   /home2/dbfiles/roll001        2          2          0    5102
    5102           0         31       0          1
<< rows deleted >>
```

Date, Time, and Version Details

The final section of `report.txt` shows the start and end time of the report and the Oracle version.

```
SQL> set lines 79;
SQL>
SQL> column start_time format a25;
SQL> column end_time   format a25;
SQL> Rem The times that bstat and estat were run.
SQL> select to_char(start_time, 'dd-mon-yy hh24:mi:ss') start_time,
  2    to_char(end_time,'dd-mon-yy hh24:mi:ss') end_time
  3    from stats$dates;

START_TIME                END_TIME
------------------------- -------------------------
17-aug-01 10:01:30        17-aug-01 10:31:24

SQL> column banner format a75 trunc;
SQL> Rem Versions
SQL> select * from v$version;

BANNER
------------------------------------------------------------
Oracle9i Enterprise Edition Release 9.0.1.0.0 - Production
PL/SQL Release 9.0.1.0.0 - Production
CORE      9.0.1.0.0 - Production
TNS for Solaris: Version 9.0.1.0.0 - Production
NLSRTL Version 9.0.1.0.0 - Production

SQL> spool off;
```

Monitoring the Shared Pool

The Oracle shared memory area is called the System Global Area (SGA). After the buffer cache, the shared pool is typically the largest component of the SGA.

The shared pool includes memory for the library cache, which caches information about database objects such as stored procedures and views, the cursor cache, which caches SQL statements and, if the Shared Server (previously known as the MultiThreaded Server, or MTS) is in use, a cache for session-specific information such as the context area and the sort area.

If the shared pool is too small, performance can suffer. It can be increased with the `shared_pool_size` parameter in `init.ora`. To monitor the shared pool, run the following statements as `sysdba`:

```
select value from v$parameter where name = 'shared_pool_size';
select name, bytes from v$sgastat where name = 'free memory';
```

If free memory is more than 40% of the shared pool, you may be able to decrease the size of the shared pool. It is less easy to determine whether the shared pool is too small; a system with only 10% free memory in the shared pool might be running efficiently. Look for `RELOADS` in the library cache statistics to see how often objects have been aged out of the shared pool (see

"The Library Cache" on page 13 for details). Frequent reloads can indicate that the shared pool is too small.

Tuning Oracle

Once the system has been appropriately configured (see Chapters 13 through 17, and especially the data layout recommendations in Chapter 17 of *Configuring and Tuning Databases on the Solaris Platform*), it is appropriate to find out if Oracle is properly configured. Oracle tuning is carried out with `init.ora` parameters, of which there are many; we focus on the most important parameters.

Tuning init.ora

When Oracle is installed, the `init.ora` file assumes a "small model" with just 400 Kbytes set aside for database buffers. For many end-user sites, several `init.ora` parameters will need to be changed in order for Oracle to work effectively.

A number of general tunable parameters unrelated to performance need to be assigned values large enough to support the required number of users and transactions:

- **`processes`:** The number of concurrent processes supported.
- **`sessions`:** The number of concurrent sessions supported.
- **`dml_locks`:** The number of locks that can be set. Needs to be set high enough to avoid "DML Lock" error. Try eight times the number of transactions.
- **`db_files`:** The number of database files that can be open while the database is running.
- **`transactions`:** The number of concurrent transactions supported.

Setting Tunable Parameters for OLTP Workloads

The most important `init.ora` tunable parameters for OLTP workloads are the following:

- **`db_block_size`:** Database block size in bytes. This parameter must be set when the database is created. You cannot change the block size once it is set without exporting and reimporting the data. All database blocks use this size. Use 2 Kbytes (set `db_block_size` to 2048), 4 Kbytes (4096), or 8 Kbytes (8192) for OLTP workloads.

  As of Oracle9i, different block sizes can be used for different tablespaces. The `blocksize` clause of the `create tablespace` statement can be used to achieve this effect.

- **db_block_buffers:** The amount of memory in database blocks set aside for the database buffer cache. The block size is defined by the `db_block_size` parameter. For OLTP workloads, this parameter is probably the most important tunable for performance. If you change it, monitor the buffer cache hit rate before and after to see how much improvement you have gained. Refer to “Calculating the Buffer Cache Hit Rate” on page 11 for details.

- **buffer_pool_keep** and **buffer_pool_recycle:** Parameters that reset the recycle and keep pools, described in “System Global Area (SGA)” on page 108 of *Configuring and Tuning Databases on the Solaris Platform*. These pools can be especially useful for improving the performance of OLTP workloads. Set the `buffer_pool_keep` and `buffer_pool_recycle` parameters in `init.ora` to the number of blocks that should be reserved for the keep and recycle pools, respectively. The number of blocks in the main (default) buffer pool will consist of those assigned to `db_block_buffers` minus those assigned to `buffer_pool_keep` and `buffer_pool_recycle`.

  To assign a table to the keep pool, for example, add the `storage (buffer_pool keep)` clause to either a `create table` or an `alter table` statement. The following statement illustrates the required syntax:

  ```
  alter table customer storage (buffer_pool keep);
  ```

- **db_cache_size:** Replacement for `db_block_buffers` in Oracle9i, although the older parameter is still supported for backward compatibility. The recommended approach is now to use `db_cache_size`, which uses a unit of bytes rather than blocks, making it easier to understand. If you use `db_block_buffers`, you won't be able to dynamically resize the buffer cache (described in “Reconfiguring Oracle9i Dynamically” on page 40).

  As previously stated, Oracle9i also supports different block sizes for different tablespaces. Consequently, separate caches must be configured for these tablespaces. The following new parameters are used to configure caches for tablespaces with 2-, 4-, 8-, 16-, and 32-Kbyte block sizes, respectively:

  ```
  db_2k_cache_size
  db_4k_cache_size
  db_8k_cache_size
  db_16k_cache_size
  db_32k_cache_size
  ```

  Any new caches will be configured in addition to the cache that is sized according to the `db_cache_size` parameter. One parameter must be avoided, though: the cache size parameter corresponding to the current block size. For example, if the standard `db_block_size` is set to `4096`, the `db_4k_cache_size` parameter cannot be used. The appropriate

parameter for tablespaces using the standard block size is db_cache_size.

- **shared_pool_size:** The size in bytes of the shared pool in the SGA. The shared pool stores the library cache, the shared SQL area, and session-specific data (only when the Shared Server is being used). This parameter is also important for performance. Depending on the environment and the amount of memory available, set it to a minimum of 20 Mbytes. Larger sites will require much more memory for the shared pool.

- **sort_area_size:** The maximum size in bytes of user memory available for sorting. Sorting is required during index creation and as a result of the group by and order by clauses of the select statement. Increase this parameter if sorts (disk) in v$sysstat (or report.txt) is more than 5% of sorts (memory) and disk sorts are occurring frequently. Note that the statspack report calculates the In-memory sort %. The sort_area_size parameter determines the memory allocation per user, so a large setting can quickly consume memory. Use 64 Kbytes, or more if plenty of memory is available.

- **log_buffer:** The size in bytes of the redo log buffer in the SGA. Log data is cached here before being written to the redo logs. Try 1 Mbyte.

- **db_writer_processes (db_writers in Oracle7):** The number of DBWR processes available to flush dirty pages from the buffer cache. For small systems you can use the default of 1, although for large OLTP systems more than one will probably be necessary. Increase the number of DBWR processes if free buffer waits appears as one of the most frequent wait events or the dirty buffers inspected statistic is much greater than zero in the report.txt file. Note that you should increase db_writer_processes rather than the dbwr_io_slaves parameter.

  In Oracle7, multiple db_writers could be used only if async_write was set to FALSE.

- **rollback_segments:** The rollback segments available for transactions. Rollback segments need to be big enough to complete large transactions, and you need enough of them to support multiple concurrent transactions without undue contention. Increase the number of rollback segments if undo header waits in v$waitstat is high and increasing.

- **disk_async_io:** A boolean parameter designating whether or not asynchronous I/O should be used. Use the default of TRUE. In Oracle7, set async_read and async_write to TRUE. In combination with raw partitions, asynchronous I/O allows the use of Kernel Asynchronous I/O (KAIO). Note, however, the recommendations and limitations on the use of asynchronous I/O explained in "Using Oracle with File Systems" on page 36.

Setting Tunable Parameters for DSS Workloads

Following are the most important tunable parameters for DSS workloads:

- **db_block_size:** Database block size in bytes. Set when the database is created. All database blocks use this size. Use 16 or 32 Kbytes for DSS workloads.
- **db_block_buffers:** The amount of memory in database blocks (as defined by db_block_size) set aside for the database buffer cache. For DSS workloads, large buffer caches are less useful and this parameter is usually set much lower than for OLTP workloads. As described above in "Setting Tunable Parameters for OLTP Workloads" on page 32, a new parameter, db_cache_size, is provided as of Oracle9i.
- **shared_pool_size:** The size in bytes of the shared pool in the SGA. The shared pool stores the library cache and the shared SQL area. This parameter is also important for performance. Depending on the amount of memory available, set shared_pool_size to a minimum of 20 Mbytes.
- **sort_area_size:** The maximum size in bytes of user memory available for sorting. Increase if sorts (disk) in v$sysstat is more than 10% of sorts (memory). This amount of memory can be allocated per user, so large values can cause Oracle to quickly consume memory. Set to 1 Mbyte or more if you have the memory available.
- **sort_direct_writes:** A boolean parameter designating whether or not writes to temporary segments should bypass the buffer cache (as of Oracle 7.2). Set to TRUE for DSS workloads to improve performance. This parameter was made obsolete in Oracle8i; direct writes now occur automatically.
- **log_buffer:** The size in bytes of the redo log buffer in the SGA. Buffers log data before writing to the redo logs. This parameter can be important for performance, especially during database updates. Try 1 Mbyte.
- **db_file_multiblock_read_count:** The number of database blocks read at once when a tablescan is performed. Set it to 64 to get 1 Mbyte reads with a db_block_size of 16 Kbytes.
- **rollback_segments:** The rollback segments available for transactions. Rollback segments need to be big enough to complete large transactions, and you need enough of them to support multiple concurrent transactions without undue contention. Increase if undo header waits in v$waitstat is high and increasing.
- **hash_area_size:** The maximum size in bytes that will be used for hash joins (the default is twice sort_area_size). This parameter is per user and is private memory, so multiple users can easily consume

more than 4 Gbytes, even with 32-bit databases on 32-bit Solaris. A large hash area can make a big difference to hash join performance for large tables. If you have a lot of memory, make it larger than 2 Mbytes.

- **`parallel_max_servers`:** The maximum size of the query server pool. This parameter determines the degree of parallelism during table scans. Use an upper limit of four times the number of CPUs. Set `parallel_min_servers` to the same value.
- **`optimizer_percent_parallel`:** A weighting factor used by the optimizer to determine how much weight to give to query parallelism. For fastest, greedy use of resources, set to `100` (favors table scans). Lower settings favor index scans.
- **`query_rewrite_enabled`:** A boolean parameter designating whether or not the optimizer should rewrite queries to take advantage of previously created materialized views. Set to `TRUE` to enable materialized views (Oracle 8i onwards). The query rewrite option can also be set by an `alter system` or `alter session` SQL command.

Applying Other Tuning Tips

This section presents a number of miscellaneous tuning tips related to file systems, load and index performance, and relinking of the `oracle` binary to allow shared memory segments larger than 2 Gbytes in size.

Using Oracle with File Systems

The issues associated with using database files on file systems are discussed elsewhere in this book:

- "Unix File System Enhancements" on page 21 of *Configuring and Tuning Databases on the Solaris Platform* outlines the enhancements made in Solaris to improve UFS database performance.
- "Raw Devices vs. UFS" on page 242 of the book discusses the performance implications of UFS.
- "UFS Files and Paging" on page 284 of the book explores UFS files in the context of system monitoring.

The bottom line is that if you decide to place your data files on UFS, use Direct I/O as of the 1/01 release of Solaris 8, especially for redo log files.

Asynchronous I/O with File Systems

Although all recent versions of Oracle support asynchronous I/O on Solaris raw devices, not all support asynchronous I/O on Unix file systems. Asynchronous I/O and its benefits are described in "Kernel Asynchronous I/O" on page 21 of *Configuring and Tuning Databases on the Solaris Platform*.

The use of asynchronous I/O by Oracle is controlled by the disk_async_io parameter in init.ora (in Oracle7, the async_read and async_write parameters are used instead). The default of TRUE should be used for raw devices.

Versions of Oracle before Oracle8.1.5 also support asynchronous I/O for database files residing on file systems. The Oracle8.1.5 and 8.1.6 releases, however, automatically disable asynchronous I/O for database files on file systems, whatever the setting of disk_async_io.

For releases as of Oracle8.1.7 (including Oracle9i), asynchronous I/O is once again supported on file systems, and a hidden parameter, _filesystemio_options, has been introduced to control its behavior. This parameter accepts the following values:

- **async.** This is the default setting, and means that asynchronous I/O is enabled for database files on Unix file systems.
- **directIO.** Enable Direct I/O, but not asynchronous I/O, for database files on Unix file systems.
- **setall.** Enable both Direct I/O and asynchronous I/O for database files on Unix file systems. You can also achieve this effect with the default setting of async and by mounting file systems with the forcedirectio option.
- **none.** Do not enable either asynchronous I/O or Direct I/O for database files on Unix file systems.

Recommended Settings with File Systems

For database files on Unix file systems, use asynchronous I/O whenever it is supported, since asynchronous I/O allows the database writers to work more effectively by issuing multiple writes simultaneously. In addition, use Direct I/O as of the 1/01 release of Solaris 8.

For database files on raw devices, always use asynchronous I/O, the default setting.

To remain consistent with my earlier caution about the use of hidden parameters (see “Viewing and Changing Hidden Parameters” on page 5), it would seem reasonable to leave the default for _filesystemio_options as async and enable Direct I/O (as of Solaris 8 1/01) by mounting file systems with the forcedirectio option. This combination will result in the same effect as the setall option.

In fact, as of the second release of Oracle9i, the parameter becomes a normal parameter (filesystemio_options). Nonetheless, I still suggest using the parameter default and enabling Direct I/O.

The usefulness of Direct I/O is enhanced by the fact that it can be dynamically turned on and off for any mounted file system, as the following commands illustrate:

```
mount -o remount,forcedirectio /my/filesystem
mount -o remount,noforcedirectio /my/filesystem
```

Optimizing Oracle Load Performance

DSS workloads in particular tend to require periodic loads, often of large volumes of data.

To reduce load times, use the sequence outlined in the following steps.

Load the Database Tables

You save significant time by avoiding access to the online redo log and archive logs during data loading. You can run Oracle in NOARCHIVELOG mode. Alternatively, you can disable redo generation with the alter table statement before loading the data and use the same statement to reenable generation of redo data after the load has completed. Remember to back up the database files after the load completes. An example is given below.

```
SQL> alter table customer nologging;
Table altered.
SQL>
< -- Load data into table -- >
SQL> alter table customer logging;
Table altered.
```

Presorting the data according to index columns and loading with the SORTED INDEX option can also save time if indexes already exist.

Use direct path loading to bypass SQL processing and buffer cache access (the direct=true parameter). Note that direct path loading imposes some restrictions, such as requiring exclusive access to the table during load; check any restrictions for your version of Oracle before attempting to use the direct path load feature.

Use load parallelism if possible to shorten the load time. The load commands for a two-way parallel load are shown in the following example:

```
sqlldr userid=id/pwd control=c1.ctl direct=true parallel=true
sqlldr userid=id/pwd control=c2.ctl direct=true parallel=true
```

Note that each load session requires its own control file.

Analyze the Database Tables

This step is important to allow the optimizer to recognize data skew. Analyze all columns that are likely to be queried (but not comment or description columns unless you know they will be queried). Analyze commands are shown in the following examples:

```
analyze table item compute statistics;
analyze table stock estimate statistics sample 15000 rows;
```

The second example illustrates a command that might be appropriate for very large tables in which the distribution of data in a subset of the table can be expected to represent the overall data distribution.

Note that the DBMS_STATS package can also be used to gather statistics.

Create the Indexes

Index creation is fastest if done after the tables have been loaded rather than during load. Index creation is also 10% to 15% faster after the tables have been analyzed. Drop indexes before the load to ensure that index creation does not take place during the load.

Use the parallel option when creating indexes if multiple CPUs are available. For example:

```
create index custname_idx on customer (cust_name)
        parallel (degree 8);
```

Analyze the Indexes

The final step is to analyze the indexes for the benefit of the optimizer. Examples of the `analyze index` command are shown below.

```
analyze index w_idx compute statistics;
analyze index d_idx compute statistics;
analyze index custname_idx estimate statistics sample 2 percent;
```

Note that the `DBMS_STATS` package can also be used to gather index statistics, and that the `compute statistics` keywords can be applied to the `create index` statement, eliminating the need for a separate analyze step.

Planning for Indexes

The primary key of a table normally has an associated index. Apart from that, deciding when to create an index is not always easy. Here are some general guidelines:

- If between one-half of 1% and 2% or more of the data in a table will be retrieved as a result of a query predicate, a table scan will be faster than retrieving the data from the base table with an index. So in this case you may be better off without an index, since an optimizer will often choose to use an index if one is present.
- If you know which columns are usually accessed in a table, consider a concatenated index to bypass the base table.
- If a query is stable and frequently run, consider the materialized view feature in Oracle8i.

Using an SGA Larger Than 2 Gbytes

Some applications need all the shared memory they can get, especially in high-throughput environments where extra memory in the buffer cache can benefit performance significantly. With 32-bit versions of Oracle, though, you will need to change the base address of the Oracle shared memory segment (`sgabeg`) and relink the `oracle` binary before you can use an SGA larger

than 2 Gbytes. The method varies slightly with different versions of Oracle; the procedure for Oracle8 and later releases is documented in this section.

The standard `oracle` binary ships with `sgabeg = 0x80000000`, which allows a little less than 2 Gbytes of SGA shared memory. To allow for an SGA of around 3.75 Gbytes in size, lower `sgabeg` to `0x08000000`.

The procedure outlined below will rebuild the `oracle` binary with `sgabeg` set to `0x08000000`:

```
oracle$ nm -P $ORACLE_HOME/bin/oracle | grep sgabeg
sgabeg     n 80000000       0
oracle$ cd $ORACLE_HOME/rdbms/lib
oracle$ genksms -s 0x08000000 > ksms.s
oracle$ make -f ins_rdbms.mk ksms.o
oracle$ make -f ins_rdbms.mk ioracle
<< various messages deleted >>
oracle$ nm -P $ORACLE_HOME/bin/oracle | grep sgabeg
sgabeg     n  8000000       0
oracle$
```

The `make -f ins_rdbms.mk ksms.o` line is only needed for some older Oracle8 releases, but it does no harm on later releases, including Oracle9i.

To allow for the maximum possible shared memory segment size, ensure that the `shmmax` parameter in `/etc/system` is set high enough. Use the following line, which sets `shmmax` to the largest possible setting for 32-bit databases:

```
set shmsys:shminfo_shmmax = 0xffffffff
```

For 64-bit versions of Oracle, set `shmmax` as shown in the following line:

```
set shmsys:shminfo_shmmax = 0xffffffffff
```

When increasing the size of the SGA, remember to leave enough room for your applications so that the system does not page. Any performance benefits from a large buffer cache will be more than outweighed by the negative performance impact of paging. Refer to "STEP 1. Monitoring Memory" on page 282 of *Configuring and Tuning Databases on the Solaris Platform* for information about how to recognize memory paging.

Reconfiguring Oracle9i Dynamically

Dynamic Reconfiguration (DR) support in Solaris and on Sun hardware is described under "Dynamic Reconfiguration" on page 26 of *Configuring and Tuning Databases on the Solaris Platform*, along with a discussion on its implications for databases.

Oracle9i Dynamic System Global Area

Oracle's Dynamic System Global Area (SGA) capability, released with Oracle9i, for the first time offers the capability of altering the size of the SGA

while the database is live. In particular, Oracle9i supports changes to the sizes of the buffer cache and shared pool, the two major components of the SGA. When intimate shared memory (ISM) is used for the SGA, the amount of memory allocated to the buffer cache and the shared pool can be traded off against each other, although the total amount of memory allocated to the SGA cannot change. For example, the buffer cache can first be reduced in size, and then the shared pool increased by the same amount (and vice versa). When Dynamic ISM (DISM) is used for the SGA, the total size of the SGA can be changed dynamically.

After connecting to Oracle as sysdba, you can change the buffer cache and shared pool with the following statements:

```
alter system set db_cache_size = new_size_in_bytes;
alter system set shared_pool_size = new_size_in_bytes;
```

The db_cache_size tunable parameter is the recommended way of setting the size of the buffer cache. The old db_block_buffers parameter is still supported for backward compatibility, but it cannot be used if dynamic SGA capabilities are required.

Oracle9i also introduces a new optional init.ora parameter: sga_max_size. If sga_max_size is not set, Oracle will calculate it automatically in accordance with the user-supplied settings for shared_pool_size, either db_cache_size or db_block_buffers, and other SGA memory requirements. More significantly, on Solaris, if sga_max_size is *not* set, then ISM rather than DISM will be used for shared memory attaches. ISM will also be used if sga_max_size is set less than or equal to the total of db_cache_size and shared_pool_size.

How Oracle Chooses Between ISM and DISM

When the sga_max_size parameter is set larger than the total of db_cache_size, shared_pool_size, and the other smaller SGA components used by Oracle, Oracle9i automatically uses DISM. In this case, the amount of memory initially locked will be equal to db_cache_size plus shared_pool_size plus the other SGA components. When the buffer cache or shared pool is resized later, Oracle will either lock additional memory or unlock and release existing memory, depending on whether the change results in an increase or decrease in size.

Oracle also makes available a sample Reconfiguration Coordination Manager (RCM) script to allow the buffer cache and shared pool to be automatically resized if a DR event takes place. A sample script is included on the book website. Database administrators should modify the script according to local requirements. For example, if a DR operation that will remove 2 Gbytes of memory is attempted, the script should decide which instances should be adjusted and by how much.

Oracle9i program global area (PGA) memory can also be resized. This memory is local to each Oracle process rather than allocated as shared memory, so neither ISM nor DISM is used for PGA memory.

The Benefits of Using Dynamic SGA

When memory must be removed from a running system, a DR operation can be attempted. The DR event may fail, though, for example, if locked memory cannot be relocated onto other system boards. Consequently, there are times when the only alternative to dynamic SGA resizing is to disconnect all users from the database, shut it down, modify the SGA size, restart the database, and reconnect the users. Application availability is clearly compromised without the flexibility to resize SGA segments. For this reason, Oracle9i represents a significant leap forward in database availability.

Two scenarios can be considered to illustrate the impact of DR events on the database administrator. The first scenario covers the case where no RCM script has been configured to allow automatic reconfiguration of the SGA. If memory is to be removed, the buffer cache or shared pool must first be resized with the appropriate `alter system` commands. Once the resizing is done, the system board can be removed. If memory has been increased by addition of a system board, the buffer cache and the shared pool can be increased at any time with a suitable `alter system` command.

The second scenario covers the case where an RCM script has been configured to automatically issue the `alter system` commands in response to memory changes in the system. In this case, the DR event can take place without operator intervention.

In summary, Oracle9i effectively leverages the DR and RCM features in Solaris 8 (starting from the 4/01 release) to provide a significant boost to application availability.

Recovering Oracle

When you configure Oracle, it is wise to take into account recovery time as well as performance. In this section we focus on crash recovery, rather than database restore and roll-forward recovery after media failure.

Crash recovery should not happen often, but when it does you'll probably be grateful for any steps you took with a view to reducing the recovery time. In this section we explore Oracle tunables that relate to checkpoints and recovery and consider the trade-off between performance and recovery time. Checkpoints are described briefly in "Pagecleaners" on page 55 of *Configuring and Tuning Databases on the Solaris Platform*, and crash recovery is described in "Database Recovery Process", also on page 55.

Checkpoints and recovery time are an issue in environments where insert, update, and delete database operations feature prominently, so the discussion that follows has less relevance to read-only and read-mostly environ-

ments. OLTP workloads are typically good candidates for consideration of the implications of crash recovery.

The Influence of Checkpoints on Recovery Time

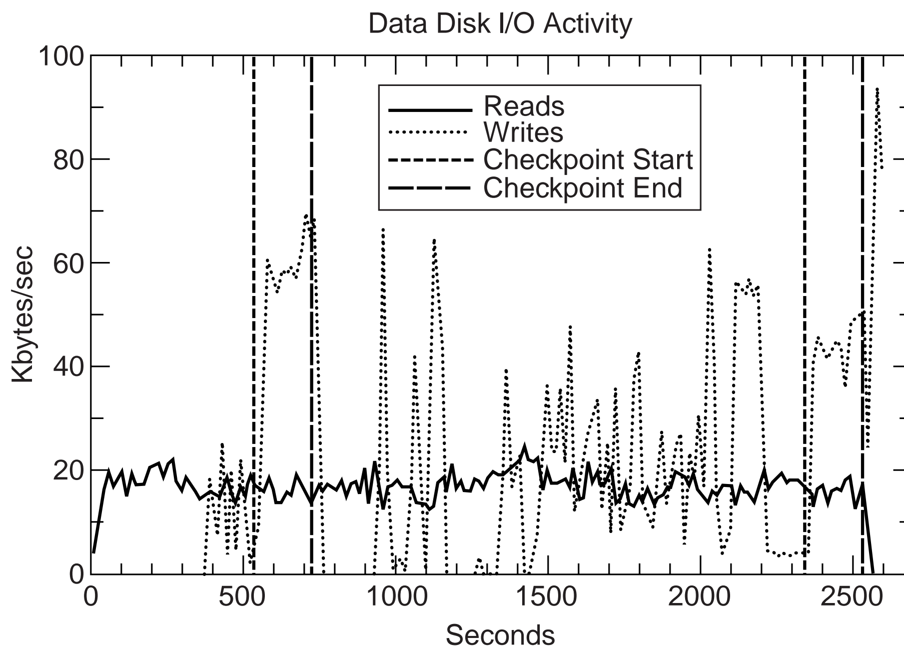
Recovery time is influenced significantly by checkpoint frequency. Oracle checkpoints are triggered when one of the following events takes place:

- An online redo log fills, causing a log file switch to occur.
- The checkpoint interval expires.
- A database administrator manually requests a checkpoint (`alter system checkpoint`) or redo log switch (`alter system switch logfile`).

Recovery is fastest when a system crash occurs immediately after a checkpoint, since all modified pages will have been flushed to disk. The longer the interval between a checkpoint and a crash, the more work will be required to recover the database.

Oracle does not wait until a checkpoint to flush its dirty buffers, so the recovery time is not directly proportional to the interval between a checkpoint and a crash. The graph in Figure 4 shows read and write I/O activity for an Oracle8.1.5 data disk with checkpoints near the start and end of the monitoring period.

Figure 4 *Data disk activity during and between checkpoints*



The graph shows that writes to disk continue sporadically between checkpoints. This write activity is needed to maintain a supply of clean buffers for applications as well as to reduce recovery time.

The Influence of Checkpoints on Performance

Checkpoint frequency affects system performance as well as recovery time. Table 1 shows results from a series of tests that measured performance and recovery time as the checkpoint interval was varied. By “checkpoint interval,” I mean the period of time between the start of successive checkpoints. Checkpoints were initiated manually with the `alter system checkpoint` command as `sysdba`.

Table 1 *Checkpoint interval, performance, and crash recovery time*

Checkpoint Interval (secs)	Checkpoint Interval Relative to Baseline	Transaction Throughput Relative to Baseline	Recovery Time (secs)	Recovery Time Relative to Baseline
150	1	1.00	252	1.0
300	2	1.06	242	1.0
600	4	1.19	646	2.6
900	6	1.27	963	3.8

Some of the conclusions that can be drawn from these results are listed below.

- Transaction throughput for the workload improves as the checkpoint interval increases. Throughput increases by 27% if checkpoints are only run every 900 seconds instead of every 150 seconds. Further increases in the checkpoint time beyond 900 seconds resulted only in minor throughput improvements.

  The reason for the performance improvement is that dirty buffers are flushed less frequently, freeing CPU cycles to do application work. As expected, monitoring shows that the average number of I/Os per transaction drops with longer checkpoint intervals, as does the average number of CPU cycles consumed per transaction.

- Recovery time does not increase as much as you might expect when checkpoints are run less frequently. For example, quadrupling the checkpoint time does not quadruple the recovery time. Nonetheless, compared to 150-second checkpoints, the recovery time more than doubles with 600-second checkpoints and more than triples with 900-second checkpoints.

CPU utilization during recovery for the four tests ranged between 42% and 45% on a four-CPU server, comprising 7% to 8% user activity and 35% to 38%

system activity. The high system utilization reflects the heavy I/O activity during recovery.

The v$instance_recovery view

Oracle8i introduced a new `v$` view, the `v$instance_recovery` view. This view offers an estimate of how long it would take to complete crash recovery if the database crashed at the moment of the query.

The following example from an Oracle8.1.7 instance shows the estimated number of I/Os that would be required to recover the instance following a crash.

```
SQL> select recovery_estimated_ios from v$instance_recovery;

RECOVERY_ESTIMATED_IOS
----------------------
                814540
```

The `recovery_estimated_ios` statistic is actually a measure of the number of dirty buffers in the buffer cache.

The number of I/Os required for recovery goes only partway toward answering the question. Most database administrators will be more interested to know how long it will take to recover. Database Engineering carried out tests to answer this question by deliberately crashing a database immediately after querying the `v$instance_recovery` view. Table 2 shows the results of the tests.

Table 2 *Recovery estimated I/Os and recovery time*

Recovery Estimated I/Os	Actual Recovery Time (secs)	Estimated I/Os per Recovery Time
1,500,160	1,802	833
868,745	1,011	859
814,540	981	830
284,210	340	836
187,520	249	753

The tests showed that for this instance, the `recovery_estimated_ios` statistic from the `v$instance_recovery` view could be divided by approximately 800 to estimate the recovery time in seconds. The results will differ in other environments because of differences in the hardware configuration, the Oracle version, and other local differences.

Oracle8i also introduced an `init.ora` parameter related to the `v$instance_recovery` view. The `fast_start_io_target` parameter, available only in Oracle8i Enterprise Edition, allows a database administrator to set a high-water mark for the number of I/Os that will be required to carry out instance recovery. Once this mark is reached, the Database Writers will begin flushing dirty buffers without waiting for a checkpoint. If the default value of `0` is used, the parameter will be ignored. The

db_block_max_dirty_target fulfills a similar function in Oracle Standard Edition.

In one Database Engineering test, fast_start_io_target was set to a value equivalent to 200 Kbytes. The result was a 27% hit in throughput, but recovery finished 20 times faster compared to recovery after no recent checkpoints and without this parameter set.

In Oracle9i, another more useful metric was added to the v$instance_recovery view: estimated_mttr, the estimated mean time to recover (MTTR) in seconds. An example from an environment running Oracle9.0.1 is shown below:

```
SQL> select estimated_mttr from v$instance_recovery;

ESTIMATED_MTTR
--------------
          1714
```

Oracle9i also introduced a new init.ora parameter to complement the new v$instance_recovery column: fast_start_mttr_target, the maximum desired recovery time, in seconds. This new parameter allows the database administrator to set the approximate maximum recovery time that is acceptable for this instance.

Once this high-water mark is reached, the Database Writers will begin flushing dirty buffers. The default value of 0 means that the parameter is ignored. If the fast_start_mttr_target parameter is set, then refer to the target_mttr column instead of estimated_mttr in the v$instance_recovery view (that is, use select target_mttr from v$instance_recovery). Note that the db_block_max_dirty_target init.ora parameter in Oracle8i has been converted to an underscore (hidden) parameter in Oracle9i (_db_block_max_dirty_target). For more information on hidden parameters, including caveats, refer to "Viewing and Changing Hidden Parameters" on page 5.

Other Parameters Influencing Recovery

Apart from the parameters already discussed, the main init.ora parameters that influence recovery are outlined in the following list:

- **log_checkpoint_interval:** The number of blocks in the redo log file that will be written before a checkpoint is triggered. This parameter has no effect if it is set larger than the size of a redo log file, since every log switch automatically triggers a checkpoint. If redo log files are sized appropriately, this parameter can simply be set to a very high number and ignored. Note that redo log files should all be set to the same size to ensure consistency in checkpoint frequency.
- **log_checkpoint_timeout:** The maximum number of seconds that should expire before a checkpoint is triggered. If the default value of 0 is used, this parameter will be ignored. In read-mostly environments this

parameter can be used to ensure that checkpoints do happen regularly, even if the write activity is not sufficient to trigger frequent redo log switches.

Carefully sizing redo log files offers a good way of managing the checkpoint interval. Set the `log_checkpoints_to_alert` parameter to `true` to allow Oracle to report redo log switches in the `alert.log` file. You can monitor redo activity by running the following command as `sysdba`:

```
SQL> select name, value from v$sysstat
  2  where name = 'redo blocks written';

NAME                                                                  VALUE
---------------------------------------------------------------- ----------
redo blocks written                                                 1489089
```

The value reported is cumulative, so you will need to run the command multiple times and calculate the difference.